Valued Discrete Timed Automata Supplement

***Abstract*—This document is intended as a supplementary to the paper under review "Smart I/O modules for mitigating cyber-physical attacks on industrial control systems". It is not intended as a standalone file. Should the submission be accepted, this document will be finalised, de-anonymised, and published as a technical report.**

Cyber-Physical Systems (CPSs) are implemented in many industrial and embedded control applications. Where these systems are safety-critical, correct and safe behaviour is of paramount importance. In this work, we propose a novel semantics of runtime enforcement which can secure the safety of real-world physical systems. We term our semantics as "Valued Discrete Timed Automata (VDTA)". Our enforcers prevent the physical damage that a malfunctioning or compromised control system might perform.

I. Introduction

There exists a significant challenge to the safe operation of Cyber-Physical Systems (CPSs) within Industry 4.0 - an assumption of trust in the underlying components making up our systems. For instance, we may assume that any plant operators in control of the system will not command the system to misbehave, that software maintenance mechanisms such as remote updating will only be used for their intended purpose, and that the libraries underpinning our programming environments and Programmable Logic Controller (PLC) operation will always behave as described.

Unfortunately, in world filled with malicious third parties, these are not safe assumptions. Yet, SCADA systems used for traditional distributed industrial control, have trust-based assumptions underpinning their very foundations [1].

As a result, although we add normal digital security measures to protect digital control systems (such as access control, strong passwords, etc.), a system can never be considered fully secure against all possible attack vectors, many of which have been extensively studied in the literature [2], [3]. For instance, misconfigured firewalls, disgruntled employees, or software bugs could allow for unauthorized external access and modification of critical resources [1].

Other forms of defence against these kinds of industrial attacks are presented through the context of *runtime assurance* [4]. Runtime assurance is an emerging technology employed by operating systems, web browsers, spam filters, intrusion-detection systems, firewalls, access-control systems, and so on, to bridge this gap. It works by monitoring untrusted applications and ensuring that they obey a set of specified policies. There are many existing runtime assurance mechanisms for addressing the security of industrial systems in practice, many of which can either be classified as either runtime verifiers or runtime enforcers.

Runtime verifiers are extensively used in commercial and industrial applications for monitoring system correctness. They may also be used to detect the presence of a malicious attack. For instance, the Argus framework [5] is a recent proposal in the literature to place external *intelligent controllers* around an industrial CPS to monitor the behaviour of a plant's physical processes, checking that they behave *process invariants*, which are rules based upon the plant's physics. Argus may also detect certain kinds of rule-breaking communications. Alternatively, PLCs themselves may have mechanisms for runtime verification, as in [6]. In general, these approaches rely on alerting an operator or system supervisor of any faults.

There are also informal approaches which have been adopted in industry, such as integrated AC drive protections to monitor frequency, voltage, resonant frequencies, etc., and abort operation if they are detected as unsuitable. However, these can often be remotely disabled, as was the case with Stuxnet [7].

In contrast, runtime enforcement mechanisms provide *formal* methodologies for runtime assurance mechanisms, mathematically guaranteeing correct operation of a system via *pre-empting* unacceptable inputs and outputs. The enforcement mechanisms can bound unpredictable and *untrustworthy* processes, and ensure that they obey desired policies [8] by either blocking, delaying, modifying, or re-ordering input and output events. A generalised bi-directional enforcer within the context of industrial control systems can be seen in Figure 1. As can be seen, it consists of independent enforcers for both inputs and outputs.

In a sense, the use of runtime enforcement for ensuring security can be thought of as behaviour monitoring misuse and anomaly detection models [9] combined with *automatic recovery mechanisms*. This has a major advantage over pure monitoring approaches, such as those provided by formal Runtime Verification [10] and the aforementioned [5], [6], as the enforcer can pre-emptively correct compromising behaviour and abort dangerous commands before they cause damage, rather than relying on any potentially latent external mechanisms which could themselves also be compromised. This is a key principle of software security: a system should continue to function correctly even in the presence of malicious attack [11].

However, not all runtime enforcement approaches are suitable for *reactive* systems, such as the overcurrent relay in Section II. For instance, mechanisms such as Edit Automata [12], which focus on buffering or deleting events, are only suitable for *transformational* systems, as reactive systems need to continually capture and emit events — time cannot pause while any improper behaviour is analysed or rectified [13].

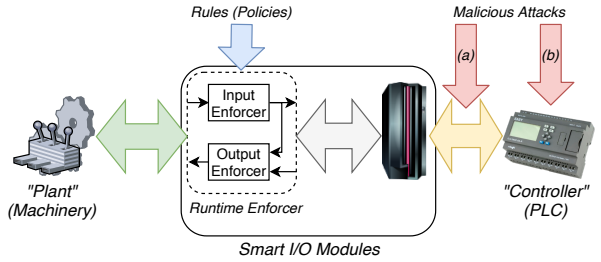

Figure 1: Proposed smart I/O modules with runtime enforcer.

Furthermore, many runtime enforcement mechanisms focus on ensuring certain system constraints without detailing their implementation; as a result, they are often located in the application layer of a given system, meaning that they cannot defend against security issues introduced at lower layers (e.g. via a hardware trojan or networking backdoors). It also means that they have the potential to be maliciously updated and/or disabled, rendering them ineffective. Enforcers have total control over a system's I/O, so a vulnerability in their implementation allowing for a situation where they are compromised could lead to catastrophic consequences.

This is true also of [13], where bi-directional runtime enforcement is presented for reactive CPSs, where safety polices are formally specified using Discrete Timed Automata (DTA). However, they do not consider the security of their approach, and do not detail secure implementations of their semantics, instead relying on application-layer enforcement. Furthermore, their semantics support only Boolean signals, leaving them underequipped for realistic CPS scenarios.

External hardware can be used for enforcers. This minimises the trust required in the target system, as no computational assets are shared. This approach is most similar to that utilised in *shield synthesis* [14], in which external hardware is used ensure safety at the hardware level for reactive systems, but focuses on uni-directional enforcement and untimed properties, meaning that it is unsuitable for many CPS and industrial systems which need to ensure strict timing deadlines.

In this paper, we propose an alternative approach based on *runtime enforcement* to mitigate these security concerns. Runtime enforcement adds an additional layer of execution which monitors the inputs and outputs of a given system, and if they deviate from acceptable values, it pre-emptively corrects them [12]. It provides an assurance mechanism which guarantees the behavior of a given plant and controller to obey a set of user-specified policies [8].

The proposed runtime enforcers would function as an additional and final layer of security sitting between the cyber controllers and their physical sensors and actuators — i.e. within the I/O modules of the PLCs themselves, as depicted in Figure 1. The goal of this defensive system layer would be to ensure the safety of a given compromised CPS by ensuring that no I/O could put the system into a dangerous state that would compromise the physical system. It would act independently of the system's controllers, and if they were compromised it would limit the potential damage that could be caused by an offsite attacker, therefore mitigating an attack's effectiveness.

We term our runtime enforcement framework semantics as Valued Discrete Timed Automata (VDTA), and design them to address the shortcomings of existing enforcement frameworks.

II. Case Study: Overcurrent Protection Relay

Accurate fault detection and isolation is critical for correct and reliable operation of power systems. A failure or delay in fault detection can cause large-scale damage across the whole network, making them an attractive target for malicious attacks. Unfortunately, mechanisms for performing attacks on overcurrent relays have already been demonstrated [15].

In general, faults in an electrical grid can be detected via a number of different means. At the distribution level (i.e. < 35 kV) it is typical to employ non-directional time overcurrent and instantaneous overcurrent protection schemes [15].

Correct operation requires the differentiation between overcurrents casued by faults and those caused by sudden changes in load. Generally, the principle for fault detection is that overcurrents due to load changes are temporary, but overcurrents due to faults occur for long durations. Figure 2 presents a model of an overcurrent fault based on [16]. In this model, current samples are captured every 20 ms. A fault is injected at the 120^{th} sample causing an overcurrent. As the overcurrent is persistent, the system recognises it as a fault, and issues a 'trip' command to a connected circuit-breaker in the next cycle. The circuit-breaker then isolates the system, causing the current to drop to zero.

A. Standard & Operation for Overcurrent Fault Detection

The fault detection principle is implemented by the Inverse Definite Minimum Time (IDMT) curve, covered by the IEC 60255 standard [17]. The curve is defined by Equation 1.

$$t = \frac{K\beta}{\left(\frac{i}{i_{set}}\right)^{\alpha} - 1} \tag{1}$$

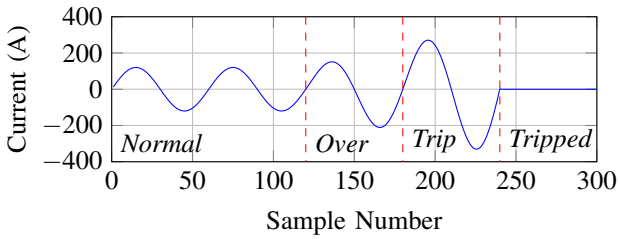

Figure 2: AC overcurrent fault model (inspired from [16]).

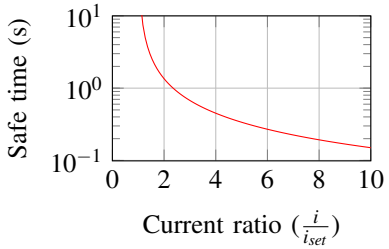

Figure 3: The IDMT curve for this case study (τ).

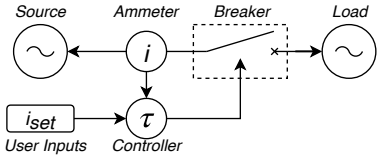

Figure 4: Our overcurrent protection relay.

Here, t is the safe overcurrent operating time, K is a time multiplier, i and i_{set} are the measured and the nominal currents, and α and β are two application-specific parameters used for tuning a system's IDMT slope. For this case study, a "*very inverse*" type curve is used, with $\alpha = 1.0$, $\beta = 13.5$, and time-multiplier $K = 0.1$. We can thus define a function $\tau(i, i_{set})$, which takes the system's current and nominal current and returns the safe time t. The curve for τ is depicted in Figure 3.

B. Typical Implementations

Microprocessor-based numerical relays are common in transmission and distribution networks [15]. A relay suitable for use in a distribution network is modeled using Figure 4, with the function τ representing our IDMT curve. Typical relay controllers sample the current signal at 2-3kHz, and execute their internal logic with clocks around 60MHz [18]. They provide a digital 'trip' command to the circuit-breaker following deterministic fault detection based on time delays as per Equation 1.

C. Security of Smart Grids

When a digital system such as this relay is part of a larger networked system, such as a nationwide smart grid, it is essential that each component behaves safely and securely. Critical infrastructures such as smart grids are usually designed to be reliable in the case of random component failures. However, attacks on key hub components can cause failures outside the scope of reliability analysis [19].

Therefore, it is essential that the circuit-breaker behaves as expected. If the control system is compromised, and the trip signal is suppressed, significant physical damage could occur. Unfortunately, there is already concern that U.S. power grids have been compromised by malicious state actors that might try to damage infrastructures during a crisis or war [20].

III. Valued Discrete Timed Automata (VDTA)

In this work, we define VDTA, which are inspired by the DTA in [13]. Unlike DTA, VDTA support valued signals, internal variables, and complex guard conditions, ensuring compatibility with real-world CPS and industrial systems. Furthermore, we alter the pre-emption *edit* mechanism, as the *random edit* mechanism for pre-empting violating signals in DTA is insufficient for real-world examples.

A. VDTA: Defining Safety Policies for CPS

We consider our industrial CPS systems to have finite ordered sets of valued input channels $I = \{i_1, i_2, \ldots i_n\}$ and valued output channels $O = \{o_1, o_2, \ldots o_n\}$.

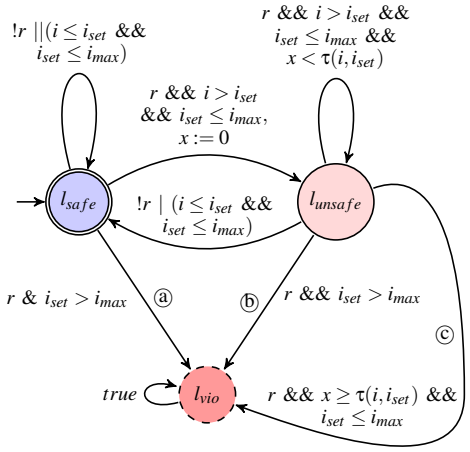

Figure 5: Complete overcurrent policy defined via VDTA $\mathcal{A}_{oc}$

A VDTA can be seen as an automata with a finite set of locations, a finite set of discrete clocks used to represent time evolution, and external input (resp. output channels) called "external variables" which are used for representing system data. They also have internal variables which are used for internal computation, compared to the external variables which model the data carried by the actions from the monitored system (resp. environment). Within VDTA, there is an implicit logical tick similar to synchronous programming languages [21]. All transitions occur relative to this tick. At the start of a tick, all input channels are sampled, and at the end of the tick, all output channels are emitted. Thus, a tick constitutes and atomic reaction of the reactive system. Before we look into the formal definition of VDTA, let us consider an example.

Example III.1. *The overcurrent relay case study can be presented as a VDTA $\mathcal{A}_{oc}$, as depicted in Figure 5. This VDTA formally specifies two properties. Firstly, entering a set current value that is beyond a safe maximum is a violation. Secondly, staying in an overcurrent state for too long without tripping the relay is a violation.*

The VDTA for these rules has a set of locations $L = \{l_{safe}, l_{unsafe}, l_{vio}\}$, with accepting locations $F = \{l_{safe}\}$, and l_{safe} as the initial state. It has the set of external variables $C = \{i, i_{set}, r\}$, where i and i_{set} (representing the measured and set currents) are input channels of integer type, and r (representing the relay command) is an output channel of Boolean type. Its set of internal variables $V = \{i_{max}\}$, where i_{max} (representing the physical max current capacity of the system) is of integer type.

In a VDTA, a transition can have guards on internal variables, external variables and clocks. For example, when in location l_{safe}, the transition from l_{safe} to l_{unsafe} can happen only if r is true, and $i \geq i_{set}$ and $i_{set} \leq i_{max}$. The clock values can be reset upon transitions. For example, upon transition from l_{safe} to l_{unsafe}, the value of clock x is reset to 0. The transitions can also depend on computable functions. For instance, τ is a function with i and i_{set} as input parameters,

which returns a "safe time" value.

It can be seen that there are three violation transitions, labelled ⓐ, ⓑ, and ⓒ. Transitions ⓐ and ⓑ occur when i_{set} is greater than the safe physical value i_{max}, and represent the implementation of the first property. Transition ⓒ encodes the second property, that is the case where the overcurrent time exceeds the safe time without the circuit-breaker 'trip' command being issued.

Let us now consider in more detail the syntax and semantics of VDTAs. For a variable (resp. channel) v, $\mathcal{D}_v$ denotes its domain, and for a tuple of variables $V = (v_1, \ldots, v_n)$, $\mathcal{D}_V$ is the product domain $\mathcal{D}_{v_1} \times \cdots \times \mathcal{D}_{v_n}$. A valuation of the variables in V is a mapping ν which maps every variable $v \in V$ to a value $\nu(v)$ in $\mathcal{D}_v$. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of integer variables representing discrete clocks. A *valuation* for x is an element of $\mathbb{N}$, that is a function from x to $\mathbb{N}$. The set of valuations for the set of clocks X is denoted by χ. For $\chi \in \mathbb{N}^X$, $\chi + 1$ (which captures the ticking of the digital clock) is the valuation assigning $\chi(x)+1$ to each clock variable x of X. Given a set of clock variables $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the valuation of clock variables χ where all the clock variables in X' are assigned to 0.

Definition III.2 (Syntax of VDTAs)**.** A VDTA is a tuple $\mathcal{A} = (L, l_0, X, V, C, \Theta, F, \Delta)$ where:

- L is a finite non-empty set of locations, with $l_0 \in L$ the initial location, and $F \subseteq L$ the set of accepting locations;
- X is a finite set of discrete clocks;
- V is a tuple of typed internal variables;
- C is a tuple of external variables, where $C = I \cup O$, where I is the set of input channels, and O is the set of output channels;
- $\Theta \subseteq \mathcal{D}_V$ is an initial condition which is a computable predicate over V;
- Δ is a finite set of transitions, and each transition $t \in \Delta$ is a tuple (l, G, A, l') also written $l \xrightarrow{G(V,C),V' := A(V,C)} l'$ such that,
  - $l, l' \in L$ are respectively the origin and target locations of the transition;
  - $G = G^D \wedge G^X$ is the guard where
    - $G^D \subseteq \mathcal{D}_V \times \mathcal{D}_C$ is a computable predicate over internal variables and external variables in $V \cup C$;
    - G^X is a clock constraint over X defined as a Boolean combinations of constraints of the form $x \sharp f(V \cup C)$, where $x \in X$ and $f(V \cup C)$ is a computable function, and $\sharp \in \{<, \leq, =, \geq, >\}$;
  - $A = (A^D, A^X)$ is the assignment of the transition where
    - $A^D : \mathcal{D}_V \times \mathcal{D}_C \to \mathcal{D}_V$ defines the evolution of internal variables.
    - $A^X \subseteq X$ is the set of clocks to be reset.

A word is a sequence $\sigma = \eta_1 \cdot \eta_2 \cdots \eta_n$ where $\forall i \in [1, n] : \eta_i$ is a tuple of values of variables in $C = I \cup O$.

Policy VDTA are required to be *deterministic*, i.e. for any given state, the conjunction of any guards of any other outgoing transitions may not be satisfiable; and *complete*, i.e. for any given state at any given time and any valuation of variables in C, at least one transition guard is satisfied.

B. Semantics for VDTA

Let $\mathcal{A} = (\Sigma, L, l_0, X, V, C, \Theta, F, \Delta)$ be a VDTA. The semantics of $\mathcal{A}$ is a timed transition system, where a state consists of a location, and valuations of internal variables V and clocks X. Each transition is associated with values of external variables in C.

Definition III.3 (Semantics of VDTAs)**.** The semantics of $\mathcal{A}$ is a timed transition system $[\![\mathcal{A}]\!] = (Q, q_0, Q_F, \Gamma, \to)$, defined as follows:

- $Q = L \times \mathcal{D}_V \times \mathbb{N}^X$, is the set of states of the form $q = (l, \nu, \chi)$ where $l \in L$ is a location, $\nu \in \mathcal{D}_V$ is a valuation of internal variables, χ is a valuation of clocks;
- $Q_0 = \{(l_0, \nu, \chi_{[X \leftarrow 0]}) \mid \Theta(\nu) = \mathsf{true}\}$ is the set of initial states;
- $Q_F = F \times \mathcal{D}_V \times \mathbb{N}^X$ is the set of accepting states;
- $\Gamma = \{\eta \mid \eta \in \mathcal{D}_C\}$ is the set of transition labels;
- $\to \subseteq Q \times \Gamma \times Q$ the transition relation is the smallest set of transitions of the form $(l, \nu, \chi) \longrightarrow \eta \langle l', \nu', \chi' \rangle$ such that $\exists (l, G, A, l') \in \Delta$, with $G^X(\chi + 1) \wedge G^D(\nu, \eta)$ evaluating to true, $\nu' = A^D(\nu, \eta)$ and $\chi' = (\chi + 1)[A^X \leftarrow 0]$.

A *run* ρ of $[\![\mathcal{A}]\!]$ from a state $q \in Q$ over a *trace* $w = \eta_1 \cdot \eta_2 \cdots \eta_n$ is a sequence of moves in $[\![\mathcal{A}]\!]$: $\rho = q \xrightarrow{\eta_1} q_1 \cdots q_{n-1} \xrightarrow{\eta_n} q_n$, for some $n \in \mathbb{N}$. A run is accepted if it starts from the initial state $q_0 \in Q$ and ends in an accepted state $q_n \in Q_F$.

Example III.4 (Run of a VDTA)**.** *An example run of the VDTA depicted in Figure 5 is elaborated here. Let the internal variable i_{max} be initialized with 10000. A run of this VDTA starting from the initial state $(l_{safe}, i_{max} = 10000, x = 0)$ for the word $\sigma = (4000, 5000, 1) \cdot (8000, 5000, 1) \cdot (7000, 5000, 1) \cdot (8000, 5000, 1) \cdot (8000, 5000, 1) \cdot (8000, 5000, 1)$ is:*

$(l_{safe}, i_{max} = 10000, x = 0) \xrightarrow{(4000,5000,1)} (l_{safe}, i_{max} = 10000, x = 1) \xrightarrow{(8000,5000,1)}$
$(l_{unsafe}, i_{max} = 10000, x = 0) \xrightarrow{(7000,5000,1)} (l_{unsafe}, i_{max} = 10000, x = 1) \xrightarrow{(8000,5000,1)}$
$(l_{unsafe}, i_{max} = 10000, x = 2) \xrightarrow{(8000,5000,1)} (l_{unsafe}, i_{max} = 10000, x = 3) \xrightarrow{(8000,5000,1)}$
$(l_{vio}, i_{max} = 10000, x = 4)$.

The run started in the initial state and ended in a non-accepting state. It is thus a non-accepting run and represents a violation scenario.

In this framework it is possible to express bi-directional enforcement policies, where the enforcer has to first transform inputs from the environment in each step according to property φ defined as VDTA $\mathcal{A}_\varphi$. We thus need to consider the input property that we obtain from $\mathcal{A}_\varphi$ by projecting on inputs.

Remark III.5 (Input VDTA $\mathcal{A}_I$)**.** *Given a property defined as VDTA $\mathcal{A} = (L, l_0, X, V, C, \Theta, F, \Delta)$, input VDTA $\mathcal{A}_I$ is obtained from $\mathcal{A}$ by ignoring outputs channels on the transitions. For every transition $t \in \Delta$, there will be a transition in the input VDTA obtained from t by discarding atomic formulas in G^D and G^X that has output channels.*

a) Edit Functions: Given property φ, defined as VDTA $\mathcal{A}_\varphi$ with semantics $[\![\mathcal{A}_\varphi]\!]$, we introduce $\mathsf{editI}_{\varphi_I}$ (resp. editO_φ), which the enforcer uses for editing input (resp. output) events (whenever necessary), according to input property φ_I (resp. input-output property φ). Note that in each step the enforcer first processes the input from the environment, and transforms it using $\mathsf{editI}_{\varphi_I}$ based on the input property φ_I obtained from the input-output property φ that we want to enforce. Later, the output produced by the program is transformed by the enforcer (when necessary) using editO_φ based on the input-output property φ that we want to enforce.

- $\mathsf{editI}_{\varphi_I}(\sigma_I)$: Given I (set of input channels), $\mathsf{editI}_{\varphi_I}(\sigma_I)$ is the set of all possible valuations η_I (where η_I is a tuple of values of variables in I) such that the word obtained by extending σ_I with η_I can be extended to a sequence that satisfies φ_I (i.e., there exists $\sigma' \in \mathcal{D}_I^*$ such that $\sigma_I \cdot \eta_I \cdot \sigma'$ satisfies φ_I). Formally,

$$\mathsf{editI}_{\varphi_I}(\sigma_I) = \{\eta_I \in \mathcal{D}_I : \exists \sigma' \in \mathcal{D}_I^*, \sigma_I \cdot \eta_I \cdot \sigma' \models \varphi_I\}.$$

- $\mathsf{editO}_\varphi(\sigma, \eta_I)$: Given an input-output word $\sigma \in \mathcal{D}_C^*$ and an input event $\eta_I \in \mathcal{D}_I$, $\mathsf{editO}_\varphi(\sigma, \eta_I)$ is the set of valuations of output channels η_O in O such that the input-output word obtained by extending σ with (η_I, η_O) can be extended to a sequence that satisfies the property φ (i.e., $\exists \sigma' \in \mathcal{D}_C^*$ such that $\sigma \cdot (\eta_I, \eta_O) \cdot \sigma' \models \varphi$). Formally,

$$\mathsf{editO}_\varphi(\sigma, \eta_I) = \{\eta_O \in \mathcal{D}_O : \exists \sigma' \in \mathcal{D}_C^*, \sigma \cdot (\eta_I, \eta_O) \cdot \sigma' \models \varphi\}.$$

b) Selecting edits: For any given violation transition, there may be many possible alternate values for η_I and η_O that would instead result in an accepting transition. To solve this issue, we define two additional functions, $\mathsf{sel\text{-}editI}_{\varphi_I}$ and $\mathsf{sel\text{-}editO}_\varphi$ which the designer can use to *select* a given edit by the designer from the set of possible accepting edits.

- $\mathsf{sel\text{-}editI}_{\varphi_I}(\sigma_I)$: Given $\sigma_I \in \mathcal{D}_I^*$ if $\mathsf{editI}_{\varphi_I}(\sigma_I)$ is non-empty, then $\mathsf{sel\text{-}editI}_{\varphi_I}(\sigma_I)$ returns an element (chosen by the designer) from $\mathsf{editI}_{\varphi_I}(\sigma_I)$, and is undefined if $\mathsf{editI}_{\varphi_I}(\sigma_I)$ is empty.
- $\mathsf{sel\text{-}editO}_\varphi(\sigma, x)$: Given $\sigma \in \mathcal{D}_C^*$, and $x \in \mathcal{D}_I$, if $\mathsf{editO}_\varphi(\sigma, x)$ is non-empty, then $\mathsf{sel\text{-}editO}_\varphi(\sigma, x)$ returns an element (chosen by the designer) from $\mathsf{editO}_\varphi(\sigma, x)$, and is undefined if $\mathsf{editO}_\varphi(\sigma, x)$ is empty.

C. Enforcer constraints

To synthesise an enforcer for a given VDTA, we borrow from the semantics presented in [13].

An enforcer for a given property φ defined as a VDTA $\mathcal{A}$ can be thought as a function $E_\varphi : \mathcal{D}_C^* \to \mathcal{D}_C^*$, where $C = I \cup O$. The enforcer aims to keep a property φ satisfied, and so will examine the updated external variables (input and output channels) each tick, and will transform any that are non-accepting.

It can be trivial to derive enforcers for a properties which do not behave in a useful manner. For instance, an enforcer which satisfies the two properties for the overcurrent relay VDTA could just force the circuit to remain permanently broken. As a result, an overcurrent event can never occur — but the system itself is rendered unusable.

To prevent this (and other situations), several constraints are provided which define enforcer correctness [13].

Definition III.6 (Enforcer for φ)**.** Given property $\varphi \subseteq \mathcal{D}_C^*$, an *enforcer* for φ is a function $E_\varphi : \mathcal{D}_C^* \to \mathcal{D}_C^*$ satisfying the following constraints:

Soundness

$$\forall \sigma \in \mathcal{D}_C^*, \exists \sigma' \in \mathcal{D}_C^* : \mathcal{E}_\varphi(\sigma) \cdot \sigma' \models \varphi. \qquad \textbf{(Snd)}$$

Monotonicity

$$\forall \sigma, \sigma' \in \mathcal{D}_C^* : \sigma \preccurlyeq \sigma' \Rightarrow \mathcal{E}_\varphi(\sigma) \preccurlyeq \mathcal{E}_\varphi(\sigma'). \qquad \textbf{(Mono)}$$

Instantaneity

$$\forall \sigma \in \Sigma^* : |\sigma| = |E_\varphi(\sigma)|. \qquad \textbf{(Inst)}$$

Transparency

$$\begin{aligned} &\forall \sigma \in \mathcal{D}_C^*, \forall \eta_I \in \mathcal{D}_I, \forall \eta_O \in \mathcal{D}_O, \exists \sigma' \in \mathcal{D}_C^* : \\ &\quad E_\varphi(\sigma) \cdot (\eta_I, \eta_O) \cdot \sigma' \models \varphi \\ &\quad \Rightarrow E_\varphi(\sigma \cdot (\eta_I, \eta_O)) = E_\varphi(\sigma) \cdot (\eta_I, \eta_O). \end{aligned} \qquad \textbf{(Tr)}$$

Causality

$$\begin{aligned} &\forall \sigma \in \mathcal{D}_C^*, \forall \eta_I \in \mathcal{D}_I, \forall \eta_O \in \mathcal{D}_O, \\ &\quad \exists \eta_I' \in \mathsf{editI}_{\varphi_I}\left(\left(E_\varphi(\sigma)\right)_I\right), \exists \eta_O' \in \mathsf{editO}_\varphi\left(E_\varphi(\sigma), \eta_I'\right) : \\ &\quad E_\varphi(\sigma \cdot (\eta_I, \eta_O)) = E_\varphi(\sigma) \cdot \left(\eta_I', \eta_O'\right). \end{aligned} \qquad \textbf{(Ca)}$$

- An enforcer must be *sound*, meaning that for any word σ given as input, the output of the enforcer $E_\varphi(\sigma)$ should satisfy the property φ.
- An enforcer must be *transparent*, meaning that the enforcer must edit the actual input and output channel values only when necessary (i.e., only when they lead to violation of the property).
- An enforcer is *online*, so it cannot undo what is released as output (*monotonicity*), and it must not delay, insert, or suppress ticks (*instantaneity*).
- An enforcer must be *causal*, meaning that the enforcer must act as an intermediary such that it first examines values of the input channels to validate them w.r.t property φ. If the actual input values will lead to a violation, the enforcer may change/correct the inputs before forwarding to the program. After the program reacts to these inputs, the enforcer must again validate the outputs from the program. It should forward the outputs from the program (without editing) to the environment if they do not lead to a violation, or edit and forward the edited values.

Definition III.7 (Enforceability)**.** Let $\varphi \subseteq \mathcal{D}_C^*$ be a property. We say that φ is *enforceable* iff an enforcer E_φ for φ exists according to Definition III.6.

D. Enforcing non-accepting I/O events: Selected Edit Actions

To synthesise an enforcer for a given VDTA, we borrow from the semantics presented in [13].

An enforcer for a given property defined as a VDTA $\mathcal{A}$ can be thought as a function $E_{\mathcal{A}} : \mathcal{D}_C^* \to \mathcal{D}_C^*$, where $C = I \cup O$. The enforcer aims to keep a property $\mathcal{A}$ satisfied, and so will examine the updated external variables (input and output channels) each tick, and will transform any that are non-accepting. In other words, enforcers prevent a given system from emitting a non-accepting trace. Whenever a given input/output channel values for a system would result in a guaranteed non-accepting trace (e.g. the next state $q' \in q_v$ is non-accepting), the input/output values must be edited so that the trace remains accepting now or in the future.

Two proposed methods for editing non-accepting I/O values in [13] are: either a *random* edit, where a random but accepting value would be chosen from a list of accepting I/O values; or a *minimum distance* edit, where an accepting value would be chosen with minimum binary distance compared to the current non-accepting value.

These two methods, however, are not always suitable for real-world applications. Consider the VDTA in Figure 5. When transition ⓒ ($l_{unsafe} \to l_{vio}$) occurs on (r && $x \geq \tau(i, i_{set})$ &&$i_{set} \leq i_{max}$), there are three possible channels to edit — i_{set}, i, and r. While editing i or i_{set} could cause the policy to be satisfied (i.e. increasing i_{set} to be larger than i would cause a transition to l_{safe}), such an action would not satisfy *reality* — an over-current event is still occurring! Instead, the edit that causes the relay to be opened, thus breaking the circuit and ensuring safe operation of the system, should be selected.

In general, then, the designer of any given policy should also *select* their preferred *edit actions* out of the list of possible safe edits for each violation transition, thus ensuring practical runtime enforcement.

Example III.8 (Selected Edit Actions for a VDTA)**.** *We can determine the Selected Edit Actions for the VDTA $\mathcal{A}_{oc}$ in Figure 5 based on the original properties we formalised. Transitions ⓐ and ⓑ were intended to stop the nominal current i_{set} from being set larger than i_{max}, the largest current the circuit can safely handle. For these transitions, then, we select the edit action "$i_{set} := i_{max}$", thus constraining the maximum value for i_{set} to be i_{max}. For transition ⓒ, which defines the maximum safe time we can remain in an overcurrent state before violating, we select the edit action "$r := 0$". Now, to prevent the violation, the overcurrent relay will open the breaker, thus ensuring the safety of the system.*

Source Access

The source codes for the runtime enforcement compiler, as well as all case studies and policies, are available online under the MIT license at [REDACTED FOR REVIEW].

References

[1] G. Loukas, *Cyber-physical attacks: a growing invisible threat*. Butterworth-Heinemann, 2015.

[2] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Conference on Security Symposium*, 2014, pp. 95–110.

[3] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 97–106.

[4] M. Clark, X. Koutsoukos, J. Porter, R. Kumar, G. Pappas, O. Sokolsky, I. Lee, and L. Pike, "A study on run time assurance for complex cyber physical systems," Air Force Research Laboratory, Vanderbilt University, Iowa State University, University of Pennsylvania, Galois Inc., Tech. Rep., 2013.

[5] S. Adepu, S. Shrivastava, and A. Mathur, "Argus: An orthogonal defense framework to protect public infrastructure against cyber-physical attacks," *IEEE Internet Computing*, vol. 20, no. 5, pp. 38–45, Sep. 2016.

[6] S. Adepu and A. Mathur, "From design to invariants: Detecting attacks on cyber physical systems," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017, pp. 533–540.

[7] N. Falliere, L. O. Murchu, and E. Chien, "W32. Stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.

[8] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results." in *ESORICS*. Springer, 2010, pp. 87–100.

[9] R. Sekar, T. F. Bowen, and M. E. Segal, "On preventing intrusions by process behavior monitoring." in *Workshop on Intrusion Detection and Network Monitoring*, vol. 1999, 1999, pp. 29–40.

[10] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832608000775

[11] G. McGraw, "Software security," *IEEE Security Privacy*, vol. 2, no. 2, pp. 80–83, March 2004.

[12] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1, pp. 2–16, Feb 2005. [Online]. Available: https://doi.org/10.1007/s10207-004-0046-8

[13] S. Pinisetty, P. S. Roop, S. Smyth, N. Allen, S. Tripakis, and R. V. Hanxleden, "Runtime enforcement of cyber-physical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 178:1–178:25, Sep. 2017.

[14] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: Runtime enforcement for reactive systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer, 2015, pp. 533–548.

[15] A. Chattopadhyay, A. Ukil, D. Jap, and S. Bhasin, "Toward threat of implementation attacks on substation security: Case study on fault detection and isolation," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 6, pp. 2442–2451, June 2018.

[16] A. Ukil, B. Deck, and V. H. Shah, "Current-only directional overcurrent protection for distribution automation: Challenges and solutions," *IEEE Transactions on Smart Grid*, vol. 3, no. 4, pp. 1687–1694, Dec 2012.

[17] *Standard for Measuring Relays and Protection Equipment*, Int. Electrotechnical Commission (IEC) Std. 60 255, 2008.

[18] A. Ukil, V. H. Shah, and B. Deck, "Fast computation of arctangent functions for embedded applications: A comparative analysis," in *2011 IEEE International Symposium on Industrial Electronics*, June 2011, pp. 1206–1211.

[19] S. Sierla, M. Hurkala, K. Charitoudi, C. W. Yang, and V. Vyatkin, "Security risk analysis for smart grid automation," in *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, June 2014, pp. 1737–1744.

[20] S. Gorman, "Electricity grid in U.S. penetrated by spies," *The Wall Street Journal*, April 2009. [Online]. Available: https://www.wsj.com/articles/SB123914805204099085

[21] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan 2003.