A Process Calculus Approach to Correctness Enforcement of PLCs*

Ruggero Lanotte[1], Massimo Merro[2], and Andrei Munteanu[2]

[1] Università degli Studi dell'Insubria, Como, Italy
[2] Università degli Studi di Verona, Verona, Italy

Abstract. We define a simple process calculus, based on Hennessy and Regan's *Timed Process Language*, for specifying networks of communicating *programmable logic controllers* (PLCs) enriched with monitors enforcing specifications compliance. We define a synthesis algorithm that given an uncorrupted PLC returns a monitor that enforces the correctness of the PLC, even when injected with *malware* that may forge/drop actuator commands and inter-controller communications. Then, we strengthen the capabilities of our monitors by allowing the insertion of actions to *mitigate* malware activities. This gives us *deadlock-freedom monitoring*: malware may not drag monitored controllers into deadlock states.

Keywords: Process calculus · PLC correctness · Runtime enforcement.

1 Introduction

Industrial Control System (ICSs) are distributed systems controlling physical processes via *programmable logic controllers* (PLCs) connected to *sensors* and *actuators*. PLCs have an ad-hoc architecture to execute simple processes known as *scan cycles*. Each scan cycle consists of three phases: (i) reading of the *sensor measurements* of the physical process; (ii) derivation of the commands to guide the evolution of the physical process; (iii) transmission of the calculated *commands* to the *actuator devices*.

Published scan data show how thousands of PLCs are directly accessible from the Internet [27]. When this is not the case, PLCs are often connected to each other in *field communications networks*, opening the way to the spreading of worms such as the PLC-Blaster worm [29] or the PLC PIN Control attack [2].

As a consequence, extra *trusted hardware components* have been proposed to enhance the security of ICS architectures [24,25]. In this respect, McLaughlin [24] proposed to add a policy-based *enforcement mechanism* to mediate the actuator commands transmitted by the PLC to the physical plant, whereas Mohan et al. [25] introduced an architecture in which every PLC runs under the scrutiny of a *monitor* which looks for deviations with respect to *safe behaviours*; if the

behaviour of the PLC is not as expected then the control passes to a *safety controller* which maintains the plant within the required safety margins.

Both architectures above have been validated by means of simulation-based techniques. However, as far as we know, formal methodologies have not been used yet to model and formally verify security-oriented architectures for ICSs.

The *goal* of the paper is to verify the effectiveness of a *process calculus approach* to formalise *runtime enforcement* of specification compliance in networks of PLCs injected with *colluding malware* that may forge/drop both actuator commands and inter-controller communications[3]. *Process calculi* represent a successful and widespread formal approach in *concurrency theory* relying on a variety of *behavioural equivalences* (*e.g.*, trace equivalence and bisimilarity) for studying complex systems, such as distributed migrating systems [6,10], IoT systems [13,15] and cyber-physical systems [14,19], and used in many security fields, including verification of security protocols [1,21] and security analysis of *cyber-physical attacks* [18]. On the other hand, *runtime enforcement* [28,20,8] is a powerful verification/validation technique aiming at correcting possibly-incorrect executions of a system-under-scrutiny (SuS) via a kind of monitor that acts as a *proxy* between the SuS and its environment.

Thus, we propose to synthesise a proxy from an uncorrupted PLC, to form a *monitored PLC* ensuring: 1. *observation-based monitoring*, *i.e.*, the proxy should only look at the observables of the PLC, and not at its internal execution; 2. *transparency*, *i.e.*, the semantics of the monitored PLC must not differ from the semantics of the genuine (*i.e.*, uncorrupted) PLC; 3. *sound execution* of the monitored PLC, to prevent incorrect executions; 4. *deadlock-freedom*, *i.e.*, an injected malware may not drag a monitored PLC into a deadlock state.

Obviously, if the PLC is compromised then its correct execution can only be enforced with the help of an extra component, a *secured proxy*, as advocated by McLaughlin [24] and Mohan et al. [25]. This means that any implementation of our proposed proxy should be *bug-free* to deal with possible infiltrations of malware. This may seem like we just moved the problem over to securing the proxy. However, this is not the case because the proxy only needs to enforce correctness, while the PLC controls its physical process relying on malware-prone communications via the Internet or the USB ports. Of course, by no means runtime reconfigurations of the secure proxy should be allowed.

Contribution. We define a simple timed process calculus, based on Hennessy and Regan's TPL [11], for specifying networks of communicating monitored controllers, possibly injected with *colluding malware* that may forge/drop both actuator commands and inter-controller communications. Monitors are formalised in terms of a sub-class of finite-state Ligatti et al.'s *edit automata* [20]. A network composed of n PLCs Ctrl_i, running in parallel, each of which injected with a malware Malw_i, and enforced by a monitor Mon_i, is represented as:

$$\mathsf{Mon}_1 \vdash \{\mathrm{Ctrl}_1 \mid \mathrm{Malw}_1\} \parallel \ldots \parallel \mathsf{Mon}_n \vdash \{\mathrm{Ctrl}_n \mid \mathrm{Malw}_n\}\,.$$

[3] We do not deal with alterations of sensor signals within a PLC, as they can already occur either at the network level or within the sensor devices [9].

Here, the parallel process $\mathrm{Ctrl}_i \mid \mathrm{Malw}_i$ is a formal abstraction of the sequential execution of the PLC code Ctrl_i injected with the malware Malw_i.

Then, we propose a *synthesis function* $[\![-]\!]$ that, given an uncorrupted (deterministic) PLC Ctrl returns, in *polynomial time*, a *syntactically deterministic* [3] edit automaton $[\![\mathrm{Ctrl}]\!]$ to form a monitored PLC that ensures: *observation-based monitoring*, *transparency*, and *sound execution* of the monitored PLC. These properties can be expressed with a single algebraic equation:

$$\prod_{i=1}^{n} [\![\mathrm{Ctrl}_i]\!] \vdash \{\mathrm{Ctrl}_i \mid \mathrm{Malw}_i\} \;\simeq\; \prod_{i=1}^{n} \mathsf{go} \vdash \{\mathrm{Ctrl}_i\} \tag{1}$$

for arbitrary malware Malw_i, where $\simeq$ denotes *trace equivalence* and go is the monitor that allows any action. Here, intuitively, each monitor $[\![\mathrm{Ctrl}_i]\!]$ prevents incorrect executions of the compromised controller $\mathrm{Ctrl}_i \mid \mathrm{Malw}_i$.

However, our monitors do not protect against malware that may drag a monitored PLC into a deadlock state. In fact, Equation 1 does not hold with respect to *weak bisimilarity*, which is a notoriously *deadlock-sensitive* semantic equivalence. Thus, in order to achieve deadlock-freedom we equip our monitors with the semantic capability to *mitigate* those malicious activities that may deadlock the controller. In practice, our monitors will be able to *insert* actions, *i.e.*, to emit correct actions in full autonomy to complete scan cycles. The enforcement resulting from the introduction of mitigation allows us to recover *deadlock-freedom monitoring* by proving Equation 1 with respect to weak bisimilarity.

Outline. Section 2 defines our process calculus to express monitored controllers injected with malware. Section 3 defines an algorithm to synthesise our monitors. Section 4 introduces mitigation to recover deadlock-freedom. Section 5 draws conclusions and discusses related work. In this extended abstract, proofs are omitted; full proofs can be found in the full version of this paper [16].

2 A timed process calculus for monitored PLCs

We define our process calculus as an extension of Hennessy and Regan's TPL [11].

Let us start with some preliminary notation. We use $s, s_k \in \mathsf{Sens}$ for *sensor signals*, $a, a_k \in \mathsf{Act}$ for *actuator commands*, and $c, c_k \in \mathsf{Chn}$ for *channel names*.

Controller. In our setting, controllers are nondeterministic sequential timed processes evolving through three different phases: *sensing* of sensor signals, *communication* with other controllers, and *actuation*. For convenience, we use four different syntactic categories to distinguish the four main states of a controller: $\mathbb{C}\mathtt{trl}$ for initial states, $\mathbb{S}\mathtt{ens}$ for sensing states, $\mathbb{C}\mathtt{om}$ for communication states, and $\mathbb{A}\mathtt{ct}$ for actuation states. In its initial state, a controller is a recursive process starting its scan cycle in the *sensing phase*:

$$\mathbb{C}\mathtt{trl} \ni P \;::=\; \mathsf{rec}\,\mathsf{X}.S$$

Table 1. LTS for controllers

$$\text{(Rec)}\ \frac{S\{^{\mathsf{rec}\,\mathsf{X}.S}/_{\mathsf{X}}\} \xrightarrow{\alpha} S'}{\mathsf{rec}\,\mathsf{X}.S \xrightarrow{\alpha} S'} \qquad \text{(TimeS)}\ \frac{-}{\mathsf{tick}.S \xrightarrow{\mathsf{tick}} S}$$

$$\text{(ReadS)}\ \frac{j \in I}{\lfloor \sum_{i\in I} s_i.S_i \rfloor S \xrightarrow{s_j} S_j} \qquad \text{(TimeoutS)}\ \frac{-}{\lfloor \sum_{i\in I} s_i.S_i \rfloor S \xrightarrow{\mathsf{tick}} S}$$

$$\text{(InC)}\ \frac{j \in I}{\lfloor \sum_{i\in I} c_i.C_i \rfloor C \xrightarrow{c_j} C_j} \qquad \text{(TimeoutInC)}\ \frac{-}{\lfloor \sum_{i\in I} c_i.C_i \rfloor C \xrightarrow{\mathsf{tick}} C}$$

$$\text{(OutC)}\ \frac{-}{\lfloor \overline{c}.C \rfloor C' \xrightarrow{\overline{c}} C} \qquad \text{(TimeoutOutC)}\ \frac{-}{\lfloor \overline{c}.C \rfloor C' \xrightarrow{\mathsf{tick}} C'}$$

$$\text{(WriteA)}\ \frac{-}{\overline{a}.A \xrightarrow{\overline{a}} A} \qquad \text{(End)}\ \frac{-}{\mathsf{end}.P \xrightarrow{\mathsf{end}} P}$$

Notice that due to the cyclic behaviour of controllers, the process variable X may syntactically occur only in the last phase, actuation. We assume *time guarded* recursion to avoid undesired *zeno behaviours*. Intuitively, in time guarded recursion the process variable must occur prefixed by at least one timed action tick.

During the sensing phase, the controller waits for a *finite* number of admissible sensor signals. If none of those signals arrives in the current time slot then the controller will *timeout* moving to the following time slot (we adopt the TPL construct $\lfloor \cdot \rfloor \cdot$ for timeout). The controller may also sleep for a while, waiting for sensor signals to become stable. The syntax is the following:

$$\mathbb{Sens} \ni S \ ::= \ \lfloor \textstyle\sum_{i\in I} s_i.S_i \rfloor S \ \mid \ \mathsf{tick}.S \ \mid \ C$$

Once the sensing phase is concluded, the controller starts its *calculations* that may depend on *communications* with other controllers. Controllers communicate to each other for mainly two reasons: either to receive notice about the state of other physical sub-processes or to require an actuation on a different physical process that will have an influence on the physical process governed by the controller. We adopt a *channel-based handshake point-to-point* communication paradigm. Notice that, in order to avoid starvation, the communication is always under timeout. The syntax for the communications phase is:

$$\mathbb{Comm} \ni C \ ::= \ \lfloor \textstyle\sum_{i\in I} c_i.C_i \rfloor C \ \mid \ \lfloor \overline{c}.C \rfloor C \ \mid \ A$$

Thus, our controllers can either listen on a *finite* number of communication channels or transmit on specific channels to pass some local information.

Finally, in the *actuation phase* the controller eventually transmits a *finite* sequence of commands to a number of different actuators, and then, it emits a special signal end to denote the end of the scan cycle. After that, it restarts its cycle in the sensing phase via a recursive call denoted with a process variable X. In order to ensure semantics closure, we also have a construct $\mathsf{end}.P$ which will be only generated at runtime but never used to write PLC programs.

$$\mathbb{Act} \ni A \ ::= \ \overline{a}.A \ \mid \ \mathsf{end}.\mathsf{X} \ \mid \ \mathsf{end}.P$$

Table 2. LTS for malware code

$$\text{(Malware)}\ \frac{j \in I}{\lfloor \sum_{i\in I} \mu_i.M_i \rfloor M \xrightarrow{\mu_j} M_j} \qquad \text{(TimeoutM)}\ \frac{-}{\lfloor \sum_{i\in I} \mu_i.M_i \rfloor M \xrightarrow{\mathsf{tick}} M}$$

$$\text{(RecM)}\ \frac{M\{^{\mathsf{rec}\,\mathsf{X}.M}/_{\mathsf{X}}\} \xrightarrow{\alpha} M'}{\mathsf{rec}\,\mathsf{X}.M \xrightarrow{\alpha} M'} \qquad \text{(TimeM)}\ \frac{-}{\mathsf{tick}.M \xrightarrow{\mathsf{tick}} M} \qquad \text{(TimeNil)}\ \frac{-}{\mathsf{nil} \xrightarrow{\mathsf{tick}} \mathsf{nil}}$$

Remark 1 (Scan cycle duration and maximum cycle limit). Notice that any scan cycle of a PLC must be completed within a *maximum cycle limit* which depends on the controlled physical process; if this time limit is violated the PLC stops and throws an exception [29]. Thus, the signal end must occur well before the *maximum cycle limit*. We assume that our PLCs successfully complete their scan cycle in less than half of the maximum cycle limit.

The operational semantics of controllers is given in Table 1. In the following, we use the metavariables α and β to range over the set of possible actions: $\{s, \overline{a}, a, \overline{c}, c, \tau, \mathsf{tick}, \mathsf{end}\}$. These actions denote: sensor readings, actuator commands, drops of actuator commands, channel transmissions, channel receptions/drops, internal actions, passage of time, end of a scan cycle, respectively.

Malware. Let us provide a formalisation of the malware code that we assume may be injected in a controller to compromise its runtime behaviour. The kind of malware we wish to deal with may perform the following malicious activities: (i) forging fake channel transmissions towards other controllers (via actions $\overline{c}$); (ii) dropping incoming communications from other controllers (via actions c); (iii) forging fake actuator commands (via actions $\overline{a}$); (iv) dropping actuator commands launched by the controller (via actions a).

The formal syntax of the admitted malware is the following:

$$\mathbb{M}\mathbb{a}\mathbb{l}\mathbb{w} \ni M \ ::= \ \lfloor \textstyle\sum_{i\in I} \mu_i.M_i \rfloor M \ \mid \ \mathsf{rec}\,\mathsf{X}.M \ \mid \ \mathsf{X} \ \mid \ \mathsf{tick}.M \ \mid \ \mathsf{nil}$$

where the prefixes $\mu_i \in \{\overline{c}, c, \overline{a}, a\}$, for $i \in I$, denote the possible malicious actions mentioned above. Again, we assume *time guarded* recursion to avoid undesired *zeno behaviours*. A straightforward operational semantics is given in Table 2.

Compromised controller. In our setting, a compromised controller may potentially run in parallel with an arbitrary piece of malware. The syntax is:

$$\begin{aligned} Z \ &::= \ P \ \mid \ S \ \mid \ C \ \mid \ A \\ \mathbb{C}\mathbb{C}\mathbb{r}\mathbb{t}\mathbb{l} \ni J \ &::= \ Z \ \mid \ Z \mid M \end{aligned}$$

where $Z \in \mathbb{C}\mathbb{t}\mathbb{r}\mathbb{l} \cup \mathbb{S}\mathbb{e}\mathbb{n}\mathbb{s} \cup \mathbb{C}\mathbb{o}\mathbb{m}\mathbb{m} \cup \mathbb{A}\mathbb{c}\mathbb{t}$ denotes a controller in an arbitrary state, and $\mid$ is the standard process algebra construct for parallel composition.

The operational semantics of a compromised controller is given by the transition rules of Table 3. Rule (Ctrl) models the genuine behaviour of the controller even in the presence of the malware. Rule (Inject) denotes the injection of a

Table 3. LTS for compromised controllers

$$\text{(Ctrl)}\ \frac{Z \xrightarrow{\alpha} Z' \quad \alpha \neq \mathsf{tick}}{Z \mid M \xrightarrow{\alpha} Z' \mid M} \qquad \text{(Inject)}\ \frac{M \xrightarrow{\alpha} M' \quad \alpha \notin \{\mathsf{tick}, a\}}{Z \mid M \xrightarrow{\alpha} Z \mid M'}$$

$$\text{(DropAct)}\ \frac{Z \xrightarrow{\overline{a}} Z' \ \ M \xrightarrow{a} M'}{Z \mid M \xrightarrow{\tau} Z' \mid M'} \qquad \text{(TimePar)}\ \frac{Z \xrightarrow{\mathsf{tick}} Z' \ \ M \xrightarrow{\mathsf{tick}} M'}{Z \mid M \xrightarrow{\mathsf{tick}} Z' \mid M'}$$

malicious fabricated action. Rule (DropAct) models the drop of an actuator command $\overline{a}$; thus, the command $\overline{a}$ never reaches its intended actuator device. Rule (TimePar) models *time synchronisation* between the controller and the malware.

We recall that recursion processes in a malware are always time guarded; thus, a malware can never inject zeno behaviours preventing the passage of time.

Remark 2 (Attacks on channels). Notice that injection/drop on communication channels affects the interaction between controllers and not within them. For this reason, we do not have a rule for channels similar to (DropAct). Inter-controller malicious activities on communication channels will be prevented by the monitor.

Monitored controller(s). The core of our runtime enforcement relies on a (recursive) timed variant of Ligatti et al.'s *edit automata* [20], *i.e.*, a particular class of automata specifically designed to modify/suppress/insert actions in a generic system in order to preserve its correct behaviour. Their syntax follows:

$$\mathbb{E}\mathrm{dit} \ni \mathsf{E} \ ::= \ \mathsf{go} \ \mid \ \textstyle\sum_{i\in I} \alpha_i/\beta_i.\mathsf{E}_i \ \mid \ \mathsf{rec}\,\mathsf{X}.\mathsf{E} \ \mid \ \mathsf{X}$$

Intuitively, the automaton go will admit any action of the monitored system, while the edit automaton $\sum_{i\in I} \alpha_i/\beta_i.\mathsf{E}_i$ *replaces* actions α_i with β_i, and then continues as E_i, for any $i \in I$, with I finite. The operational semantics of our edit automata is the following:

$$\text{(Go)}\ \frac{-}{\mathsf{go} \xrightarrow{\alpha/\alpha} \mathsf{go}} \qquad \text{(Edit)}\ \frac{j \in I}{\sum_{i\in I} \alpha_i/\beta_i.\mathsf{E}_i \xrightarrow{\alpha_j/\beta_j} \mathsf{E}_j} \qquad \text{(recE)}\ \frac{\mathsf{E}\{^{\mathsf{rec}\,\mathsf{X}.\mathsf{E}}/_{\mathsf{X}}\} \xrightarrow{\alpha/\beta} \mathsf{E}'}{\mathsf{rec}\,\mathsf{X}.\mathsf{E} \xrightarrow{\alpha/\beta} \mathsf{E}'}$$

When an edit automaton performs a transition labeled α/β, with $\alpha \neq \tau$ and $\beta = \tau$, we say that the automaton *suppresses* the observable action α.

Our *monitored controllers*, written $\mathsf{E}\vdash\{\!|J|\!\}$, are constituted by a (potentially) compromised controller J and an edit automaton E enforcing the behaviour of J according to the following transition rule for correction/suppression:

$$\text{(Enforce)}\ \frac{J \xrightarrow{\alpha} J' \quad \mathsf{E} \xrightarrow{\alpha/\beta} \mathsf{E}'}{\mathsf{E}\vdash\{\!|J|\!\} \xrightarrow{\beta} \mathsf{E}'\vdash\{\!|J'|\!\}}.$$

In a monitored controller $\mathsf{E}\vdash\{\!|J|\!\}$ with no malware inside, the enforcement never occurs, *i.e.*, in rule (Enforce) we always have $\alpha = \beta$, and the two components E and J evolve in a tethered fashion, moving through related correct states.

Table 4. LTS for monitored field communications networks

$$\text{(ParL)}\ \frac{N_1 \xrightarrow{\alpha} N_1'}{N_1 \parallel N_2 \xrightarrow{\alpha} N_1' \parallel N_2} \qquad \text{(ParR)}\ \frac{N_2 \xrightarrow{\alpha} N_2'}{N_1 \parallel N_2 \xrightarrow{\alpha} N_1 \parallel N_2'}$$

$$\text{(ChnSync)}\ \frac{N_1 \xrightarrow{c} N_1' \quad N_2 \xrightarrow{\overline{c}} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

$$\text{(TimeSync)}\ \frac{N_1 \xrightarrow{\mathsf{tick}} N_1' \quad N_2 \xrightarrow{\mathsf{tick}} N_2' \quad N_1 \parallel N_2 \xrightarrow{\tau} \not\rightarrow}{N_1 \parallel N_2 \xrightarrow{\mathsf{tick}} N_1' \parallel N_2'}$$

Obviously, we can easily generalise the concept of monitored controller to a *field communications network* of communicating monitored controllers, each one acting on different actuators. These networks are defined via the grammar:

$$\mathbb{FN}\mathbb{et} \ni N \ ::= \ \mathsf{E} \vdash \{J\} \ \mid \ N \parallel N$$

and described via the operational semantics given in Table 4. Notice that monitored controllers may interact with each other via channel communication. Moreover, they may evolve in time when no communication occurs (we recall that neither controllers nor malware admit zeno behaviours). This ensures us *maximal progress* [11], a desirable time property when modelling real-time systems: communications are never postponed to future time slots.

Behavioural equalities. In the paper, we adopt standard behavioural equivalences between (networks of) monitored controllers. In particular, we use *trace equivalence*, written $\simeq$, *weak similarity*, denoted $\sqsubseteq$, and *weak bisimilarity*, written $\approx$.

2.1 Use case: a small water-tank system

In this section, we specify the controller of a simple water-tank system depicted in Figure 1. Basically, in this system the water is pumped into the tank via a *pump*. Furthermore, a *valve* connects the tank with an *external unit* which is not represented. Here, we assume that the incoming water flow is lower than the out-coming flow passing through the valve.

The PLC works as follows: it waits for one time slot (to get stable sensor signals) and then checks the water level of the tank, distinguishing between three possible states. If the tank reaches a low level (signal l) then the pump is turned on (command $\overline{\mathsf{on}}$) and the *valve* gets closed (command $\overline{\mathsf{close}}$). If the level of the tank is high (signal h) then the PLC listens for requests arriving at channel open_req to open the valve; if the PLC gets a request then it opens the valve (command $\overline{\mathsf{open}}$) and returns; otherwise, it timeouts and then turns the pump off (commands $\overline{\mathsf{off}}$) and closes the valve (command $\overline{\mathsf{close}}$). Finally, if the tank is at some intermediate level between l and h (signal m) the PLC listens for requests of water; if it gets a request of water (via the channel open_req) then it turns the pump on and opens the valve, letting the water flow out; otherwise, if it receives

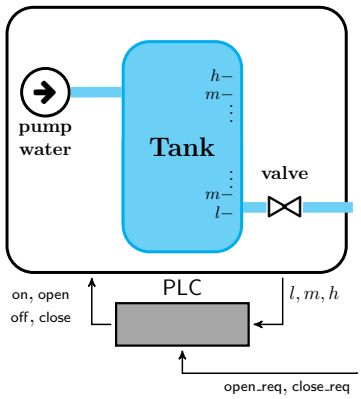

Fig. 1. A simple water-tank system

a request to close the valve (via the channel close_req) then it closes the valve, and then returns.

The specification of the controller PLC mentioned above is the following:

$$\mathsf{rec}\,\mathsf{X}.\big(\mathsf{tick}.\lfloor l.\overline{\mathsf{on}}.\overline{\mathsf{close}}.\mathsf{end}.\mathsf{X} + h.\lfloor \mathsf{open\_req}.\overline{\mathsf{open}}.\mathsf{end}.\mathsf{X}\rfloor(\overline{\mathsf{off}}.\overline{\mathsf{close}}.\mathsf{end}.\mathsf{X})\rfloor(\mathsf{end}.\mathsf{X}) + m.\lfloor \mathsf{open\_req}.\overline{\mathsf{on}}.\overline{\mathsf{open}}.\mathsf{end}.\mathsf{X} + \mathsf{close\_req}.\overline{\mathsf{close}}.\mathsf{end}.\mathsf{X}\rfloor(\mathsf{end}.\mathsf{X})\big).$$

3 Synthesis of monitoring proxies

In Table 5, we provide a synthesis function $[\![-]\!]$ that given a controller $P \in \mathbb{C}\mathtt{trl}$ returns a *syntactically deterministic* edit automaton $\mathsf{E} \in \mathbb{E}\mathtt{dit}$ enforcing the correct behaviour of P, independently of the presence of an arbitrary malware $M \in \mathbb{M}\mathtt{alw}$ that attempts to *inject* and/or *drop* both *actuator commands* and *channel communications*.

In our synthesis, we adopt the following notation for co-actions regarding actuator commands and channel communications: $\overline{\mathsf{Act}} \triangleq \{\overline{a} \mid a \in \mathsf{Act}\}$ and $\overline{\mathsf{Chn}} \triangleq \{\overline{c} \mid c \in \mathsf{Chn}\}$. Furthermore, we set $\mathsf{Act}^* \triangleq \mathsf{Act} \cup \overline{\mathsf{Act}}$ and $\mathsf{Chn}^* \triangleq \mathsf{Chn} \cup \overline{\mathsf{Chn}}$.

Let us comment on the details of the synthesis function $[\![-]\!]$ of Table 5. The edit automaton associated to listening on sensor signals allows all incoming signals expected by the controller, together with the passage of time due to eventual timeouts. All other actions are suppressed. The edit automaton associated to the listening on communication channels is similar, except that communications that are not admitted by the controller are suppressed to prevent both *drops and injections on system channels*, as well as, covert communications between *colluding malware* running in different PLCs. Channel transmissions are allowed only when occurring, in the right order, on those channels intended by the controller; all other actions are suppressed. Only genuine actuator commands (again, in the

Table 5. The synthesis algorithm $[\![-]\!]$

$$
\begin{array}{ll}
[\![\mathsf{rec}\,\mathsf{X}.S]\!] & \triangleq \mathsf{rec}\,\mathsf{X}.[\![S]\!] \\
[\![\lfloor \sum_{i\in I} s_i.S_i \rfloor S]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big(\sum_{i\in I} {}^{s_i}\!/_{s_i}.[\![S_i]\!] + {}^{\mathsf{tick}}\!/_{\mathsf{tick}}.[\![S]\!] + \sum_{\alpha\in \mathsf{Act}^*\cup\mathsf{Chn}^*} {}^{\alpha}\!/_{\tau}.\mathsf{Y}\big) \\
[\![\mathsf{tick}.S]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big({}^{\mathsf{tick}}\!/_{\mathsf{tick}}.[\![S]\!] + \sum_{\alpha\in \mathsf{Act}^*\cup\mathsf{Chn}^*} {}^{\alpha}\!/_{\tau}.\mathsf{Y}\big) \\
[\![\lfloor \sum_{i\in I} c_i.C_i \rfloor C]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big(\sum_{i\in I} {}^{c_i}\!/_{c_i}.[\![C_i]\!] + {}^{\mathsf{tick}}\!/_{\mathsf{tick}}.[\![C]\!] + \sum_{\alpha\in \mathsf{Act}^*} {}^{\alpha}\!/_{\tau}.\mathsf{Y} + \sum_{\gamma\in \mathsf{Chn}^*\setminus\cup_{i\in I}\{c_i\}} {}^{\gamma}\!/_{\tau}.\mathsf{Y}\big) \\
[\![\lfloor \overline{c}.C_1 \rfloor C_2]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big({}^{\overline{c}}\!/_{\overline{c}}.[\![C_1]\!] + {}^{\mathsf{tick}}\!/_{\mathsf{tick}}.[\![C_2]\!] + \sum_{\alpha\in \mathsf{Act}^*} {}^{\alpha}\!/_{\tau}.\mathsf{Y} + \sum_{\gamma\in \mathsf{Chn}^*\setminus\{\overline{c}\}} {}^{\gamma}\!/_{\tau}.\mathsf{Y}\big) \\
[\![\overline{a}.A]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big({}^{\overline{a}}\!/_{\overline{a}}.[\![A]\!] + {}^{\tau}\!/_{\tau}.\mathsf{Y} + \sum_{\alpha\in \mathsf{Act}^*\setminus\{a,\overline{a}\}} {}^{\alpha}\!/_{\tau}.\mathsf{Y} + \sum_{\gamma\in \mathsf{Chn}^*} {}^{\gamma}\!/_{\tau}.\mathsf{Y}\big) \\
[\![\mathsf{end}.\mathsf{X}]\!] & \triangleq \mathsf{rec}\,\mathsf{Y}.\big({}^{\mathsf{end}}\!/_{\mathsf{end}}.\mathsf{X} + \sum_{\alpha\in \mathsf{Act}^*\cup\mathsf{Chn}^*} {}^{\alpha}\!/_{\tau}.\mathsf{Y}\big)
\end{array}
$$

right order) are allowed. *Drops of actuator commands*, the only possible intra-controller interaction occurring between the genuine controller and the malware, are allowed because we want an observation-based monitoring. Finally, the monitoring edit automaton and the associated controller do synchronise at the end of each controller cycle via the action end: all other actions emitted by the compromised controller are suppressed, included those actions coming from the genuine controller that was left behind in its execution due to some injection attack mimicking (part of) some correct behaviour. We recall that only the construct $\mathsf{end}.\mathsf{X}$ (and not $\mathsf{end}.P$) is used to write PLC programs.

Let us start with two easy observations.

Remark 3 (Observation-based monitoring). The edit automata resulting from our synthesis never correct τ-actions (*i.e.*, non-observable actions).

Remark 4 (Colluding malicious activities). Any inter-controller activity which does not comply with the genuine behaviour of the PLC under scrutiny is suppressed by the enforcement.

The synthesis proposed in Table 5 is suitable for implementation.

Proposition 1 (Determinism preservation). *Let $P \in \mathbb{C}\mathtt{trl}$ be a deterministic controller. The automaton $[\![P]\!]$ is syntactically deterministic in the sense of [3].*

Furthermore, our synthesis algorithm is computationally feasible. The complexity of the synthesis is quadratic on the dimension of the controller, where, intuitively, the dimension of a controller $P \in \mathbb{C}\mathtt{trl}$, written $\mathsf{dim}(P)$, is given by the number of prefixes $\alpha \in \overline{\mathsf{Act}} \cup \mathsf{Chn}^* \cup \mathsf{Sens} \cup \{\mathsf{tick}, \mathsf{end}\}$ occurring in it (see [16]).

Proposition 2 (Polynomial complexity). *Let $P \in \mathbb{C}\mathtt{trl}$ be a deterministic controller, the complexity to synthesise $[\![P]\!]$ is $\mathcal{O}(n^2)$, with $n = \mathsf{dim}(P)$.*

As required at the beginning of this section, the synthesised edit automata are always *transparent*, *i.e.*, they never introduce non-genuine behaviours.

Proposition 3 (Transparency). *If $P \in \mathbb{C}\mathtt{trl}$ then $[\![P]\!] \vdash \{P\} \approx \mathsf{go} \vdash \{P\}$.*

Furthermore, our enforcement enjoys *soundness preservation*: in a monitored controller, a malware may never trigger an incorrect behaviour.

Proposition 4 (Soundness). *Let P be an arbitrary controller and M be an arbitrary malware. Then, $[\![P]\!] \vdash \{P \mid M\} \sqsubseteq [\![P]\!] \vdash \{P\}$.*

In the next proposition, we provide a result that is somehow complementary to Proposition 4. The intuition being that in a monitored controller $[\![P]\!] \vdash \{P \mid M\}$ the controller P *may* execute all its (genuine) execution traces even in the presence of an arbitrary malware M. Said in other words, the controller P has a chance to follow (and complete) its correct execution, even when compromised by the presence of a malware M.

Proposition 5. *Let P be an arbitrary controller and M be an arbitrary malware. Then, $[\![P]\!] \vdash \{P \mid M\} \sqsupseteq [\![P]\!] \vdash \{P\}$.*

By applications of Propositions 3, 4, and 5 we can summarise our enforcement in a single equation.

Theorem 1 (Weak enforcement). *Let $P \in \mathbb{C}\mathtt{trl}$ be an arbitrary controller and $M \in \mathbb{M}\mathtt{alw}$ be an arbitrary malware. Then, $[\![P]\!] \vdash \{P \mid M\} \simeq \mathsf{go} \vdash \{P\}$.*

The result of weak enforcement scales to *field communications networks* of communicating controllers compromised by the presence of *colluding malware*.

Proposition 6 (Weak enforcement of field networks). *Let $P_i \in \mathbb{C}\mathtt{trl}$ and $M_i \in \mathbb{M}\mathtt{alw}$, for $1 \leq i \leq n$. Then, $\prod_{i=1}^{n} [\![P_i]\!] \vdash \{P_i \mid M_i\} \simeq \prod_{i=1}^{n} \mathsf{go} \vdash \{P_i\}$.*

However, our enforcement does not enjoy deadlock-freedom.

Remark 5 (Injection attacks may prevent deadlock-freedom). In a monitored controller of the form $[\![P]\!] \vdash \{P \mid M\}$, it may well happen that the malware M misleads the edit automaton $[\![P]\!]$ by injecting an *untimed* trace $M \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} M'$ of actions, with $\alpha_i \neq \mathsf{tick}$, compatible with the correct behaviour of the controller, in the sense that the very same trace may be executed by P: $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$, for some *state* Q. This would give rise to the following admissible execution trace for the monitored controller: $[\![P]\!] \vdash \{P \mid M\} \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} [\![Q]\!] \vdash \{P \mid M'\}$, in which the actual controller P remains inactive. At that point, if the malware M' stops following an admissible runtime behaviour for the controller, the edit automaton $[\![Q]\!]$ will suppress all possible actions, even those proposed by P, which was left behind in its execution. Thus, the monitored controller will continue its evolution as follows: $[\![Q]\!] \vdash \{P \mid M'\} \xrightarrow{\tau} \ldots \xrightarrow{\tau} [\![Q]\!] \vdash \{P' \mid M''\}$. In this case, as neither the controller nor the malware can give rise to zeno behaviours, the enforced system may eventually reach a *deadlock state* in which (i) $P' = \mathsf{end}.\mathsf{X}$, (ii) $M'' = \mathsf{tick}.M'''$, for some M''', or $M'' = \mathsf{nil}$, and (iii) $[\![Q]\!]$ does not allow end-actions because it requires some actions (*e.g.*, actuations) to be performed before ending the controller cycle.

Notice that Remark 5 is not in contradiction with Proposition 5 because in that proposition we proved that a controller has a chance to follow and complete its correct behaviour in the presence of an arbitrary malware. Here, we say a different thing: a malware has a chance to deadlock our monitored controllers.

4 Mitigation: the recipe for deadlock-freedom

In this section, we introduce an extra transition rule for monitored controllers to implement *mitigation*, *i.e.*, the *insertion* of activities in full autonomy, when the controller has lost contact with its enforcer:

$$\text{(Mitigation)}\ \frac{J \xrightarrow{\mathsf{end}} J' \quad \mathsf{E} \xrightarrow{\alpha/\alpha} \mathsf{E}' \quad \alpha \in \mathsf{Chn}^* \cup \overline{\mathsf{Act}} \cup \{\mathsf{tick}\}}{\mathsf{E} \vdash \{J\} \xrightarrow{\alpha} \mathsf{E}' \vdash \{J\}}$$

Intuitively, if the compromised controller signals the end of the scan cycle by emitting the action end and, at the same time, the current edit automaton E is not in the same state, then E will command the execution of a safe trace, without any involvement of the controller, to reach the end of the controller cycle. When both the controller and the edit automaton will be aligned (at the end of the cycle) they will synchronise on the action end, via an application of the transition rule (Enforce), and from then on they will continue in a tethered fashion.

Notice that in a monitored controller $\mathsf{E} \vdash \{J\}$ where J is corrupted by some malware, the two components E and J may get misaligned as they may reach unrelated states. For instance, in case of drop of actuator commands the corrupted controller J may reach an incorrect state, leaving behind its monitoring edit automata E. In this case, the remaining observable actions in the current cycle will be suppressed until the controller reaches the end of the scan cycle, signalled by the emission of an end-action (notice that since our malware are time-guarded they cannot introduce zeno behaviours to prevent a controller to reach the end of its scan cycle). Once the compromised controller has been driven to the end of its cycle, the transition rule (Mitigation) goes into action.

Remark 6. The assumption made in Remark 1 ensures us enough time to complete the mitigation of the scan cycle, well before the maximum cycle limit.

As a main result, we prove that with the introduction of the rule (Mitigation) our runtime enforcement for controllers works faithfully up to weak bisimilarity, ensuring deadlock-freedom.

Theorem 2 (Strong enforcement). *Let $P \in \mathbb{C}\mathtt{trl}$ be an arbitrary controller and $M \in \mathbb{M}\mathtt{alw}$ be an arbitrary malware. Then, $[\![P]\!] \vdash \{P \mid M\} \approx \mathsf{go} \vdash \{P\}$.*

Strong enforcement easily scales to *field networks* of communicating controllers compromised by the presence of (potentially) *colluding malware*.

Corollary 1 (Strong enforcement of field networks). *Let $P_i \in \mathbb{C}\mathtt{trl}$ and $M_i \in \mathbb{M}\mathtt{alw}$, for $1 \leq i \leq n$. Then, $\prod_{i=1}^{n} [\![P_i]\!] \vdash \{P_i \mid M_i\} \approx \prod_{i=1}^{n} \mathsf{go} \vdash \{P_i\}$.*

5 Conclusions and related work

We have defined a formal language to express networks of monitored PLCs, potentially compromised with colluding malware that may forge/drop actuator commands and inter-controller communications. We do not deal with alterations of sensor signals within a PLC, as they can already occur either at the network level or within the sensor devices [9]. The runtime enforcement has been achieved via a finite-state sub-class of Ligatti's edit automata equipped with an ad-hoc operational semantics to deal with *system mitigation*, by inserting actions in full autonomy when the monitored controller is not able to do so in a correct manner. Then, we have provided a synthesis algorithm that, given a deterministic uncorrupted controller, returns, in polynomial time, a syntactically deterministic edit automata to enforce the correctness of the controller. The proposed enforcement meets a number of requirements: observation-based monitoring, transparency, soundness, and deadlock-freedom.

Related work. The notion of *runtime enforcement* was introduced by Schneider [28] to enforce security policies. These properties are enforced by means of *security automata*, a kind of automata that terminates the monitored system in case of violation of the property. Ligatti et al. [20] extended Schneider's work by proposing the notion of *edit automaton*, *i.e.*, an enforcement mechanism able of *replacing*, *suppressing*, or even *inserting* system actions. In general, Ligatti et al.'s edit automata have an enumerable number of states, whereas in the current paper we restrict ourselves to finite-state edit automata. Furthermore, in its original definition the insertion of actions is possible at any moment, whereas our monitoring edit automata can insert actions, via the rule (Mitigation), only when the PLC under scrutiny reaches a specific state, *i.e.*, the end of the scan cycle. Finally, our actions of the form α/β can be easily expressed in the original formulation by inserting the action β and then suppressing the action α. Unlike Schneider and Ligatti et al., we do not enforce specific properties for all admissible systems (in our case, controllers) but we ensure the preservation of the correct semantics of a corrupted controller. Bielova [5] provided a stronger notion of enforceability by introducing a *predictability* criterion to prevent monitors from transforming invalid executions in an arbitrary manner. Falcone et al. [8] proposed a synthesis algorithm, relying on Street automata, to translate most of the property classes defined within the *Safety-Progress hierarchy* [22] into enforcers. Könighofer et al. [12] proposed a synthesis algorithm that given a safety property returns a monitor, called *shield*, that analyses outputs of reactive systems. More recently, Pinisetty et al. [26] have proposed a bi-directional runtime enforcement mechanism for reactive systems, and more generally for cyber-physical systems, to correct both inputs and outputs. Aceto et al. [4] developed an operational framework to enforce safety properties expressed in HML logic with recursion (μHML) by relying on suppression only. Enforceability of modal μ-calculus (a reformulation of μHML) was previously tackled by Martinelli and Matteucci [23]. More recently, Cassar [7] defined a general framework

to compare different enforcement models and different correctness criteria, including optimality.

Finally, in our companion paper [17] we have abstracted over PLC implementations and provided a simple language of regular properties to express correctness properties that should be enforced upon completion of PLC scan cycles.

References

1. Abadi, M., Blanchet, B., Fournet, C.: The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. Journal of the ACM **65**(1), 1:1–1:41 (2018)
2. Abbasi, A., Hashemi, M.: Ghost in the PLC designing an undetectable orogrammable logic controller rootkit via pin control attack. In: Black Hat (2016)
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: On the Complexity of Determinizing Monitors. In: CIAA. LNCS, vol. 10329, pp. 1–13. Springer (2017)
4. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On runtime enforcement via suppressions. In: CONCUR. pp. 34:1–34:17. Schloss Dagstuhl (2018)
5. Bielova, M.: A theory of constructive and predictable runtime enforcement mechanisms. Ph.D. thesis, University of Trento (2011)
6. Cardelli, L., Gordon, A.: Mobile ambients. TCS **240**(1), 177–213 (2000)
7. Cassar, I.: Developing Theoretical Foundations for Runtime Enforcement. Ph.D. thesis, University of Malta and Reykjavik University (2020)
8. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. FMSD **38**(3), 223–262 (2011)
9. Giraldo, J., Urbina, D.I., Cardenas, A., Valente, J., Faisal, M., Ruths, J., Tippenhauer, N.O., Sandberg, H., Candell, R.: A Survey of Physics-Based Attack Detection in Cyber-Physical Systems. ACM Comput. Surv. **51**(4), 76:1–76:36 (2018)
10. Hennessy, M., Merro, M., Rathke, J.: Towards a behavioural theory of access and mobility control in distributed systems. TCS **322**(3), 615–669 (2004)
11. Hennessy, M., Regan, T.: A process algebra for timed systems. Information and Computation **117**(2), 221–239 (1995)
12. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. FMSD **51**(2), 332–361 (2017)
13. Lanese, I., Bedogni, L., Di Felice, M.: Internet of Things: a process calculus approach. In: ACM SAC. pp. 1339–1346. ACM (2013)
14. Lanotte, R., Merro, M.: A Calculus of Cyber-Physical Systems. In: LATA. LNCS, vol. 10168, pp. 115–127. Springer (2017)
15. Lanotte, R., Merro, M.: A semantic theory of the Internet of Things. Information and Computation **259**(1), 72–101 (2018)
16. Lanotte, R., Merro, M., Munteanu, A.: A process calculus approach to correcteness enforcement of PLCs (full version). CoRR abs/2007.09399 (2020)
17. Lanotte, R., Merro, M., Munteanu, A.: Runtime Enforcement for Control System Security. In: CSF. pp. 246–261. IEEE (2020)
18. Lanotte, R., Merro, M., Munteanu, A., Viganò, L.: A Formal Approach to Physics-based Attacks in Cyber-physical Systems. ACM TOPS **23**(1), 3:1–3:41 (2020)
19. Lanotte, R., Merro, M., Tini, S.: A Probabilistic Calculus of Cyber-Physical Systems. Information and Computation **104618**, 1–30 (2020)
20. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Sec. **4**(1-2), 2–16 (2005)

21. Macedonio, D., Merro, M.: A semantic analysis of key management protocols for wireless sensor networks. Science of Computer Programming **81**, 53–78 (2014)
22. Manna, Z., Pnueli, A.: A Hierarchy of Temporal Properties. Tech. rep., Stanford University (1987)
23. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. ENTCS **179**, 31–46 (2007)
24. McLaughlin, S.E.: CPS: stateful policy enforcement for control system device usage. In: ACSAC. pp. 109–118. ACM (2013)
25. Mohan, S., Bak, S., Betti, E., Yun, H., Sha, L., Caccamo, M.: S3A: secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In: HiCoNS. pp. 65–74. ACM (2013)
26. Pinisetty, S., Roop, P.S., Smyth, S., Allen, N., Tripakis, S., Hanxleden, R.: Runtime enforcement of cyber-physical systems. ACM TECS **16**(5s), 178:1–178:25 (2017)
27. Radvanovsky, B.: Project shine: 1,000,000 internet-connected SCADA and ICS stystems and counting (2013), Tofino Security
28. Schneider, F.B.: Enforceable security policies. ACM TISSEC **3**(1), 30–50 (2000)
29. Spenneberg, R., Brüggerman, M., Schwartke, H.: PLC-Blaster: A Worm Living Solely in the PLC. In: Black Hat. pp. 1–16 (2016)