IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 31 January 2023, accepted 13 February 2023, date of publication 16 February 2023, date of current version 27 February 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3246121

RESEARCH ARTICLE

Incremental Security Enforcement for Cyber-Physical Systems

ABHINANDAN PANDA[1], ALEX BAIRD[2], SRINIVAS PINISETTY[1], (Member, IEEE), AND PARTHA ROOP[2], (Member, IEEE)

[1]School of Electrical Sciences, Indian Institute of Technology (IIT) Bhubaneswar, Bhubaneswar 752050, India
[2]Department of Electrical and Computer Engineering, The University of Auckland, Auckland 1010, New Zealand

Corresponding authors: Srinivas Pinisetty (spinisetty@iitbbs.ac.in) and Abhinandan Panda (ap53@iitbbs.ac.in)

This work was supported in part by the Ministry of Human Resource Development, Government of India, under Grant SPARC P#701; and in part by the Indian Institute of Technology Bhubaneswar Seed under Grant SP093.

ABSTRACT Cyber-Physical attacks (CP-attacks) are launched either from the cyber-space or from the physical-space to take control of a Cyber-Physical System (CPS). Unlike conventional cyber-attacks, which are prevented through new security patches as new attacks emerge, there are no known mechanisms for incrementally patching CPS in the event of new attacks. To this end, we develop a novel approach based on recent advances in mitigating CP-attacks using run-time enforcement (RE). RE-methods have been developed for CPS, such as industrial processes and pacemakers. However, the proposed solutions are not developed considering the need for future patching as new attacks emerge. To this end, we develop the first compositional RE framework, which is specifically developed to be able to add new security patches as new security policies are added. We illustrate our approach using the case study of a drone swarm. The experimental results show that the proposed compositional/incremental approach does not suffer from the state space explosion, unlike the monolithic composition. We demonstrate a linear relationship between compile time, compile size, and execution time as the number of policies increases in the proposed compositional scheme.

INDEX TERMS Security, runtime enforcement, synchronous programming, cyber-physical systems.

I. INTRODUCTION

Cyber-physical systems (CPSs) incorporate distributed embedded controllers that control a certain physical process [1]. Such systems have been an integral part of the modern environment, associated with various applications, including the internet, smart grids, sensor networks, and intelligent transportation systems. The security of cyber-physical systems has been the primary concern in recent times. With time, these systems are getting more complex and interconnected, making them more vulnerable to security attacks [2], [3].

In cyber-physical security attacks, a remote attacker can take control of the system and intervene with the system's physical processes, resulting in severe damage and even loss of life. Recently, a massive cyber-attack was deployed on one of the largest steel manufacturing organizations in Iran, resulting complete shutdown of the production systems [4]. Several such CPS attacks have been reported in the literature, such as the Stuxnet worm damaging Iranian centrifuges [5], the Maroochy Shire Water Services attack [6], and the German Steel Mill attack [7]. The cyber-attacks on drones that are primarily used in transport have been reported by Yaacoub et al. in [8].

In this regard, formal runtime enforcement techniques [9], [10], [11], [12] have been proposed as a reliable mechanism that works to mitigate these security concerns in cyber-physical systems [11], [13], [14]. The area of research widely recognized as runtime verification (RV) [15] is focused on ways for dynamically verifying a set of desirable policies during the execution of a ''black-box'' system. A formal RV monitor does not influence the systems'

The associate editor coordinating the review of this manuscript and approving it for publication was Peng-Yong Kong.

execution; instead, it observes and verifies if the execution trace of the system satisfies/violates a set of desired policies. Runtime enforcement (RE) is an alternative to passive runtime analysis. An enforcer is created in RE mechanisms to monitor the executions of a black-box system to guarantee that a set of desired policies are maintained. In the event of a violation, the enforcer takes evasive measures to prevent the violation. Blocking the execution [9], altering the input sequence by suppressing and/or adding actions [11], and buffering input actions until a later time when they could be forwarded [10], [12] are examples of evasive actions.

However, in practice, the runtime security policies intended for monitoring tend to grow over time in response to cyber-attacks. Also, as software evolves, understanding the system's secured and unsecured behavior evolves. Cyber-physical systems in new application domains such as e-commerce and medical-database frequently necessitate new security policies along with existing ones. New security policies can be added incrementally. Ensuring security is always an incremental problem, i.e., as new threats emerge, new patches are applied in cyber-security.

Consider a drone swarm as a motivating example of CPS as the drones use distributed controllers to control the physics of their movement, i.e., the position, velocity, and acceleration [16]. The use of drones is slowly increasing in multiple domains due to drones' ability to live-stream, collect real-time video and images, and fly and transport goods. Each drone is permitted a specific air space within which it can operate. Drones have restrictions in each dimension of air-space borders to prevent collisions and violations and a maximum speed for efficient battery use. With the wide use of drones, security threats have emerged, and drones are prone to malicious attacks [17], [18]. It is imperative to add more complex security policies to existing ones for drones for enhanced security.

As a result, as new policies evolve, it is interesting to investigate how the security enforcers should be composed incrementally to enforce all the policies efficiently. Incremental security enforcement allowing the composition of the enforcers with the addition of new security policies is currently an active area of research [19], [20]. While combining the policies and synthesizing a monolithic enforcer for the combined policy may seem doable, it has a lot of downsides, as discussed in [21]. For example, the software goes through multiple stages of development and may eventually need to provide extra functionality, necessitating the fulfillment of more policies. If we proceed with the process of creating a monolithic enforcer for enforcing from scratch each time a new policy is introduced, making tweaks, as well as re-validating and re-certifying the entire system, becomes an extremely expensive task. Again, there may be policies that are concealed for security reasons and are enforced in a secret manner earlier, but changing the enforcing mechanism may have an impact on the related security. Thus, it is challenging to use the monolithic method to enforce a new policy when information about previously enforced policies is unavailable.

To avoid all these problems, for the security of CPS, enforcing new additional policies incrementally (when new issues/attacks are detected) without affecting and without the knowledge of the previously enforced policies is essential. In this work, we focus on the study of incremental enforcement considering the enforcement mechanism proposed in [22] (suitable for reactive cyber-physical systems).

In this work, we model CPS using the synchronous approach. A synchronous reactive system is non-terminating and interacts with the surrounding environment continually. As a result, the system's execution can be thought of as a series of steps, with each step requiring the system to read inputs from the environment, call a reaction function, and compute outputs for emission.

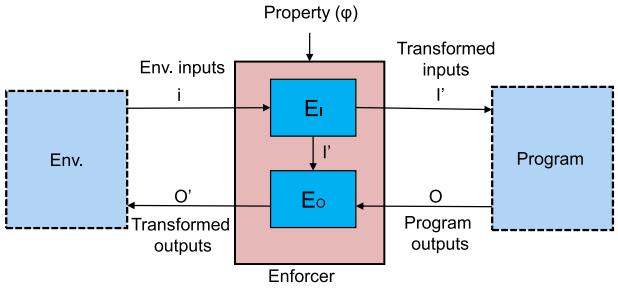


FIGURE 1. Bi-directional enforcement for synchronous programs.

A. OVERVIEW OF THE PROPOSED APPROACH

In this work, we consider the bi-directional RE framework for synchronous reactive systems presented in [22].

The general context of the framework proposed in [22] is illustrated in Figure 1, where i is the input from the environment to the enforcer, i' is the *transformed input* from the enforcer to the program (input enforcement), o is the output of the program to the enforcer, and o' is the *transformed output* from the enforcer to the environment (output enforcement). In this enforcement mechanism, the enforcer reacts instantaneously when a policy violation is observed. Moreover, the framework considers bi-directional enforcement where the enforcer considers the status of the environment and the program in order to enforce the policies. The enforcer respects the *causality* aspects, i.e., every reactive cycle must start with the environment, where the status of the environment inputs must determine the reaction. After the program has reacted, the generated outputs are emitted to the environment. Considering this, the enforcer acts as an intermediary such that it first intercepts the inputs from the environment to validate them relative to the policy and forward the inputs to the program once the policy is satisfied. In the event of any violation, the enforcer suitably alters the inputs before forwarding them to the program.

Following the program's response to these inputs, the enforcer makes sure that either the policy has been met and

the outputs are sent to the environment unchanged, or if it will lead to a violation, the enforcer will handle it by altering the outputs to prevent policy violation.

Problem description: Let us consider a set of initial security policies denoted as $\phi_K = \{\varphi_1, \cdots, \varphi_k\}$ and consider that we have an enforcer E_K for ϕ_K. Assume that some policies in the set ϕ_K are hidden/encrypted (i.e., $\exists j \leq k : \phi_J \subseteq \phi_K$ and $\forall \varphi_i \in \phi_J$, φ_i is hidden/encrypted). Now, suppose there is another new security policy φ_{k+1} to be enforced due to a new threat. How to patch (add a new enforcement layer incrementally) to the existing enforcer E_K such that the new enforcer monitors φ_{k+1} in addition to all the policies in ϕ_K is the problem that we tackle in this work.

In other words, assume, we have an enforcement framework E_K that can monitor a set of policies $\phi_K = \{\varphi_1, \cdots, \varphi_k\}$. When a new policy φ_{k+1} needs to be enforced, we explore the following: Synthesize a new enforcement monitor $E_{\varphi_{k+1}}$ for the policy φ_{k+1} and, how the monitor $E_{\varphi_{k+1}}$ can be incrementally added/composed with the existing enforcement framework E_K.

In this paper, we address the above problem of *incremental security enforcement* as the problem of the *compositionality of security enforcers*. We investigate the composition of two or more enforcers in series (serial composition) in the bi-directional enforcement framework for enforcing policies suitable for reactive cyber-physical systems [22]. Serial composition means that the output of one enforcer is fed as input to the next enforcer in a chain, as shown in Figure 2.

Given two policies φ_1 and φ_2 (where φ_{1I} and φ_{2I} are their corresponding input policies), we can synthesize input and output enforcer for each of these policies $E_I\varphi_1$, $E_I\varphi_2$, $E_O\varphi_1$, and $E_O\varphi_2$, and then compose all the input enforcers and all the output enforcers. Then the composed input enforcer can be combined with the composed output enforcer (see Figure 2).

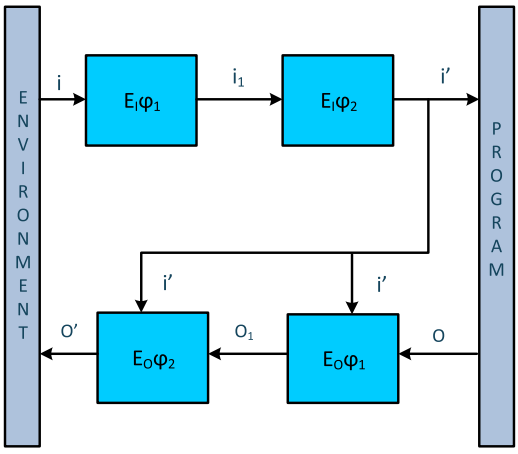


FIGURE 2. Incremental enforcement via serial composition.

For example, the drones operate in physical space (X and Y position) with a single measure of motor rotations per minute (RPM) and controlled by a centrallised swarm controller. Each drone is attributed with a limit in X and Y position and RPM measure such as the set of inputs $I = \{$ min_x_limit, max_x_limit, min_y_limit, max_y_limit, rpm_limit $\}$. The swarm controller measures the set of outputs $O = \{$x_up, x_down, y_up, y_down, rpm_up, rpm_down $\}$, for acceleration and position of each drone. So, to prevent collisions and violation of airspace boundaries, the drones have limits in each dimension, and also maximum RPM is defined to maximise flight time by using the battery efficiently. Each drone is assigned a number, N, and assigned inputs and outputs are denoted max_y$_N$_limit. For example, Drone 1 has max_y_limit denoted max_y$_1$_limit.

Considering collisions and security attacks (detailed discussion in Section VII), the following security policies may be considered for a drone system *incrementally*:

- Policy φ_1 that prevents control signals from causing a *Boundary Breach* of X and Y airspace limits as follows:
  - max_x_limit and x_up should not occur simultaneously
  - max_y_limit and y_up should not occur simultaneously
  - min_x_limit and x_down should not occur simultaneously
  - min_y_limit and y_down should not occur simultaneously
- Policy φ_2 preventing *conflicting control signals* attack is as follows:
  - x_up and x_down should not occur simultaneously
  - y_up and y_down should not occur simultaneously
  - rpm_up and rpm_down should not occur simultaneously
- Policy φ_3: The drone should descend to minimum altitude when control packets are not received for 5 seconds (prevents *Block Control Signals* attack).
- Policy φ_4: rpm_up and rpm_limit should not occur simultaneously (mitigating *Drain Batteries* attack).
- Policy φ_5: Drones with shared boundaries should not simultaneously hover at these boundaries. That means max_x$_1$_limit and min_x$_2$_limit should not occur simultaneously without x$_1$_down or x$_2$_up (mitigating *Shared Boundary* attack for drones 1 and 2).

Assume that the policies φ_1, φ_2, φ_3 and φ_4 exist in the beginning, and the policy φ_5 is to be enforced later with the previous policies. So, it is worth investigating how these policies can be composed such as monolithic composition $(\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_5)$ or *incremental composition via serial composition* $(\varphi_1 \Rrightarrow \varphi_2 \Rrightarrow \varphi_3 \Rrightarrow \varphi_4 \Rrightarrow \varphi_5)$ so that the resulting policy would be enforceable.

In this work, we present the following results:

- Given two policies φ_1 and φ_2 that are individually enforceable, the composed policy $\varphi_1 \wedge \varphi_2$ may not be always enforceable. Only if the composed policy $\varphi_1 \wedge \varphi_2$ is enforceable, a monolithic enforcer can be synthesized for the policy $\varphi_1 \wedge \varphi_2$.
- We show that the bi-directional enforcement framework (in [22]) does not straightaway support the compositionality of enforcers. When using the framework proposed

in [22], we show that bi-directional runtime enforcement is not serial compositional for safety policies. That is, given two policies φ_1 and φ_2 where $\varphi_1 \wedge \varphi_2$ is enforceable, when we synthesize individual enforcers for φ_1 and φ_2, and compose the enforcers in series, the behavior/output of the compositional enforcer does not always satisfy all the constraints of the enforcement mechanism w.r.t $\varphi_1 \wedge \varphi_2$.
- We propose a *revised framework* for bi-directional RE for safety policies, and we show that the proposed revised framework supports extending the policy enforcer in a compositional way using the serial approach. That is, for any two given policies φ_1 and φ_2 if $\varphi_1 \wedge \varphi_2$ is enforceable, the enforcer obtained using the proposed serial compositional framework also acts as an enforcer w.r.t $\varphi_1 \wedge \varphi_2$ (its behavior will be equivalent to the monolithic enforcer for $\varphi_1 \wedge \varphi_2$).
- We developed a compiler which produces serially composed enforcers via our revised framework. This is an extension to *easy-rte* [23] which we call *easy-rte-composition*.
- We considered the case of a swarm of drones and examined a variety of policies that need to be monitored and enforced in order to prevent malicious agents from launching attacks. We implemented the drone swarm policies in C for a software simulation of two drone swarms to evaluate and compare the monolithic and proposed serial composition approach. The controller, enforcers, and drones were all part of this.
- Our evaluation/analysis results using a set of policies in the context of drone swarms clearly demonstrate that the serially composed enforcers do not suffer from the state space explosion that the monolithic approach suffers from. As the number of policies grows for the proposed serial composition approach, our results also clearly demonstrate a linear relationship between compile time, compile size, and execution time.

Outline. In Section II, we discuss different related works. In Section III, we introduce the preliminaries and notations and the RE problem for synchronous programs. We review the runtime enforcement framework for synchronous programs from earlier work in Section IV. The enforcement approaches, monolithic and serial composition, are presented in Section V. Section VI defines Select functions and presents an approach for compositional enforcers, showing that the serial composition approach works using the proposed approach. In Section VII, we present a case study of the enforcement of policies on drone swarms. Section VIII discusses how these policies are implemented and evaluated using both monolithic and serial compositions. Finally, conclusions are drawn in Section IX.

II. RELATED WORK

Runtime verifiers are widely used in commercial and industrial applications to ensure the accuracy of systems. They are also capable of detecting the presence of a malicious attack. For example, in [24], the Argus framework has been proposed to surround an industrial CPS with external intelligent controllers to monitor the behavior of a plant's physical processes, ensuring that they follow process invariants, which are policies based on the plant's physics. Certain types of rule-breaking communications may also be detected by Argus. Alternatively, as in [25], PLCs may include procedures for runtime verification. These approaches, in general, rely on informing an operator or system supervisor of any issues.

There are informal measures used in industry, such as inbuilt ac drive protections that monitor frequency, voltage, and resonant frequencies and terminate operation if they are found to be unsuitable.

de Sá et al. [26] examined a covert attack for service degradation and created a backtracking search optimization algorithm to deal with the system identification attack in cyber-physical control systems. Beg et al. [27] focused on the false-data injection attack and created a detection method to identify changes in cyber-physical dc microgrids based on a set of potential invariants deduced from Simulink/Stateflow diagrams. Sun et al. [28] developed a resilient control method using a dual-mode algorithm to deal with a common sort of denial-of-service (DoS) attack in CPS.

Several AI-based approaches for cyber-physical system security have been discussed, including attack identification, fault detection, and tolerant control [29]. Kim et al. [30] investigated cyber-physical vulnerabilities and proposed a software-defined networking-based framework for man-in-the-middle attacks. They used it in a specific communication-based train control system to improve attack detection resiliency. To monitor and identify cyber and physical threats in IoT environments, Li et al. [31] developed a dual deep learning (DL) model with an energy auditing method. They devised a disaggregation-aggregation framework to learn system behaviors in order to detect attacks.

Synthesizing enforcers from properties is a topic of interest in research. Several RE models have been presented based on how an enforcer is authorized to amend the input sequence. Schneider [9] proposes security automata that focus on the enforcement of safety properties, with the enforcer blocking execution when it detects a sequence of events that does not satisfy the required property. Edit automata [11] allows the enforcer to rectify the input sequence by suppressing and (or) inserting events. The RE methods proposed in [10] and [12] allow events to be buffered and released when a sequence that matches the required property is observed. Edit-automata [11] was enhanced by Mandatory Result Automata (MRAs) [32], which took into account bi-directional runtime enforcement. Unlike other RE frameworks such as [9], [10], [11], and [12], MRA focuses on communication between two parties. Pearce et al. [23] proposed an approach based on runtime enforcement to enforce various types of cyber-physical threats in an industrial CPS application using timed policies (bi-directional framework). Baird et al. [33] modeled jamming, injection, and alteration

attacks to create runtime attack enforcers in a simulated single drone delivery system.

In [19] and [20] investigated the compositionality of enforcers for a unidirectional RE architecture that allows the enforcer to buffer (delay) events. The problem investigated in [19] and [20] is whether it is possible to synthesize several enforcers, one for each property, and whether composing enforcers (in series or in parallel) can enforce all of the properties given a set of properties across the same alphabet.

When considering reactive systems, the enforcer must react instantly. Thus, none of the foregoing approaches, however, are ideal for reactive systems since they halt the program and delay actions.

For reactive systems, the enforcement frameworks developed in works such as [34] and [22] are applicable. In [34], authors introduce a framework to synthesize enforcers for reactive systems, called *shields*, from a set of safety properties. According to [34], the shield is unidirectional, observing inputs from the environment and system (program) outputs, and transforming erroneous outputs.

Several tools have been proposed in specific domains for the composition of security policies. In [35], authors have proposed a tool, polymer, which allows for enforcing the composable policies in java applications. A GUI-based tool PoliSeer proposed in [36] allows for specifying complex security policies. The expandable grid in [37] is an interaction mechanism for developing, updating, and visualizing security policies w.r.t access-control that are a subset of runtime-enforceable policies. The Fang [38] and Firmato tool [39] allows the composition of security policies for firewall management.

As mentioned in the introduction, our work in this study is based on the framework proposed in [22], which addresses bi-directional enforcement. None of the RE frameworks for reactive systems took into account enforcer compositionality, which is the core emphasis and contribution of our research. In this work, we develop an approach where enforcers can be composed in series, addressing the incremental runtime enforcement problem. In the context of security of CPS, the framework allows to incrementally add new enforcement/security layers when needed (e.g., when any new security related concern/issue raises).

III. PRELIMINARIES AND NOTATION

In this section, we introduce the notations and the safety automaton formalism used to define policies to be monitored and enforced. We also briefly recall the RE problem for synchronous programs (all the constraints that an enforcer should fulfill).

A finite word over a finite alphabet Σ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ of members of Σ, and Σ^* denotes the set of finite words over Σ. Considering a finite word σ, its length is denoted as $|\sigma|$. ϵ_Σ is used to denote the empty word over Σ is denoted by ϵ_Σ, or ϵ (when the context makes it evident). Given two words σ and σ', their *concatenation* is indicated as $\sigma \cdot \sigma'$. A word σ' is a *prefix* of a word σ, represented as $\sigma' \preccurlyeq \sigma$, whenever a word σ'' is present such that $\sigma = \sigma' \cdot \sigma''$; σ is called an *extension* of σ'.

A reactive system with a finite ordered sets of Boolean inputs $I = \{i_1, i_2, \cdots, i_n\}$ and Boolean outputs $O = \{o_1, o_2, \cdots, o_m\}$ is considered. $\Sigma_I = 2^I$ denotes the input alphabet, $\Sigma_O = 2^O$ denotes the output alphabet, and the input-output alphabet is $\Sigma = \Sigma_I \times \Sigma_O$. A bit-vector/complete monomial will be used to represent each input (resp. output) event. For example, let us consider $I = \{P, Q\}$. Then, the input $\{P\} \in \Sigma_I$ is denoted as 10, while $\{Q\} \in \Sigma_I$ is denoted as 01 and $\{P, Q\} \in \Sigma_I$ is denoted as 11. A reaction (or input-output event) has the following structure: (x_i, y_i), where $x_i \in \Sigma_I$ and $y_i \in \Sigma_O$.

Given $\sigma = (x_1, y_1) \cdot (x_2, y_2) \cdots (x_n, y_n) \in \Sigma^*$ which is an input-output word, the input word acquired from σ is $\sigma_I = x_1 \cdot x_2 \cdots x_n \in \Sigma_I$, which is a projection that ignores outputs and is based on inputs. Similarly, the output word obtained from σ is $\sigma_O = y_1 \cdot y_2 \cdots y_n \in \Sigma_O$ is the projection on outputs ignoring inputs.

A policy denoted as φ (over Σ) represents a set $\mathcal{L}(\varphi) \subseteq \Sigma^*$. Given a word $\sigma \in \Sigma^*$, $\sigma \models \varphi$ iff $\sigma \in \mathcal{L}(\varphi)$. A policy φ is *prefix-closed* if all prefixes of all words from $\mathcal{L}(\varphi)$ are also in $\mathcal{L}(\varphi)$: $\mathcal{L}(\varphi) = \{w \mid \exists w' \in \mathcal{L}(\varphi) : w \preccurlyeq w'\}$. Prefix-closed policies are the focus of this study. Policies are formalized as safety automata, which we define next in this section.

Synchronous programming languages [40] are ideal for developing synchronous reactive systems. They express safety properties via observers [41], which are statically verified (using model checking). Safety automata are analogous to observers but are enforced at runtime.

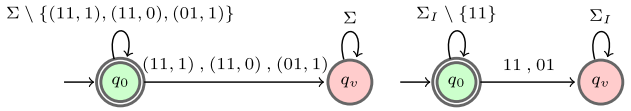
Definition 1 (Safety Automaton): A safety automaton *(SA)* $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ *is a tuple, where* Q *denotes the set of states, known as* locations, $q_0 \in Q$ *is a distinct starting location,* $q_v \in Q$ *is a distinct non-accepting (violating) location, the alphabet is* $\Sigma = \Sigma_I \times \Sigma_O$, *and the transition relation is* $\rightarrow \subseteq Q \times \Sigma \times Q$. *Except for* q_v, *all the other locations are accepting (i.e., all the locations in* $Q \setminus \{q_v\}$*). Location* q_v *is a distinct violating (trap) location, thus no transitions in* $\rightarrow$ *from* q_v *to a location in* $Q \setminus \{q_v\}$. *Whenever there exists* $(q, a, q') \in \rightarrow$, *we denote it as* $q \xrightarrow{a} q'$. *Relation* $\rightarrow$ *is extended to words* $\sigma \in \Sigma^*$ *by noting* $q \xrightarrow{\sigma.a} q'$ *whenever there exists* q'' *such that* $q \xrightarrow{\sigma} q''$ *and* $q'' \xrightarrow{a} q'$. *A location* $q \in Q$ *is reachable from* q_0 *if there exists a word* $\sigma \in \Sigma^*$ *such that* $q_0 \xrightarrow{\sigma} q$.

An SA $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is *deterministic* if $\forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \implies (q' = q'')$. $\mathcal{A}$ is *complete* if $\forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q'$. A word σ is *accepted* by $\mathcal{A}$ if there exists $q \in Q \setminus \{q_v\}$ such that $q_0 \xrightarrow{\sigma} q$. The set of all words accepted by $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$.

Remark 1: We can first determinize and complete a non-deterministic or incomplete automaton provided by the user. We further assume that Q *has no (redundant) locations that are unreachable from* q_0. *Hence, in the rest of this work,* φ *is a safety policy specified as deterministic and complete SA* $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$.

The enforcer must first alter inputs from the environment in each step according to policy φ specified as SA $\mathcal{A}_\varphi$ according to the causality requirement. As a result, we must examine the input policy obtained by projecting on inputs from $\mathcal{A}_\varphi$.

Definition 2 (Input SA $\mathcal{A}_{\varphi_I}$): Given $\varphi \subseteq \Sigma^*$, specified as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, by discarding outputs on the transitions, input SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ is derived from $\mathcal{A}_\varphi$.That is, for every transition $q \xrightarrow{(x,y)} q' \in \rightarrow$ where $(x, y) \in \Sigma$, there is a transition $q \xrightarrow{x} q' \in \rightarrow_I$, where $x \in \Sigma_I$. $\mathcal{L}(\mathcal{A}_{\varphi_I})$ is represented as $\varphi_I \subseteq \Sigma_I^*$.



(a) SA $\mathcal{A}_P$. (b) Input SA from $\mathcal{A}_P$.

FIGURE 3. SA (left), and its input SA (right).[2]

Example 1 (Example policy defined as SA and its input SA): Consider $I = \{B, Q\}$ and $O = \{X\}$. Let us consider the policy: P: "B and Q can't happen at the same time, and Q and X can't happen at the same time". Policy P is defined by the safety automaton in Figure 3a. The input SA for the SA in Figure 3a defining policy P is shown in Figure 3b. Though the SA $\mathcal{A}_\varphi$ is deterministic, the input SA $\mathcal{A}_{\varphi_I}$ may be non-deterministic. This is the case with the considered example as shown in Figure 3b.

Lemma 1: Consider $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ be the input automaton derived from $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. The policies we have are as follows:

1 $\forall (x, y) \in \Sigma, \forall q, q' \in Q : q \xrightarrow{(x,y)} q' \implies q \xrightarrow{x}_I q'$.

2 $\forall x \in \Sigma_I, \forall q, q' \in Q : q \xrightarrow{x}_I q' \implies \exists y \in \Sigma_O : q \xrightarrow{(x,y)} q'$.

Lemma 1 is an immediate consequence from Definitions 1 and 2. Policy 1 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ in the automaton $\mathcal{A}_\varphi$ upon input-output event $(x, y) \in \Sigma$, then there is a transition from state q to state q' in the input automaton $\mathcal{A}_{\varphi_I}$ upon the input event $x \in \Sigma_I$. Policy 2 states that if there is a transition from state $q \in Q$ to state $q' \in Q$ upon input event $x \in \Sigma_I$, then there must be an output event $y \in \Sigma_O$ s.t. there is a transition from state q to state q' upon event (x, y) in the automaton $\mathcal{A}_\varphi$.

Definition 3 (Product of SA): Given two SA $\mathcal{A}_{\varphi_1} = (Q^1, q_0^1, q_v^1, \Sigma, \rightarrow_1)$, and $\mathcal{A}_{\varphi_2} = (Q^2, q_0^2, q_v^2, \Sigma, \rightarrow_2)$, their product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2} = (Q, q_0, q_v, \Sigma, \rightarrow)$ where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $q_v = (q_v^1, q_v^2)$, and the transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ with $((q^1, q^2), a, (q'^1, q'^2)) \in \rightarrow$ if $(q^1, a, q'^1) \in \rightarrow^1$ and $(q^2, a, q'^2) \in \rightarrow_2$.

In the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$, all the locations in $(Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ are trap locations. All the outgoing transitions from these locations can be replaced with self-loops, and all such locations can be merged into a single violating location labeled as q_v. Any outgoing transition from a location in $Q \setminus (Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ to a location in $(Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ goes to q_v instead.

The product of SAs is useful to enforce multiple policies using the monolithic approach by first constructing a product of the given SAs. Given two deterministic and complete SAs $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$, the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ is deterministic and complete which recognizes the language $\mathcal{L}(\mathcal{A}_{\varphi_2}) \cap \mathcal{L}(\mathcal{A}_{\varphi_2})$.

A. EDIT FUNCTIONS

Let us consider policy $\varphi \subseteq \Sigma^*$, specified as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, and SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$ derived from $\mathcal{A}_\varphi$ by discarding outputs. The enforcer utilizes the following $\mathsf{editI}_{\varphi_|}$ (resp. editO_φ), for editing input (resp. output) events (when required), as per the policy φ_I (resp. φ).

- **$\mathsf{editI}_{\varphi_|}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$, $\mathsf{editI}_{\varphi_|}(\sigma_I)$ is the set of input events $x \in \Sigma_I$ s.t. the word obtained by concatenating x after σ_I satisfies policy φ_I. Formally,

$$\mathsf{editI}_{\varphi_|}(\sigma_I) = \{x \in \Sigma_I : \sigma_I \cdot x \models \varphi_I\}.$$

When we consider the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the members in Σ_I that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:

$$\mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q) = \{x \in \Sigma_I : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$

Let us, for example, consider the SA in Figure 3b derived from the SA in Figure 3a by projecting on inputs. If we consider $\sigma = (10, 0) \cdot (01, 1)$, we have $\sigma_I = 10 \cdot 01$. Then, $\mathsf{editI}_{\varphi_|}(\sigma_I) = \Sigma_I \setminus \{11\}$. Moreover, $q_0 \xrightarrow{10\cdot01}_I q_0$, and $\mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q_0) = \Sigma_I \setminus \{11\}$.

If $\mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q)$ is non-empty, then $\mathsf{nondet} - \mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q)$ returns an element (chosen randomly) from $\mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q)$ and is undefined if $\mathsf{editI}_{\mathcal{A}_{\varphi_|}}(q)$ is empty.

- **$\mathsf{editO}_\varphi(\sigma, x)$**: Consider an input event $x \in \Sigma_I$, and an input-output word $\sigma \in \Sigma^*$. We have $\mathsf{editO}_\varphi(\sigma, x)$, the set of output events y in Σ_O s.t. the input-output word obtained by concatenating σ followed by (x, y) (i.e., $\sigma \cdot (x, y)$) satisfies policy φ. Formally,

$$\mathsf{editO}_\varphi(\sigma, x) = \{y \in \Sigma_O : \sigma \cdot (x, y) \models \varphi\}.$$

When we consider the automaton $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$ specifying policy φ, and an input event $x \in \Sigma_I$, the set of output events y in Σ_O permitting to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ with (x, y) is defined as:

$$\mathsf{editO}_{\mathcal{A}_\varphi}(q, x) = \{y \in \Sigma_O : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v\}.$$

For example, consider policy P defined by the automaton in Figure 3a. We have $\mathsf{editO}_{\mathcal{A}_\varphi}(q_0, 01) = \{0\}$.

If $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is not empty, then $\mathsf{nondet} - \mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ returns a random element from $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$, and if $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty $\mathsf{nondet} - \mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is undefined.

[2]Here, $\Sigma = \{(00, 0), (00, 1), (01, 0), (01, 1), (10, 0), (10, 1), (11, 0), (11, 1)\}$. So $\Sigma \setminus \{(11, 1), (11, 0), (01, 1)\} = \{(00, 0), (00, 1), (01, 0), (10, 0), (10, 1)\}$.

B. RUNTIME ENFORCEMENT FOR SYNCHRONOUS PROGRAMS

In this section, we briefly recall the RE problem for synchronous programs from [22]. In this setting, that we also consider in this work, as illustrated in Figure 1, an enforcer monitors and corrects both inputs and outputs of a synchronous program according to a given safety policy $\varphi \subseteq \Sigma^*$.

The model hypothesizes that the black-box synchronous program can be called using a custom function call called *ptick* that is called just once during each reaction / synchronous step. We can formally consider ptick as a function from Σ_I to Σ_O that accepts a bit vector $x \in \Sigma_I$ and returns a bit vector $y \in \Sigma_O$.

An enforcer for the policy φ can only alter an input-output event when it's absolutely essential; it can't block, postpone, or suppress events. Let's remember the two functions $\mathsf{editI}_{\varphi_|}$ and editO_φ from Section III, which the enforcer for φ uses to edit the current input (or output) event according to the policy φ. An enforcer may be thought of as a function that modifies input-output words at a high level. An enforcement function for the policy φ takes an input-output word over Σ as input and produces an input-output word over Σ that conforms to φ as output.

We reproduce from [22] and briefly discuss, Definition 4 of the constraints that an enforcer for any given policy φ should satisfy:

Definition 4 (Enforcer for φ): An enforcer *for a given policy $\varphi \subseteq \Sigma^*$ is a function $E_\varphi : \Sigma^* \to \Sigma^*$ satisfying the following constraints:*

Soundness

$$\forall \sigma \in \Sigma^* : E_\varphi(\sigma) \models \varphi. \qquad \textbf{(Snd)}$$

Monotonicity

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \Rightarrow E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma'). \qquad \textbf{(Mono)}$$

Instantaneity

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

Transparency

$$\begin{aligned} &\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O : \\ &E_\varphi(\sigma) \cdot (x, y) \models \varphi \\ &\quad \Longrightarrow E_\varphi(\sigma \cdot (x, y)) = E_\varphi(\sigma) \cdot (x, y). \end{aligned} \qquad \textbf{(Tr)}$$

Causality

$$\begin{aligned} &\forall \sigma \in \Sigma^*, \forall x \in \Sigma_I, \forall y \in \Sigma_O, \exists x' \in \mathsf{editI}_{\varphi_|}(E_\varphi(\sigma)_I), \\ &\exists y' \in \mathsf{editO}_\varphi(E_\varphi(\sigma), x') : E_\varphi(\sigma \cdot (x, y)) \\ &= E_\varphi(\sigma) \cdot (x', y'). \end{aligned} \qquad \textbf{(Cau)}$$

The enforcer releases the input-output sequence $E_\varphi(\sigma)$ as output after reading the input-output sequence σ, and $E_\varphi(\sigma)_I \in \Sigma_I^*$ is the projection on the inputs. Note that $\mathsf{editI}_{\varphi_|}(E_\varphi(\sigma)_I)$ returns a set of input events in Σ_I, s.t. $E_\varphi(\sigma)_I$ (which is the input alphabet projection of the input-output word $E_\varphi(\sigma)$) followed by any event from $\mathsf{editI}_{\varphi_|}(E_\varphi(\sigma)_I)$ satisfies φ_I. $\mathsf{editO}_\varphi(E_\varphi(\sigma), x')$ returns a set of output events in Σ_O, such that for any event y in $\mathsf{editO}_\varphi(E_\varphi(\sigma), x')$, $E_\varphi(\sigma) \cdot (x', y)$ satisfies φ.

- Soundness (*Snd*) states that the output of the enforcer $E_\varphi(\sigma)$ must satisfy φ for any word $\sigma \in \Sigma^*$.
- Monotonicity (*Mono*) specifies that the enforcer's output for an extended word σ' of a word σ extends the enforcer's output for σ. The enforcer cannot undo what has already been transmitted as output due to the monotonicity condition.
- Instantainety (*Inst*) states that for any given input-output word σ, the enforcer's output $E_\varphi(\sigma)$ should contain exactly the same number of events as σ (i.e., E_φ is length-preserving). As a result, the enforcer is unable to delay, insert, or suppress events. When the enforcer receives a new event, it must respond immediately and provide an output event instantaneously.
- Transparency (*Tr*) states that for any given word σ and event (x, y), if the enforcer's output for σ (i.e., $E_\varphi(\sigma)$) followed by the event (x, y) fulfills the policy φ (i.e., $E_\varphi(\sigma) \cdot (x, y) \models \varphi$), then the output that the enforcer produces for input $\sigma \cdot (x, y)$ will be $E_\varphi(\sigma) \cdot (x, y)$. This means that when no modification is required to meet the policy φ, the enforcer does nothing.
- Causality (*Cau*) states that the enforcer generates input-output event (x', y') for every input-output event (x, y), where the enforcer first processes the input portion x to produce the transformed input x' according to policy φ using $\mathsf{editI}_{\varphi_|}$ for every input-output event (x, y). After executing function ptick with the transformed input x', the enforcer reads and transforms output $y \in \Sigma_O$, which is the program's output, to generate the transformed output y' using editO_φ.

Remark 2: After reading input-output sequence $\sigma \in \Sigma^*$, let $E_\varphi(\sigma)$ be the input-output sequence produced as output by the enforcer for φ. If what has already been computed as output by the enforcer $E_\varphi(\sigma)$ followed by (x, y) does not allow to satisfy the policy φ, the enforcer edits (x, y) using functions $\mathsf{editI}_{\varphi_|}$ and editO_φ when reading a new event (x, y). When editing the current event (x, y), important to note that there may be several options.

Let us consider the policy P from Example 1. If we consider $\sigma = (10, 1) \cdot (01, 0)$, the output of the enforcer after reading σ should be $E_\varphi(\sigma) = (10, 1) \cdot (01, 0)$. Consider another new event $(11, 0)$, and $E_\varphi(\sigma) \cdot (11, 0)$ does not satisfy φ, and the enforcer thus has to alter this new event $(11, 0)$. We have $E_\varphi(\sigma)_I = 10 \cdot 01$, and since $\mathsf{editI}_{\varphi_|}(10 \cdot 01) = \{00, 01, 10\}$ the enforcer can choose any element from this set as the transformed input.

Definition 5 (Enforceability): Let $\varphi \subseteq \Sigma^*$ be a policy. We say that φ is enforceable *iff an enforcer E_φ for φ exists according to Definition 4.*

Remark 3 (Not all safety policies are enforceable): Not all policies are enforceable, even if we restrict ourselves

to prefix-closed safety policies as is shown and illustrated in [22].

Remark 4 (Condition for enforceability): We recall the enforceability condition proposed and proved in [22]. Consider a policy φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. Policy φ is enforceable iff the following condition holds:

$$\forall q \in Q, q \neq q_v \implies \exists (x,y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v \quad \textbf{(EnfCo)}$$

It's worth noting that given any policy φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$, to test whether $\mathcal{A}_\varphi$ satisfies condition (EnfCo) is straightforward.

IV. RUNTIME ENFORCEMENT FRAMEWORK FOR POLICIES DEFINED AS SA

In this section, we recall the definition of an enforcement function from [22], which incrementally builds the output and presents how any given word $\sigma \in \Sigma^*$ is transformed according to the policy φ.

A pair (x, y) is an input-output event (reaction), where $x \in \Sigma_I$ is the input, and $y \in \Sigma_O$ is the output. The enforcer immediately produces an input-output event (x', y') as output after receiving an input-output event (x, y) as input. The enforcer processes the input x first, producing a transformed input x', and then the output y, producing the transformed event (x', y'). The enforcement function E_φ is made up of two functions: E_I and E_O. E_I reads the input x (from the environment) and produces a transformed input x', while E_O reads the transformed input x' (output of E_I) and the output y (which is the output obtained by invoking ptick with x') and adds the transformed event (x', y') to the output of the enforcer.

Definition 6 (Enforcement function): The enforcement function $E_\varphi : \Sigma^* \rightarrow \Sigma^*$ for a given policy $\varphi \subseteq \Sigma^*$, is defined as $E_O(E_I(\sigma_I), \sigma_O)$:

where:

- *$E_I : \Sigma_I^* \rightarrow \Sigma_I^*$ is defined as*:

$$E_I(\epsilon_{\Sigma_I}) = \epsilon_{\Sigma_I}$$

$$E_I(\sigma_I \cdot x) = \begin{cases} E_I(\sigma_I) \cdot x & \text{if } E_I(\sigma_I) \cdot x \models \varphi_I, \\ E_I(\sigma_I) \cdot x' & \text{otherwise} \end{cases}$$

  where $x' = \mathsf{nondet} - \mathsf{editI}_{\varphi_I}(E_I(\sigma_I))$.

- *$E_O : \Sigma_I^* \times \Sigma_O^* \rightarrow (\Sigma_I \times \Sigma_O)^*$ is defined as*:

$$E_O(\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O}) = \epsilon_\Sigma$$

$$E_O(\sigma_I \cdot x, \sigma_O \cdot y) = \begin{cases} E_O(\sigma_I, \sigma_O) \cdot (x, y) & \text{if} \\ \qquad E_O(\sigma_I, \sigma_O) \cdot (x, y) \models \varphi, \\ E_O(\sigma_I, \sigma_O) \cdot (x, y') & \text{otherwise} \end{cases}$$

  where $y' = \mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$.

The function E_φ accepts a word over Σ^* and outputs another word over Σ^*. We have $\sigma_I \in \Sigma_I^*$ is the projection of σ on inputs, and $\sigma_O \in \Sigma_O^*$ is the projection of σ on outputs, for a word $\sigma \in \Sigma^*$. The result of function E_O is the output of the enforcement function E_φ, which is defined through two functions, E_I and E_O.

TABLE 1. Functional definition example.

σ_I	σ_O	$E_I(\sigma_I)$	$E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$
ϵ_I	ϵ_O	ϵ_I	$(\epsilon_I, \epsilon_O) = \epsilon$
01	0	01	$(01, 0)$
$01 \cdot 01$	$0 \cdot 1$	$01 \cdot 01$	$(01, 0) \cdot (01, \mathbf{0})$

Function E_I: Function E_I accepts the word obtained by projecting on the inputs ($\sigma_I \in \Sigma_I^*$) as input and returns a word in Σ_I^* as output for a given word $\sigma \in \Sigma^*$. Inductively, the function E_I is defined. When the input $\sigma_I = \epsilon_{\Sigma_I}$, it returns$\epsilon_{\Sigma_I}$. When Σ_I is read as input and $E_I(\sigma_I)$ is returned as output, there are two possible possibilities depending on whether $E_I(\sigma_I) \cdot x$ fulfills the policy φ_I or not.

- If $E_I(\sigma_I)$ succeeded by the new input x satisfies the input policy φ_I, then the new input x is concatenated to the previous output of function E_I (that is, $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x$).
- Otherwise, $E_I(\sigma_I) \cdot x$ does not satisfy φ_I. In this case, input x is converted using $\mathsf{nondet} - \mathsf{editI}_{\varphi_I}(E_I(\sigma_I))$ to obtain transformed input x', which is appended to the previous output of function E_I (that is, $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x'$). $\mathsf{nondet} - \mathsf{editI}_{\varphi_I}(E_I(\sigma_I))$ returns $x' \in \Sigma_I$, such that φ_I is satisfied by the preceding output of function E_I followed by x'.

Function E_O: Function E_O takes an input word from Σ_I^* and an output word from Σ_O* as input and returns an input-output word in Σ^*, which is a sequence of tuples with an input and an output for each event. Inductively, the function E_O is defined. The output of E_O is ϵ when both the input and output words are empty. If $\sigma_I \in \Sigma_I^*$ and $\sigma_O \in \Sigma_O^*$ is read, the output will be $E_O(\sigma_I, \sigma_O)$, and if another fresh input event x and output event y are observed, there are two alternatives depending on whether $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ satisfies φ or not.

- If $E_O(\sigma_I, \sigma_O)$ succeeded by (x, y) respects φ, then (x, y) is added to the previous output of function E_O (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y)$).
- If the preceding case is not satisfied, then $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ does not respect/satisfy φ. $\mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$ is thus used to alter output y to obtain y' (altered output), and the event (x, y') is added to the previous output of the function E_O (i.e., $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y')$). $\mathsf{nondet} - \mathsf{editO}_\varphi(E_O(\sigma_I, \sigma_O), x)$ outputs $y' \in \Sigma_O$ such that φ is satisfied by the preceding output of function E_O followed by (x, y').

Remark 5 (Functional definition satisfies constraints): In [22], it is proved that for any given policy φ that is enforceable, the enforcer defined as function E_φ (Definition 6) satisfies the (Snd), (Tr), (Mono), (Inst), and (Cau) constraints (Definition 4).

Example 2 (Functional definition): Let us consider the policy "B and Q can't happen at the same time, and Q and X can't happen at the same time" illustrated in Figure 3, where $I = \{B, Q\}$ and $O = \{X\}$. The output of functions E_I, E_O is illustrated in Table 1 when the input sequence $\sigma = (01, 0) \cdot (01, 1)$ (where $\sigma_I = 01 \cdot 01$ and $\sigma_O = 0 \cdot 1$) is

TABLE 2. Example illustrating behavior of enforcer for $\mathcal{A}_{S_1 \cap S_2}$.

σ_I	σ_O	$E_I(\sigma_I)$	$E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$
ϵ_I	ϵ_O	ϵ_I	$(\epsilon_I, \epsilon_O) = \epsilon$
100	1	100	(100, 1)
100 · 110	1 · 1	100 · 100	(100, 1) · (100, 1)
100 · 110 · 011	1 · 1 · 0	100 · 100 · 001	(100, 1) · (100, 1) · (001, 0)

processed incrementally by the enforcement function. When σ is (01,0), since it satisfies policy P, it is emitted without any alteration. For the second event (01,1) (σ = (01, 0) · (01, 1)), the input enforcer $E_I(\sigma_I) = 01 \cdot 01$ since it does not violate the input policy but since the output of 1 in this step violates the policy P; it is transformed into 0 satisfying the policy. Thus, the enforcer outputs (01, 0) · (01, 0).

V. MONOLITHIC AND INCREMENTAL SCHEMES FOR ENFORCING MULTIPLE POLICIES

In this section, we focus on the problem of how we enforce a given set of policies expressed as SA in the considered reactive systems framework.

Example 3 (Example policies): Let $I = \{A, B, C\}$ and $O = \{R\}$. Consider the following policies: S_1: "A and B cannot happen simultaneously, and also B and R cannot happen simultaneously" and S_2: "B and C cannot happen simultaneously". The safety automaton in Figure 4a and Figure 4b define policies S_1 and S_2 respectively.

A. MONOLITHIC SECURITY ENFORCEMENT

Composing all of the policies first is one way to enforce a collection of policies (taking the product of all the SA). We can synthesize one enforcer for the resulting policy if the resulting SA is enforceable according to Definition 5.

In the monolithic approach, policies (specified as SA) are first combined using intersection (see the Definition 3, the product of SA), and an enforcer for the resulting policy is synthesized. Specifically, given any two safety policies φ_1 and φ_2, to enforce both these policies, we first compute $\varphi = \varphi_1 \cap \varphi_2$ (by computing the product of SA for φ_1 and φ_2). Then if the resulting SA for φ is enforceable as per Definition 5, we synthesize an enforcer for φ using the approach described in Section IV.

Example 4 (Monolithic approach): Consider policies S_1 and S_2 defined as SAs illustrated in Figure 4. The SA obtained by taking the product of both these automata is shown in Figure 5 defining the policy $S_1 \cap S_2$. The policy $S_1 \cap S_2$ is enforceable since for every accepting state, there is at least one outgoing transition to an accepting state (See Remark 4). Table 2 illustrates behavior of enforcer for policy $\mathcal{A}_{S_1 \cap S_2}$ when the input-output word (100, 1)·(110, 1)·(011, 0) *is processed incrementally.*

Theorem 1 (Enforceability using the monolithic approach): Consider two policies φ_1, φ_2 defined as SA, and $\varphi = \varphi_1 \cap \varphi_2$.

If policy φ_1 or policy φ_2 is non-enforceable, then $\varphi_1 \cap \varphi_2$ is non-enforceable.

The proof of Theorem 1 is given in Appendix A.

Remark 6 (Enforceability using the monolithic approach): Though policies φ_1 and φ_2 are enforceable individually, policy $\varphi_1 \cap \varphi_2$ may not be enforceable, as illustrated in the following example.

Example 5 (Monolithic approach does not always work): Consider the two policies shown in Figure 6. Here $I = \{0, 1\}$ and $O = \{0, 1\}$. Though they are enforceable individually, the policy that we obtain by taking the product of both the SA $\varphi_1 \cap \varphi_2$ is not enforceable. Suppose that the first event is (1, 1)*, the output of the enforcer will be* (1, 1)*, and the state of the product automaton will be updated to (q_1, l_1). When in state (q_1, l_1), whatever may be the input event, it is not possible to correct it (as there will be no path to an accepting state from the state (q_1, l_1) in the product automaton).*

As discussed in the problem description in the introduction, we focus on how to add a new enforcement layer incrementally to the existing enforcer (e.g., when a new property to be enforced is identified due to a new threat). How can an enforcer for the new property be composed with an existing enforcer such that the composed enforcement system, along with the new policy, will also continue to enforce all the previously enforced policies is what we explore and study in this work.

In our framework, since the framework allows editing of events, and since enforcers are bi-directional, it is not possible to compose enforcers (as per Definition 6) for the following reasons:

- As illustrated in Figure 1, the enforcer is bi-directional. Firstly, input from the environment is read (and edited/corrected if necessary), which is fed to the program, and the resulting output of the program is later checked (and edited if necessary) by the enforcer before it is forwarded to the environment. This does not allow the enforcers that we have to be composed directly in series.
- Moreover, since editing of events is allowed, the edit/correction made by one enforcer may not be compatible w.r.t the other enforcer, where there may be some edit/correction that is suitable for both policies to be enforced.

Thus we need to revisit the definition of the enforcement function, which will also be suitable for incremental enforcement schemes.

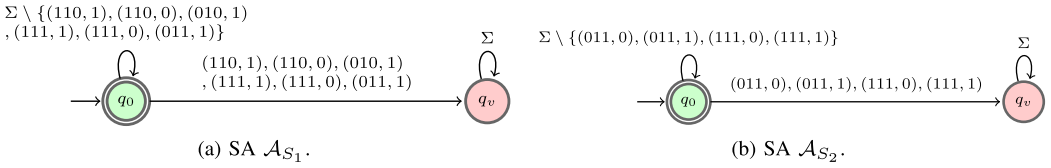
B. INCREMENTAL COMPOSITION OF SECURITY ENFORCERS

Suppose that we rely on the internals of the enforcement function, which composes an input enforcement function and an output enforcement function, then we can consider:

- composing all the input enforcement functions in series, where the (corrected) input that is released by the last function is fed to the program.

[2]Here, $\Sigma = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. So $\Sigma \setminus (1, 1) = \{(0, 0), (0, 1), (1, 0)\}$.

[3]Here, $\Sigma = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$. So $\Sigma \setminus (0, 0) = \{(0, 1), (1, 0), (1, 1)\}$.



(a) SA $\mathcal{A}_{S_1}$. (b) SA $\mathcal{A}_{S_2}$.

FIGURE 4. Safety automaton for S_1 and S_2.

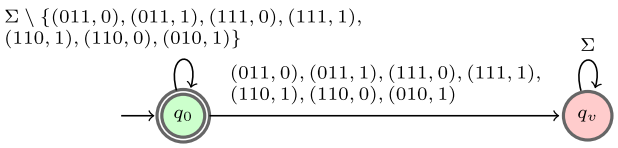
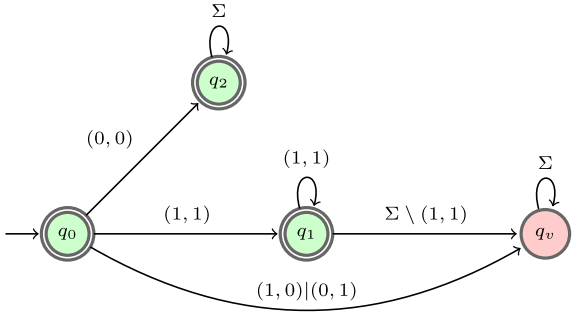
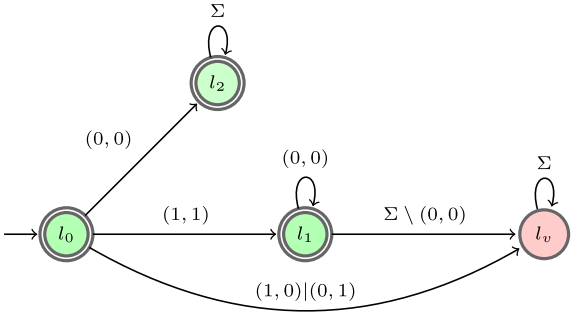


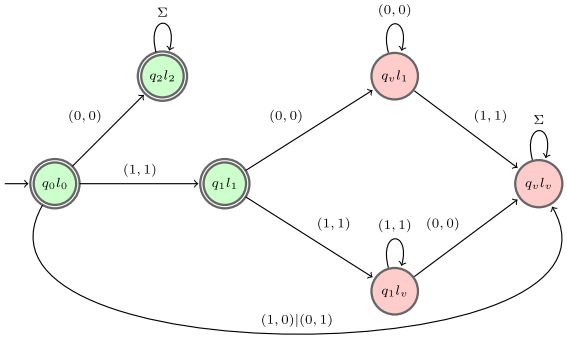
FIGURE 5. $\mathcal{A}_{S_1 \cap S_2}$: Product of Automaton S_1 and S_2.



(a) SA defining φ_1- Enforceable [2]



(b) SA defining φ_2- Enforceable [3]



(c) SA defining $\varphi_1 \cap \varphi_2$

FIGURE 6. Policy $\varphi_1 \cap \varphi_2$ is non-enforceable.

- similarly, the output enforcement functions can also be combined in series, and the corrected output released by the last function can be emitted to the environment.

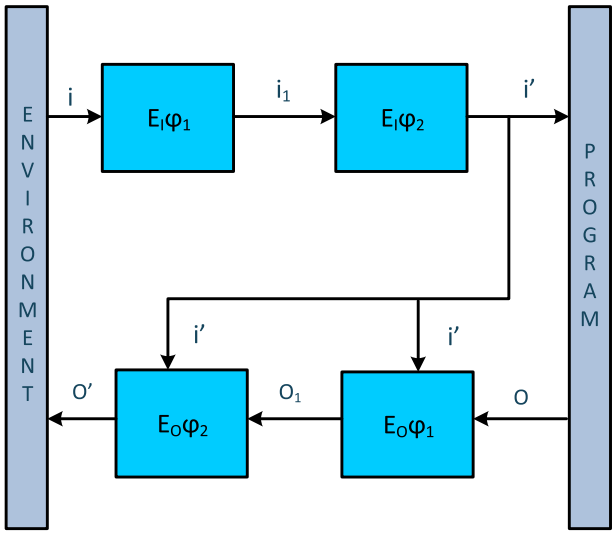


FIGURE 7. Incremental enforcement via serial composition

The incremental scheme by composing enforcers in series is shown in Figure 7.

Given two policies φ_1 and φ_2 (where φ_{1I} and φ_{2I} are their corresponding input policies), we can synthesize input and output enforcement functions for each of these policies $E_I\varphi_1$, $E_I\varphi_2$, $E_O\varphi_1$, and $E_O\varphi_2$, and then compose all the input enforcers and all the output enforcers. The composed input enforcer can be then combined with the composed output enforcer (see Figure 7).

We denote this type of incremental composition of enforcers in series as $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$. In this section we investigate whether $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ generally enforces $\varphi_1 \cap \varphi_2$. We are also interested to see whether the final output that we obtain using the incremental composition approach is equal to the output we would obtain using the monolithic approach.

Let us now formally define incremental composition in series of two enforcers.

Definition 7 (Incremental enforcement via serial composition): Let $E_I\varphi_1 : \Sigma_I^* \to \Sigma_I^*$ *(resp.* $E_I\varphi_2 : \Sigma_I^* \to \Sigma_I^*$*) be the input enforcement function for policy* $\varphi_{1I} \subseteq \Sigma_I^*$ *(resp.* φ_{2I}*), and let* $E_O\varphi_1 : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ *(resp.* $E_O\varphi_2$*) be the output enforcement function for policy* $\varphi_1 \subseteq \Sigma^*$ *(resp.* φ_2*). Their serial composition is a new enforcer* $E_{\varphi_1} \Rrightarrow E_{\varphi_2} : \Sigma^* \to \Sigma^*$ *defined as follows*:

$$\forall \sigma \in \Sigma^*, (E_{\varphi_1} \Rrightarrow E_{\varphi_2})(\sigma) = E_O\varphi_2(\sigma_I', \sigma_O').$$

with $\sigma_I' = E_I\varphi_2(E_I\varphi_1(\sigma_I))$*, and* $\sigma_O' = E_O\varphi_1(\sigma_I', \sigma_O)$.

As per the composition Definition 7, the output of the serial composition of the enforcers is the output emitted from the output enforcer $E_O\varphi_2$. The input emitted from the input enforcer $E_I\varphi_2$ is considered as the final corrected input. The output enforcer $E_O\varphi_1$ is invoked with the corrected input σ_I' from the serially composed input enforcers and the output σ_O of the reactive system. The corrected output of the enforcer is input to the output enforcer $E_O\varphi_2$, which finally emits the output to the environment.

Remark 7: Definition 7 is formulated in order to support incrementally adding a new enforcement layer. Suppose that we only have the input and output enforcement functions w.r.t policy φ_1 and a new policy to be enforced φ_2 is given. We can obtain enforcement functions for φ_2 individually and compose with the existing enforcement functions as per Definition 7.

Note that serial composition of enforcers as per Definition 7 does not always work. That is, though given two policies φ_1, φ_2 and also $\varphi_1 \cap \varphi_2$ are all enforceable, the serial composition of enforcers of φ_1 and φ_2 as per the above definition may not work. The final output obtained may not satisfy $\varphi_1 \cap \varphi_2$. Moreover, there may also be situations where other constraints, such as instantaneity, may be violated. Let us consider input enforcement to understand this (similar reasoning also applies for output enforcement). As per the serial composition definition (Def. 7), the input emitted from the input enforcer $E_I\varphi_2$ is considered as the final corrected input, but the final input selected by the function $E_I\varphi_2$ may violate the policy monitored by the input enforcer $E_I\varphi_1$.

Let us consider the following example to understand this further.

Example 6 (Serial composition as per Definition 7 does not always work): Let us again consider policies S_1 and S_2 illustrated in Figure 4, presented again in Figure 8. Both policies S_1 and S_2 are enforceable individually. The policy $S_1 \cap S_2$ is also enforceable. However, when we compose input and output enforcers for these policies in series as per Definition 7, the final output obtained may not satisfy policy $S_1 \cap S_2$. Also, there may be situations where constraints such as instantaneity may be violated. For example, consider the word $(100, 1) \cdot (110, 1) \cdot (011, 0)$ *to be processed incrementally. In the first step,* $(100, 1)$ *satisfies both policies and thus will be emitted as it is. In the second step, let us consider that the altered input produced by the second input enforcement function is* 110. *When* 110 *is fed as input to the output enforcement function of policy S_1, there is no possible output event that it can release to satisfy policy S_1. The incremental processing of the considered word is shown in Table 3.*

In this section, we have seen that the enforcement function in Definition 6 is not always suitable for the incremental security enforcement by composing enforcers in series, also when we consider first composing all the input enforcement functions, followed by the composition of the output enforcement functions (Definition 7).

We thus revisit the incremental security enforcement scheme in the next section, to propose an incremental scheme that can tackle all the enforceable properties.

VI. REVISITING INCREMENTAL SECURITY ENFORCEMENT SCHEME

In this section we propose an incremental composition scheme. First, we define the following Select functions and later present how the compositional scheme can be defined using the Select functions.

A. SELECT FUNCTIONS

- **$\mathsf{SelectI}_{\varphi_I}(\sigma_I, X)$**: Given an input word $\sigma_I \in \Sigma_I^*$, and a set of input events $X \subseteq \Sigma_I$, $\mathsf{SelectI}_{\varphi_I}(\sigma_I, X)$ is the set of input events x that belong to set X such that the word obtained by extending σ_I with x satisfies policy φ_I. Formally,

$$\mathsf{SelectI}_{\varphi_I}(\sigma_I, X) = \{x \in X : \sigma_I \cdot x \models \varphi_I\}.$$

  Considering the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the set of events in X that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:

$$\mathsf{SelectI}_{\mathcal{A}_{\varphi_I}}(q, X) = \{x \in X : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$

  For example, let us consider the input automaton corresponding to policy P in Figure 3a. Initially, when $\sigma_I = \epsilon$ we have $X = \{00, 01, 10, 11\}$, and $\mathsf{SelectI}_P(\epsilon, X) = \{00, 01, 10\}$. If we consider $\sigma_I = 00 \cdot 01 \cdot 01$, and $X = \{00, 01, 10\}$, we have $\mathsf{SelectI}_P(00 \cdot 01, X) = \{00, 01, 10\}$. Also, $q_0 \xrightarrow{00\cdot01}_I q_0$, and $\mathsf{SelectI}_P(q_0, \{00, 01, 10\}) = \{00, 01, 10\}$.
- **$\mathsf{SelectO}_{\varphi}(\sigma, x, Y)$**: Given an input-output word $\sigma \in \Sigma^*$, an input event $x \in \Sigma_I$, and a set of output events $Y \subseteq \Sigma_O$, $\mathsf{SelectO}_{\varphi}(\sigma, x, Y)$ is the set of output events y in Y s.t. the input-output word obtained by extending σ with (x, y) satisfies policy φ. Formally,

$$\mathsf{SelectO}_{\varphi}(\sigma, x, Y) = \{y \in Y : \sigma \cdot (x, y) \models \varphi\}.$$

  Considering the automaton $\mathcal{A}_{\varphi} = (Q, q_0, q_v, \Sigma, \rightarrow)$ defining policy φ, and an input event $x \in \Sigma_I$, the set of output events y in Y that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ with (x, y) is defined as:

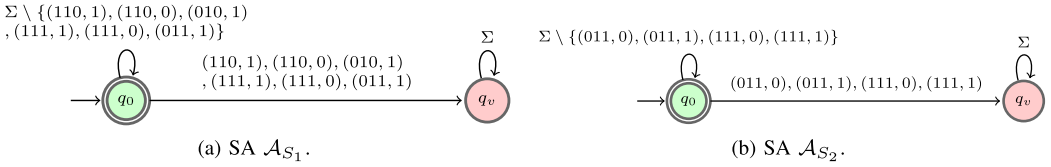
$$\mathsf{SelectO}_{\mathcal{A}_{\varphi}}(q, x, Y) = \{y \in Y : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v\}.$$

  For example, consider policy P illustrated in Figure 3. We have $\mathsf{SelectO}_P(q_0, 01, \{0, 1\}) = \{0\}$.

  $\mathsf{minD}(x, X')$ (resp. **$\mathsf{minD}(y, Y')$**): Consider X' (resp. Y') as a set of input (resp. output) events acceptable to all policies φ, and x (resp. y) as the original input (resp. output). $\mathsf{minD}(x, X')$ (resp. $\mathsf{minD}(y, Y')$) non-deterministically selects an edit $x' \in X'$ (resp. $y' \in Y'$) such that it is of minimum deviation from the original input event x (resp. output event y).

TABLE 3. Serial composition using Definition 7.

σ_I	σ_O	$\sigma'_I = E_I S1(\sigma_I)$	$\sigma''_I = E_I S2(\sigma'_I)$	$\sigma'_O = E_O S1(\sigma''_I, \sigma_O)$	$E_O S2(\sigma''_I, \sigma'_O)$
100	1	100	100	$(100, 1)$	$(100, 1)$
$100 \cdot 110$	$1 \cdot 1$	$100 \cdot 011$	$100 \cdot 110$	$(100, 1) \cdot ??$	–



(a) SA $\mathcal{A}_{S_1}$. (b) SA $\mathcal{A}_{S_2}$.

FIGURE 8. Safety automaton for S_1 and S_2.

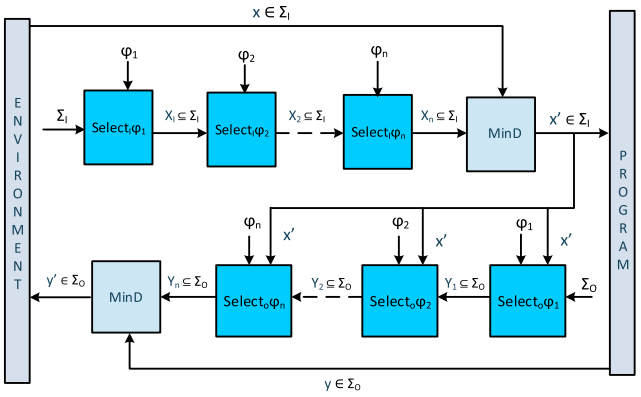


FIGURE 9. Incremental Enforcement via Serial Composition using Select functions

B. INCREMENTAL ENFORCEMENT SCHEME USING SELECT FUNCTIONS

In serial composition using Select, the input enforcer E_I is implemented by the serial composition of SelectI() followed by MinD(). The input set Σ_I is fed to SelectI() in serial to pre-compute a valid input set satisfying all policies. As shown in Figure 8, $\mathsf{SelectI}_{\varphi_1}$ produces input set X_1 according to policy φ_{1I} from input Σ_I. Similarly, $\mathsf{SelectI}_{\varphi_2}$ outputs set X_2 w.r.t policy φ_{2I} from input X_1 and so on. The final input set obtained X_n satisfying all policies $\varphi_1, \varphi_2 \cdots \varphi_n$ is input to MinD(). Whenever an input x is received from the environment, it is fed to MinD() that selects a minimal edit $x' \in X_n$ such that $\sigma_I \cdot x'$ (σ_I already output) satisfies input policies.

Similarly, the serial composition of output enforcer E_O is implemented with the serial composition of SelectO() followed by MinD(). The output of input enforcer a' along with all possible output Σ_O are fed to SelectO() in serial to pre-compute a valid output set satisfying all policies. As shown in Figure 8, $\mathsf{SelectO}_{\varphi_1}$ chooses output set Y_1 according to policy φ_1 for input x'. Similarly, $\mathsf{SelectO}_{\varphi_2}$ chooses set Y_2 according to policy φ_2 from input Y_1 and so on. The final output set obtained Y_n satisfying all policies $\varphi_1, \varphi_2 \cdots \varphi_n$ is input to MinD(). Whenever an output y is received from the program, it is fed to MinD() that selects a minimal edit $y' \in Y_n$ such that $\sigma \cdot (x', y')$ (σ_I already output) satisfies policies.

Definition 8 (Incremental enforcement via serial composition using select): Given two properties φ_1 and φ_2 (where φ_{1I} and φ_{2I} are their corresponding input policies), we define the enforcement function $E_{\varphi_1} \Rrightarrow E_{\varphi_2} : \Sigma^* \to \Sigma^*$ as $E_O(E_I(\sigma_I), \sigma_O)$ where:

- $E_I : \Sigma_I^* \to \Sigma_I^*$ *is defined as*:

$$E_I(\epsilon_{\Sigma_I}) = \epsilon_{\Sigma_I}$$
$$E_I(\sigma \cdot a) = \sigma'_I \cdot MinD(a, \mathsf{SelectI}_{\varphi_2}(\sigma'_I, (\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I))))$$

  where $\sigma'_I = E_I(\sigma)$.

- $E_O : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ *is defined as*:

$$E_O(\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O}) = \epsilon_{\Sigma}$$
$$E_O(\sigma_I \cdot x, \sigma_O \cdot y) = \sigma' \cdot (x, y')$$

  where $\sigma' = E_O(\sigma_I, \sigma_O)$
  $y' = MinD(y, \mathsf{SelectO}_{\varphi_2}(\sigma', x, \mathsf{SelectO}_{\varphi_1}(\sigma', x, \Sigma_O)))$.

Note that the serial composition of enforcers using Select functions always works. That is, given two policies φ_1, φ_2 and also $\varphi_1 \cap \varphi_2$ are all enforceable, serial composition of enforcers of φ_1 and φ_2 as per the above definition works and the final output obtained will always satisfy $\varphi_1 \cap \varphi_2$. Let us consider input enforcement to understand this (similar reasoning also applies to output enforcement). As per the serial composition Definition 8, all possible inputs Σ_I are fed to the input enforcers in serial composition, and the set obtained (using SelectI()) is a valid one satisfying all input policies ($\varphi_1$and φ_2 in this case). When a new input word is received, it is input to the MinD() that chooses (if required) a suitable element from the valid set available with it.

Let us consider the following example to understand this further.

Example 7 (Serial composition scheme using Select()): Let us again consider policies S_1 and S_2 illustrated in Figure 4. Both policies S_1 and S_2 are enforceable individually. The policy $S_1 \cap S_2$ is also enforceable. Also, when we compose input and output enforcers for these policies in series as per Definition 8, the final output obtained does satisfy policy

TABLE 4. Serial composition scheme using Select()-input enforcement.

σ_I	σ_O	$X_1 = \mathsf{SelectI}_{S1}(\sigma_I', \Sigma_I)$	$X' = \mathsf{SelectI}_{S2}(\sigma_I', X_1)$	$x' = MinD(x, X')$	σ_I'
100	1	$\mathsf{SelectI}_{S1}(\epsilon_I, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(\epsilon_I, X_1) = \{100, 101, 010, 001\}$	$MinD(100, \{100, 101, 010, 001\}) = 100$	100
$100 \cdot 110$	$1 \cdot 1$	$\mathsf{SelectI}_{S1}(100, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(100, X_1) = \{100, 101, 010, 001\}$	$MinD(110, \{100, 101, 010, 001\}) = 100$	$100 \cdot 100$
$100 \cdot 110 \cdot 011$	$1 \cdot 1 \cdot 0$	$\mathsf{SelectI}_{S1}(100.100, \Sigma_I) = \{100, 101, 010, 001, 011\}$	$\mathsf{SelectI}_{S2}(100 \cdot 100, X_1) = \{100, 101, 010, 001\}$	$MinD(011, \{100, 101, 010, 001\}) = 001$	$100 \cdot 100 \cdot 001$

TABLE 5. Serial composition scheme using Select()-output enforcement.

$Y' = \mathsf{SelectO}_{S1}(\sigma', x', \Sigma_O)$	$Y'' = \mathsf{SelectO}_{S2}(\sigma', x', Y')$	$MinD(y, Y'')$	σ'
$\mathsf{SelectO}_{S1}(\epsilon, 100, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}(\epsilon, 100, Y') = \{0, 1\}$	$MinD(1, \{0, 1\}) = 1$	$(100, 1)$
$\mathsf{SelectO}_{S1}((100, 1), 100, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}((100, 1), 100, Y') = \{0, 1\}$	$MinD(1, \{0, 1\}) = 1$	$(100, 1) \cdot (100, 1)$
$\mathsf{SelectO}_{S1}((100, 1) \cdot (100, 1), 001, \Sigma_O) = \{0, 1\}$	$\mathsf{SelectO}_{S2}((100, 1) \cdot (100, 1), 001, Y') = \{0, 1\}$	$MinD(0, \{0, 1\}) = 0$	$(100, 1) \cdot (100, 1) \cdot (001, 0)$

$S_1 \cap S_2$. *For example consider the word* $(100, 1) \cdot (110, 1) \cdot (011, 0)$ *to be processed incrementally is shown in Tables 4 and 5. Whenever any input is given, the MinD() always selects a valid element to input to the output enforcement function, always satisfying all the policies.*

Theorem 2 (Serial composition using select): Consider two policies φ_1, φ_2 defined as SA, and where $\varphi = \varphi_1 \cap \varphi_2$. If policy φ is enforceable, then $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 is an enforcer for φ (satisfies all the constraints as per Definition 4).

The proof of Theorem 2 is given in Appendix A.The proof is based on induction on the length of the input word σ.

Remark 8: Definition 8 supports incrementally adding a new enforcement layer. Whenever another new policy to be enforced φ_i is given, we can obtain the input and output select functions for the policy φ_i individually, and these can easily be plugged-in to enforce φ_i in addition to the previously enforced policies.

Remark 9 (ORDER OF COMPOSITION OF ENFORCERS DOES NOT MATTER): The order in which the input (resp. output) enforcers are composed, does not affect the final outcome in the proposed incremental enforcement approach. From the definition 8, we can see that each enforcer from the input set computes the set of all valid events w.r.t its policy, which is fed to the next enforcer in the sequence. The output set produced by the last enforcer in the sequence satisfies all the policies from which one element is chosen by MinD. Thus the order of input (resp. output) enforcers do not matter in the proposed incremental enforcement scheme.

VII. CASE STUDY

In this section, we introduce a swarm of drones that we take as our case study. In our setting, we consider a swarm controller that gets information about the environment from the drone systems and sends control signals to the drones based on that information. We illustrate how enforcement can mitigate attacks from malicious actors by introducing a range of policies.

A. DRONE SWARMS

In Shanghai, the 2020 New Year was marked with a swarm of drones lighting the night sky with a range of firework-like displays. Subsequently, QR codes and brand logos have also been recreated. Also, in 2021, over 3200 drones, each behaved as a single pixel and together created a complete image in Shanghai [42]. Precisely choreographed drone swarms will become more common as drones are increasingly ubiquitous.

These playful examples are not all; delivery and transport services are also exploring drone applications [43]. An example is Alphabet's Wing project [44], which has up to 50 drones operating at the test facility in San Francisco, and beta customers in the US and Australia receiving drone deliveries.

B. ATTACKS ON DRONES

The entertaining displays and practical benefits of transport services are not risk-free as they are an opportunity for malicious actors. Recent work by Yaacoub et al. in [8] surveys a range of attacks on drone systems. WiFi-based jamming and deauthentiacation attacks have been demonstrated to allow man-in-the-middle snooping and spoofing, and denial of service attacks [45]. Other RF jamming methods can trigger landing [46] and prevent communication [47]. Such jamming attacks prevent the flow of status and control signals between drones and the central controller; this could cause a range of unsafe outcomes. Hardware trojans in communication systems [48] can provide a platform for injection, interception, and alteration attacks. Attackers could cause drones to violate airspace boundaries, colliding with other aircraft, people, or property.

In this work, we consider four potential attacks on a drone swarm QR code:

- A1 *Boundary Breach (Injection and Alteration)*: where the attacker instructs the drone(s) to breach airspace clearances.
- A2 *Overwhelm Inputs (Injection and Alteration)*: an attack where drone(s) are overwhelmed with simultaneous conflicting control signals.
- A3 *Block Control Signals (Jamming)*: any attack which prevents updated control signals from reaching one or more drone(s) in the swarm.
- A4 *Drain Batteries (Injection and Alteration)*: where the attacker sets motor RPM above that selected for efficient use of battery during the display.
- A5 *Shared Boundary (Alteration)*: where the attacker forces drones 1 and 2 to hover at their shared boundary.

We group attacks into common attack types of *jamming* and *injection and alteration*. We group *injection and alteration*

as such attacks add or alter control packets, and *jamming* removes packets. To defend against these attacks, we develop policies that are enforced using the proposed enforcement framework.

C. MITIGATION WITH ENFORCEMENT

We focus on mitigating the four attacks introduced previously, which originate from a compromised controller. These attacks interfere with communication with one or more drones in the swarm. To mitigate these attacks, we design policies that synthesise to incremental enforcers that ensure the policy is satisfied. The policies are designed to prevent the impact of the attacks. The incremental enforcers are placed between the swarm and controller as illustrated in Figure 10 to allow editing of status and control signals before they are emitted to the controller and drones, respectively.

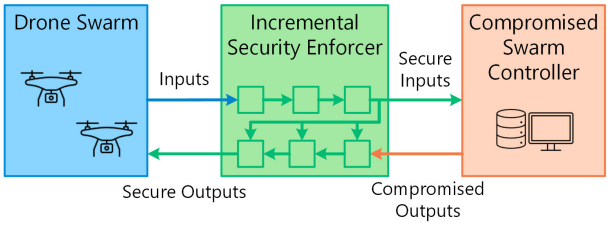


FIGURE 10. System diagram of incremental enforcement for a drone swarm.

For our example of a swarm that displays a 16 × 16 pixel QR code at 200m above ground level,[4] we simplify drones operation into two-dimensional space (X, Y) as illustrated in Figure 11.

We consider an example airspace clearance given to the swarm to have limits (X_limit, Y_limit). Within the cleared airspace, the QR code is created. Each drone is assigned a two-meter by two-meter set of limits (min_y, max_y, min_x, max_x) for the particular pixel it represents. A space of one meter between each pixel ensures drones operating at their airspace limits do not collide. The total airspace needed for the QR code is 47 by 47 meters, with X and Y starting at zero. For the display at 200 meters above ground level, the Y_limit is 247 meters, and X_limit is 47 meters.

We also consider the drone motor rotations per minute (RPM). We use the specifications for an off-the-shelf motor (EMAX MT2213 935KV motor with 8045 paddles [49]) to determine the RPM limit of 7825 to ensure that the motors consume less than 100W.

This gives the set of per drone inputs (status signals) I ={min_x_limit, max_x_limit, min_y_limit, max_y_limit, rpm_limit}. The drones are controlled with binary signals for increasing and decreasing in each dimension and RPM. This

[4]Our example is synthetic, inspired by real QR code displays. As, to the best of our knowledge, no public drone QR code telemetry is available, we have selected limits that mimic reality.

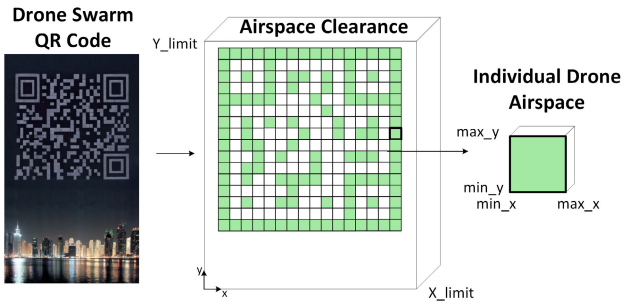


FIGURE 11. Example allocation of airspace to a drone swarm and individual drones.

gives the set of per drone outputs O = {x_up, x_down, y_up, y_down, rpm_up, rpm_down}.

We introduce one security policy per attack introduced in Section VII-B:

- Policy φ_1 for mitigating attack A1, *Boundary Breach* is as follows:
  - max_x_limit and x_up should not occur simultaneously
  - max_y_limit and y_up should not occur simultaneously
  - min_x_limit and x_down should not occur simultaneously
  - min_y_limit and y_down should not occur simultaneously
- Policy φ_2 for mitigating attack A2, *Overwhelm Inputs* is as follows:
  - x_up and x_down should not occur simultaneously
  - y_up and y_down should not occur simultaneously
  - rpm_up and rpm_down should not occur simultaneously
- Policy φ_3: The drone should descend to minimum altitude when control packets are not received for 5 seconds (mitigating attack A3, *Block Control Signals*).
- Policy φ_4: rpm_up and rpm_limit should not occur simultaneously (mitigating attack A4, *Drain Batteries*).
- Policy φ_5: max_x_1_limit and min_x_2_limit should not occur simultaneously without x_1_down or x_2_up (mitigating A5, *Shared Boundary* attack for drones 1 and 2).

1) POLICY φ_1 MITIGATING ATTACK A1, *BOUNDARY BREACH*

The *Boundary Breach* attack instructs a drone to breach airspace clearances or separation between drones. These airspace clearances are set as the upper and lower limits of X and Y dimensions in the policy φ_1, illustrated in Figure 12, which prevents a boundary breach. This policy transitions to violating location (lv) when the control signals which increase (respectively decrease) X or Y when at or above the upper (respectively lower) limit. Limits are defined per drone. The synthesised enforcer will, therefore, prevent violations by suppressing any control signals which would cause a violating transition, thus preventing any breach of the boundaries.

For readability, our notation here differs from earlier sections. We provide only those inputs and outputs which are important for understanding the transition; others are omitted and could be of any value. For example in policy φ_1, transition from $l0$ to lv (on x_down & min_x_limit) means the transition is taken for all elements of Σ where x_down is and min_x_limit is true.

Policy φ_1 has an initial location (q_0) of $l0$ and only one other location lv which is the non-accepting (violating) location (q_v). There are four transitions of interest, from $l0$ to lv, when the drone is at a limit and a control signal to breach that limit is present. For the lower limits, transitions on x_down & min_x_limit and y_down & min_y_limit when the drone is at the minimum X (respectively Y) and is instructed to decrease further. For the upper limits, transitions x_up & max_x_limit and y_up & max_y_limit when the drone is at the maximum X (respectively Y) and is instructed to increase further. While these violating conditions are not met, the policy remains in $l0$.

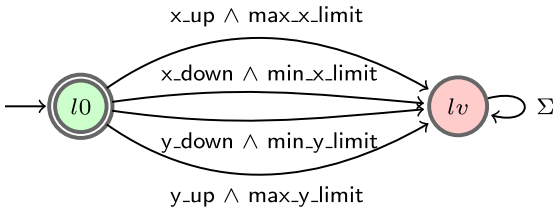


FIGURE 12. Automaton for policy φ_1 which prevents a drone from breaching X and Y limits.

2) POLICY φ_2 MITIGATING ATTACK A2, *OVERWHELM INPUTS*

In attack A2, *Overwhelm Inputs*, the attacker sends conflicting control signals to the drone. Policy φ_2 is designed such that simultaneously present conflicting controls cause violation. The synthesised enforcer will therefore suppress one of these conflicting signals to ensure the drone is not overwhelmed.

Policy φ_2 has an inital location (q_0) of $l0$ and only one other location lv which is the non-accepting violating location (q_v). There are three transitions of interest, from $l0$ to lv, when a control packet contains simultaneous inputs for a single channel. For the X dimension, a transition on x_down & x_up cause a violation. For the Y dimension, a transition on y_down & y_up cause a violation. For the RPM control, transitions on rpm_down & rpm_up cause a violation. While these violating conditions are not met the policy remains in $l0$.

3) POLICY φ_3 MITIGATING ATTACK A3, *BLOCK CONTROL SIGNALS*

The attack A3, *Block Control Signals*, which jams signals prevents the drone from getting updated controls. This could result in it continuing to climb or move in undesirable ways. The policy detects if no control signals are sent in any 5-second period. If the timeout is detected, the drone is instructed to descend to the minimum altitude (min_x_limit). The policy ensures that the y_down control signal is present until the drone has descended to min_x_limit at which point the drone is considered safe until receiving the next valid control signal. This could be landing the drone if min_x_limit is 0 for the particular drone.

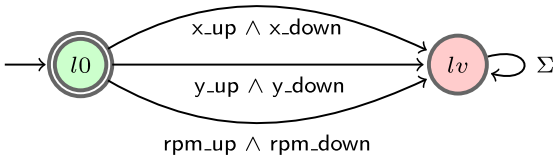


FIGURE 13. Automaton for policy φ_2 which prevents conflicting control signals being set to the drone.

Policy φ_3 has locations (Q) $l0$, $l1$, $l2$, and lv, with an inital location (q_0) of $l0$ and the non-accepting violating location (q_v) lv. Initially, in $l0$, when any input/output event in Σ is present, the transition to $l1$ is taken. This sets the clock t to 0; this tracks the time between events. If any input/output packet in Σ is received, the self-transition is taken from $l1$ to $l1$, resetting the clock t. If no packet is received, and t is less than 5 seconds, the self-transition is taken without resetting the clock t. If the clock is greater than 5 seconds, and the control signal y_down is absent, the violating transition from $l1$ to lv is taken. To avoid this, the control signal y_down must be present, which results in the transition from $l1$ to $l2$. In $l2$, absence of y_down and min_y_limit results in a violating transition from $l2$ to lv. To avoid this, the control output signal y_down must be present while the input status signal min_y_limit is absent; this results in the self transition from $l2$ to $l2$. Once min_y_limit is present, a transition from $l2$ to $l0$ is taken, and the drone is at the minimum altitude limit.

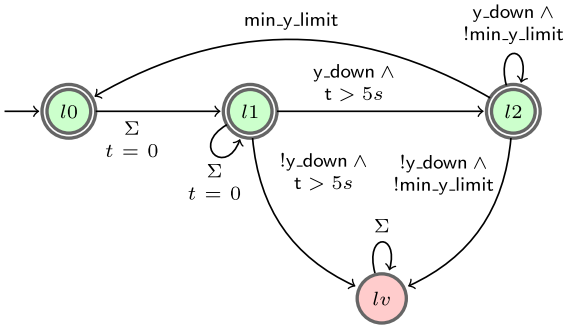


FIGURE 14. Automaton for policy φ_3 which mitigates a jamming attack.

4) POLICY φ_4 MITIGATING ATTACK A4, *DRAIN BATTERIES*

The attack A4, *Drain Batteries*, uses injection or alteration attacks to increase drone RPM above the efficiency limit, which ensures adequate flight endurance for the QR display. While the higher RPM will result in faster movement, it is less efficient and will drain the battery faster. For this reason, an RPM threshold is set. For our selected motor, this is set at 7825 RPM to ensure that the motors consume less

than 100W at 14.8V. Policy φ_3 is a simple policy defined to be violated when the RPM is increased with rpm_up above the threshold at_max_rpm. The synthesised enforcer will, to prevent violation, suppress the rpm_up signal when at_max_rpm is present.

Policy φ_3 has an inital location (q_0) of $l0$ and only one other location lv which is the non-accepting violating location (q_v). There is one transition of interest, when rpm_up & at_max_rpm a transition from $l0$ to lv is taken, causing a violation. While this condition is not met, the policy remains in $l0$.

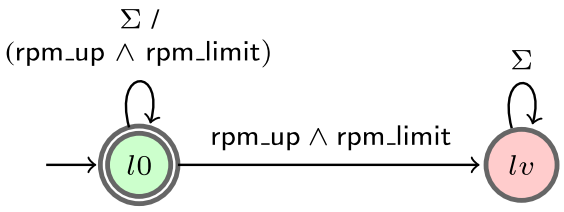


FIGURE 15. Automaton for policy φ_4 which prevents a drone from breaching RPM limits for maximum efficiency.

5) POLICY φ_5 MITIGATING ATTACK A5, *BOUNDARY HOVER*

The attack A5, *Shared Boundary*, uses alteration attacks to force drones to hover at a shared boundary. If drones hover near each other along shared boundaries, a gust of wind or inertia from maneuvers could cause boundary breaches and collisions. Drones 1 and 2 share a boundary along the airspace Y axis at the highest X for drone 1 and lowest X for drone 2. Policy φ_5 is responsible for ensuring either or both drones take evasive maneuvers away from this shared boundary.

Policy φ_5 has an inital location (q_0) of $l0$ and one other location lv which is the non-accepting violating location (q_v). There is one transition of interest, from $l0$ to lv, when drone 1 is at $\mathsf{max\_x_1\_limit}$, drone 2 is at $\mathsf{min\_x_2\_limit}$, and neither drone is moving away from the shared boundary, $\mathsf{!x_1\_down}$ & $\mathsf{!x_2\_up}$. While this condition is not met, the policy remains in $l0$.

These policies can be replicated for any number of drones, and shared airspace boundaries, in a swarm; however, we demonstrate the scalability issues with monolithic composition for even two drones in the following section.

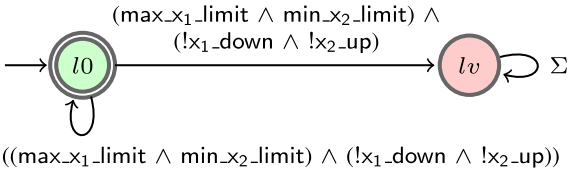


FIGURE 16. Automaton for policy φ_5 which prevents drones 1 and 2 from hovering at the shared airspace boundary.

VIII. IMPLEMENTATION AND EVALUATION

A. IMPLEMENTATION

To evaluate the proposed approach of incremental serial composition, we implemented a simple simulator of a two-drone swarm. This follows the illustration in Figure 10 with a controller, enforcer, and drone swarm. This implementation focuses on comparing the monolithic and the proposed incremental serial composition. To do this, we deliberately compromise our simulated controller, which requires the enforcers to perform edits consistently. This allows us to examine the time taken to execute the enforcers.

The software tool *easy-rte* [23] was used to produce enforcers for this implementation. The tool provides support for monolithic composition, which we use to produce our monolithic enforcers. Additionally, we have extended the compiler to produce incremental series enforcers as per our framework. We call this new tool *incremental-easy-rte* and make the source code available.[5] The structure of the incremental enforcement is defined as follows:

```
Begin simulation
while tick <= max_tick do
    input = environment();
    goodInputs = inEnf();
    input = minD(input, goodInputs);
    input = controller(input);
    goodOutputs = outEnf(input);
    output = minD(output, goodOutputs);
    environment(output);
    tick+ = 1;
end while;
```

The simulator runs in ticks. A logical clock tick is in which the environment, input enforcement, controller, and output enforcement steps are completed. These steps are repeated until max_tick is reached. In our simulation, max_tick is set to 10 million, which means 10 million cycles of environment input, enforcement, and controller output are executed. The first step is sensing input from the environment with *environment*() where the drones status signals (X, Y, and RPM) are obtained, then performing input enforcement on these signals with the **Select** function *inEnf* () and **Edit** function *minD*(). The controller, *controller*(), is then run to obtain control signals, acceleration, and holding for each dimension and RPM. The output enforcer is run with the **Select** function *outEnf* () and **Edit** function *minD*() to select the final output with minimum deviation from the controller's original signals. Finally, these output control signals are exposed to the environment.

To require the enforcers to take action, the controller was artificially compromised. This controller constantly emitted all control signals, which would cause the drone to accelerate in all dimensions, eventually violating the airspace boundary and RPM limitation. The composed enforcer intervened when the control signal would cause such a violation, preventing it from being sent to the drones.

[5]The compiler *easy-rte-incremental*, a fork of *easy-rte*, can be accessed here https://github.com/PRETgroup/easy-rte-incremental. The source code includes the drone swarm QR code case study and other examples which can be compiled and run.

To demonstrate the impact of an increasingly complex composition, we started with a single φ_1 policy which defends against the boundary breach attack A1 for the first drone, then subsequently added φ_2, φ_3, and φ_4 policies for a single drone which defend against attacks A2, A3, and A4 respectively. The first drone is the lowest and leftmost pixel in the display.[6]

We then repeat these policies for a second drone in the swarm, which is one pixel to the right of the first drone. This requires new instances of the policies with different limits[7] and are denoted with subscript b, $\varphi_{1b} \ldots \varphi_{4b}$.

Therefore we created nine enforcers for both monolithic and serial compositions to satisfy the following:

$E_{\varphi 1}$: enforces φ_1
$E_{\varphi 2}$: enforces φ_1, and φ_2
$E_{\varphi 3}$: enforces φ_1, φ_2, and φ_3
$E_{\varphi 4}$: enforces φ_1, φ_2, φ_3, and φ_4
$E_{\varphi 5}$: enforces φ_1, φ_2, φ_3, φ_4, and φ_{1b}
$E_{\varphi 6}$: enforces φ_1, φ_2, φ_3, φ_4, φ_{1b}, and, φ_{2b}
$E_{\varphi 7}$: enforces φ_1, φ_2, φ_3, φ_4, φ_{1b}, φ_{2b}, andφ_{3b}
$E_{\varphi 8}$: enforces φ_1, φ_2, φ_3, φ_4, φ_{1b}, φ_{2b}, φ_{3b}, and φ_{4b}
$E_{\varphi 9}$: enforces φ_1, φ_2, φ_3, φ_4, φ_{1b}, φ_{2b}, φ_{3b}, φ_{4b}, and φ_5

Monolithic compositions were composed into a single policy first and are denoted $\varphi_{1M}, \varphi_{2M}, \ldots, \varphi_{9M}$. These are then synthesized into enforcers denoted $E_{\varphi_{1M}}$. Serial compositions are denoted $E_{\varphi_{1S}}, E_{\varphi_{2S}}, \ldots, E_{\varphi_{9S}}$.

Remark 10: Let us suppose that we initially have $E_{\varphi 4}$, and later a new policy to be enforced φ_{1b} is given. To obtain an enforcer using the monolithic approach, all the properties enforced by $E_{\varphi 4}$ should be known in detail, and we are building a new enforcer to enforce all the properties in this case.

1) MONOLITHIC COMPOSITION

The monolithic composition (product) of policies was computed by *easy-rte* for each of the following combinations of drone policies:

φ_{1M}: φ_1
φ_{2M}: $\varphi_1 \times \varphi_2$
φ_{3M}: $\varphi_1 \times \varphi_2 \times \varphi_3$
φ_{4M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4$
φ_{5M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_{1b}$
φ_{6M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_{1b} \times \varphi_{2b}$
φ_{7M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_{1b} \times \varphi_{2b} \times \varphi_{3b}$
φ_{8M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_{1b} \times \varphi_{2b} \times \varphi_{3b} \times \varphi_{4b}$
φ_{9M}: $\varphi_1 \times \varphi_2 \times \varphi_3 \times \varphi_4 \times \varphi_{1b} \times \varphi_{2b} \times \varphi_{3b} \times \varphi_{4b} \times \varphi_5$

Product compositions for φ_{6M}, φ_{7M}, φ_{8M}, and φ_{9M} were not successfully compiled using *easy-rte* due to a lack of support for large compositions resulting in compilation timeouts. This is discussed further in Section VIII.

The successfully compiled monolithic compositions were synthesized into C runtime enforcers using *easy-rte*. The resulting five enforcers ($E_{\varphi 1M} \cdots E_{\varphi 5M}$) could be used in the system illustrated in Figure 10 for simulation. This provided a baseline by which we compare our proposed serial composition.

2) INCREMENTAL COMPOSITION VIA SERIAL COMPOSITION

The serial composition requires a modified enforcer structure. The single enforcer from the monolithic composition (responsible for all policies) is now replaced with multiple enforcers, responsible for a single policy, and with different **Select** functions, *inEnfS*() and *outInfS*(). The high-level simulator is the same produced by *easy-rte* as used in the monolithic enforcers. As shown in the structure below:

```
Begin simulation
while tick <= max_tick do
    input = environment();
    for all input enforcers do
        goodInputs = inEnfS(goodInputs);
    end for
    input = minD(input, goodInputs);
    output = controller(input);
    for all output enforcers do
        goodOutputs = outEnfS(input, goodOutputs);
    end for
    output = minD(output, goodOutputs);
    environment(output);
    tick+ = 1;
end while;
```

Serial enforcement requires a loop to sequentially execute the **Select** functions for each enforcer. Each enforcer considers the current location of its internal monitor and the set of possible inputs *goodInputs* or outputs *goodOutputs* provided to it. Based on this, it produces a subset of inputs or outputs that would continue to satisfy the internal monitor. This can be passed on to the next enforcer or to the **minD** function. This is as previously discussed in detail in Section VI-A.

To demonstrate the impact of adding polices the following enforcers were used:

$E_{\varphi 1S}$: E_{φ_1}
$E_{\varphi 2S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$
$E_{\varphi 3S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3}$
$E_{\varphi 4S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4}$
$E_{\varphi 5S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4} \Rrightarrow E_{\varphi_{1b}}$
$E_{\varphi 6S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4} \Rrightarrow E_{\varphi_{1b}} \Rrightarrow E_{\varphi_{2b}}$
$E_{\varphi 7S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4} \Rrightarrow E_{\varphi_{1b}} \Rrightarrow E_{\varphi_{2b}} \Rrightarrow E_{\varphi_{3b}}$
$E_{\varphi 8S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4} \Rrightarrow E_{\varphi_{1b}} \Rrightarrow E_{\varphi_{2b}} \Rrightarrow E_{\varphi_{3b}} \Rrightarrow E_{\varphi_{4b}}$
$E_{\varphi 9S}$: $E_{\varphi_1} \Rrightarrow E_{\varphi_2} \Rrightarrow E_{\varphi 3} \Rrightarrow E_{\varphi_4} \Rrightarrow E_{\varphi_{1b}} \Rrightarrow E_{\varphi_{2b}} \Rrightarrow E_{\varphi_{3b}} \Rrightarrow E_{\varphi_{4b}} \Rrightarrow E_{\varphi_5}$

Remark 11: Let us again consider that we initially have $E_{\varphi 4S}$, and later, a new property to be enforced φ_{1b} is given. In our incremental composition approach, we only synthesize the required enforcement select functions for φ_{1b}, and they are integrated into the enforcement framework. This does not require detailed knowledge of the properties enforced

[6]The first drone's limits are as follows: min_x_limit = 0m; max_x_limit = 2m; min_y_limit = 200m; max_y_limit = 202m; rpm_limit = 7825 RPM.

[7]The second drone's limits are as follows: min_x_limit = 3m; max_x_limit = 5m; min_y_limit = 200m; max_y_limit = 202m; rpm_limit = 7825 RPM.

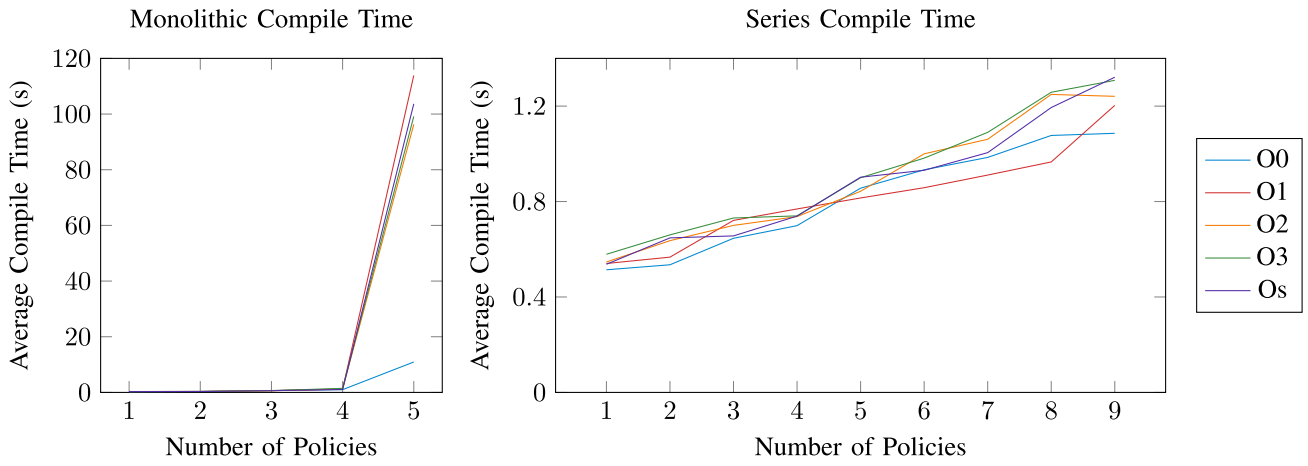


FIGURE 17. Results comparing compile time in seconds as the number of policies composed increases. Note both the axes scales differ.

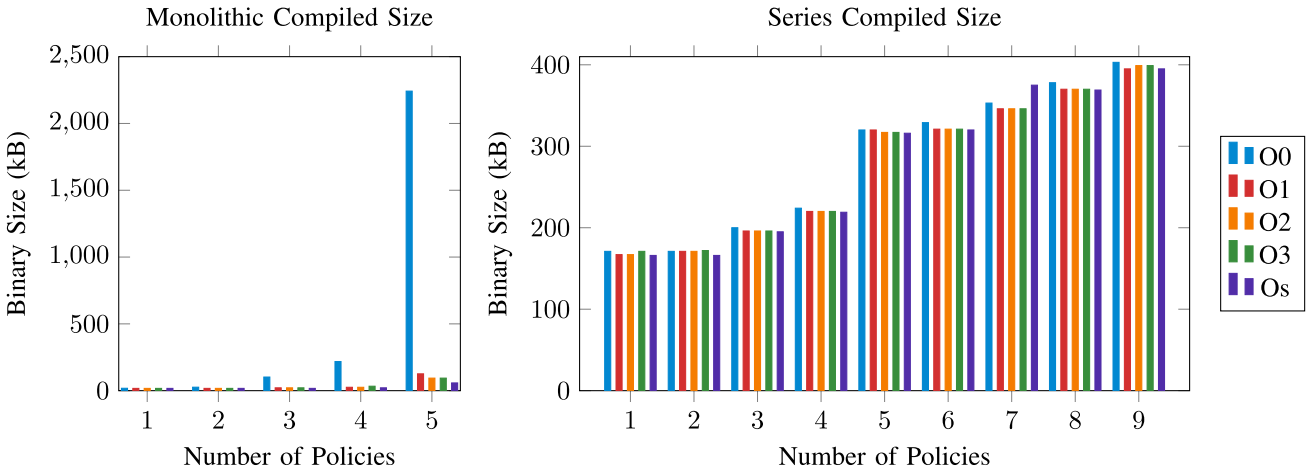


FIGURE 18. Results comparing compiled binary size in kilobytes as the number of policies composed increases. Note both axes scales differ.

by $E_{\varphi 4S}$. The new select functions of φ_{1b} can be plugged-in at the end of existing select functions in the incremental enforcement framework.

B. EVALUATION

Enforcers are correct by definition, this is *Soundness* as defined in Definition 4, and both methods of composition satisfy it. Therefore, to evaluate and compare the performance of the proposed serial composition, we report on the compile time, compiled size, and the maximum time taken for the enforcer to execute per tick for both composition approaches. Compile time is measured using the GCC flag *-fitme-report* and averaged over three compilations; this was repeated with optimization flags *-O0*, *-O1*, *-O2*, *-O3*, and *-Os*. From *-O0* to *-O3* the level of optimization is increased and *-Os* optimises for binary size rather than execution time.[8] Compiled size is the kilobyte size of the compiled executable with each level of optimization. Execution time was measured over 10 million ticks, repeated 10 times, and the longest execution time per tick is reported. These results are reported for each of $E_{\varphi 1}$, $E_{\varphi 2}$, $E_{\varphi 3}$, $E_{\varphi 4}$, $E_{\varphi 5}$, $E_{\varphi 6}$, $E_{\varphi 7}$, $E_{\varphi 8}$, and $E_{\varphi 9}$ to show the impact of policy complexity on the performance metrics.

Due to memory management limitations of *easy-rte*, the monolithic compositions for φ_{6M}, φ_{7M}, φ_{8M} and, φ_{9M} could not be computed. This means enforcers $E_{\varphi_{6M}}$, $E_{\varphi_{7M}}$, $E_{\varphi_{8M}}$ and, $E_{\varphi_{9M}}$ could not be synthesised. While alterations to *easy-rte* may have increased the complexity of policies it could compute, there will remain a state-space explosion challenge which can be illustrated with the five compositions that were able to be computed as shown in the following results.

1) COMPILE TIME

The compile time is reported in Figure 17, serial results on the left chart, and monolithic on the right chart. All series compilation takes less than 1.4 seconds, with a visible linear trend. Increasing the level of optimisation, in general,

[8]Additional information about optimization flags can be found in the GCC documentation available online https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

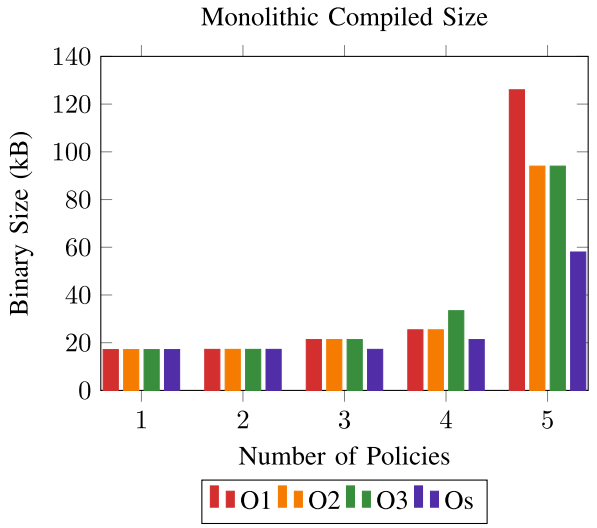


FIGURE 19. Monolithic compiled binary size in kilobytes as the number of policies composed increases with *-O0* results removed for clarity.

slightly increases the compile time. The monolithic composition displays exponential trends, with more rapid growth by higher optimization settings. This result confirms the intuition that the monolithic approach's exponentially increasing code complexity, with far more locations and transitions, is reflected in compilation time. For example, the compile time of the seven and eight policies series compositions is around 1 second, which is similar to the four-policy monolithic composition, and two orders of magnitude less than the five-policy monolithic composition.

2) COMPILED BINARY SIZE

The compiled binary size, measured in kilobytes, kB, is reported in Figures 18 and 19. Series results on the left chart and monolithic on the right chart of Figure 18. Figure 19 removes the no optimization (*-O0*) results from the monolithic composition to better illustrate the trend in optimised binary sizes.

The serial composition displays a piece-wise linear trend as the number of composed policies increases. A discontinuity between four and five composed policies is shown. This is considered a reflection of the relative complexity between the added policies. Between compositions four, three, and four, the simple policy φ_4 is added. Between compositions four and five, the more complex policy $\varphi_1 b$ is added. The monolithic composition displays an exponentially increasing trend in compile size as complexity increases; this is illustrated to apply to optimized compilations in Figure 19.

In general, increasing optimization has a positive effect on both series and monolithic compositions. However, optimization *-O3* slightly increases compiled size in compositions series one, five, and eight, and monolithic four. This reflects the added optimizations, which may increase compile size but reduce execution time. The size-focused optimization *-Os* shows consistently reduced binary size. The trends meet the intuition of state space explosion for monolithic composition.

3) ENFORCER EXECUTION TIME

The enforcer execution time, reported in nanoseconds (ns), is reported in Figure 20, serial results on the left chart, and monolithic on the right chart. Additionally, Figure 21 reports on monolithic execution time with no optimization (*-O0*) removed to more clearly show optimized results. The serial composition displays a linear trend, with optimization resulting in significantly lower recorded times and more gentle positive trend.

The monolithic composition displays an exponential increase execution time as policy complexity increases, with higher optimization reducing the execution time and trend. The size-focused optimization *-Os* shows longer execution times than *-O2* and-*O3* which reflects the objective of *-Os* to minimise binary size. Generally, monolithic execution time is two orders of magnitude lower than series. The exponential increase in monolithic execution time was unexpected, given the resulting enforcers only evaluate a single automaton's transitions per tick. This increase may reflect the exponential increase in the complexity of the automaton's transitions which need to be evaluated.

These results are consistent with the intuition that additional enforcers placed in series increases execution time above that of monolithic.

4) DISCUSSION

The recorded results were consistent with widely observed state space explosion in monolithic composition and with our intuition that increasing policies in serial composition would result in longer execution time but not suffer from state space explosion. The lack of state space explosion in the serial composition is due to individual policies remaining independent, which means the number of locations and transitions considered is the sum of those in each policy, in contrast to monolithic where the product is taken, and so locations and transitions grow exponentially.

The exponential trend in monolithic execution time was not expected. As discussed earlier, this may be due to the transition condition complexity increases exponentially. Finding the point where the complexity of a single transition in a large monolithic composition is slower than executing the equivalent series composition remains a point of interest for future work. Prior to this point being found, the faster execution time of monolithic compositions is preferable in situations where the response time is more important than compile time or size. This trade-off must be balanced by designers.

The practical limitations of monolithic compositions were found in our use of *easy-rte* as it was unable to compile the most complex four. While improvements to memory management in *easy-rte* may enable greater monolithic compositions, where compile time and binary size are important, series compositions are preferable. Memory-limited embedded applica-

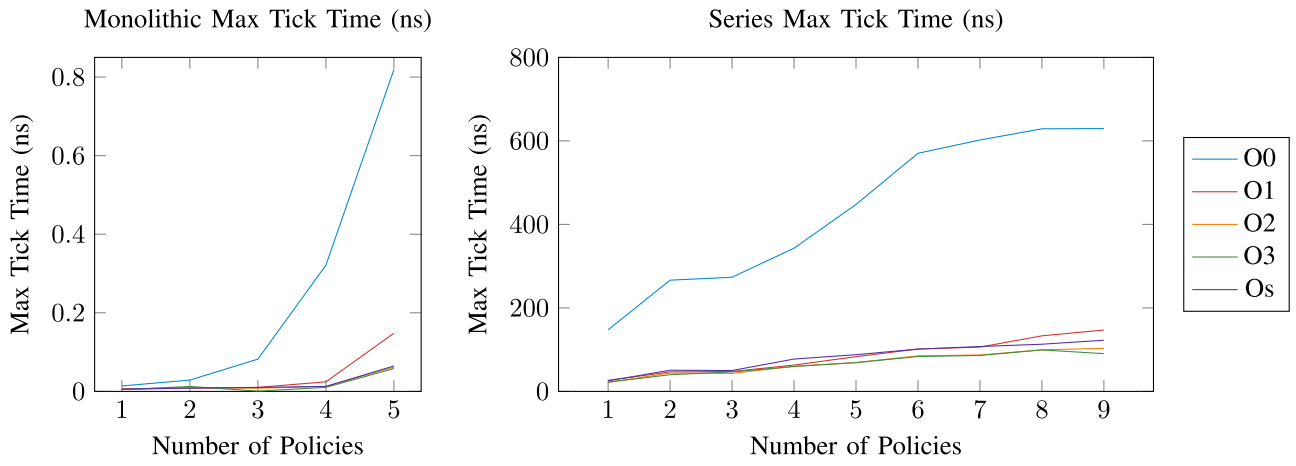


FIGURE 20. Results comparing the maximum time it takes for an enforcer to run as the number of policies composed increases. Note both axes scales differ.

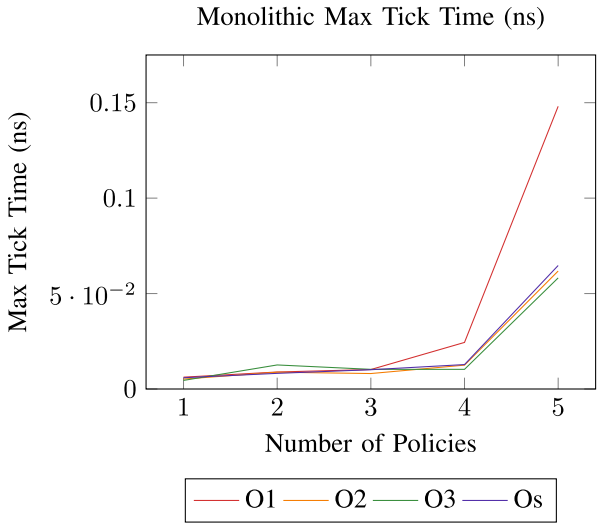


FIGURE 21. The maximum time it takes for monolithic enforcer to run each tick as the number of policies composed increases with *-O0* results removed for clarity.

tions such as enforcers deployed on the drones from our case study are an example where series is preferable.

IX. CONCLUSION AND FUTURE WORK

As security policies evolve pertaining to security attacks in cyber-physical systems, its worth investigating how the security framework needs to adapt. The monolithic technique, in which the framework must be developed from scratch, does not appear to be efficient. In this work, we investigate the serial compositionality of runtime enforcers in response to the rise of new security policies. We consider the bi-directional enforcement mechanism in a synchronous reactive system. We propose an approach for the composition of enforcers in series. We show that using the proposed compositional framework, enforcers can be composed serially and can be used to enforce any set of policies that can be enforced using the monolithic approach. As a result, the suggested framework enables the gradual insertion of additional enforcement and security layers as needed (for instance, whenever a new security-related risk or issue arises) without affecting the policies that have already been in action. The results of our evaluation and analysis employing a set of policies in the context of drone swarms show that serially composed enforcers do not suffer from the state space explosion that a monolithic approach does. Our results for the proposed serial composition approach clearly illustrate a linear relationship between compile time, compilation size, and execution time as the number of policies grows.

In the future, we would like to study other schemes supporting the composition of enforcers, such as parallel composition (executing all the enforcers in parallel and feeding in the same input to them).

APPENDIX A

proof of Theorem 1: Let us recall Theorem 1. Consider two policies φ_1, φ_2 defined as SA, and $\varphi = \varphi_1 \cap \varphi_2$. If policy φ_1 or policy φ_2 is non-enforceable, then $\varphi_1 \cap \varphi_2$ is non-enforceable.

Proof: Let us recall the condition of enforceability. Consider a property φ defined as SA $\mathcal{A}_\varphi = (Q, q_0, q_v, \Sigma, \rightarrow)$. Property φ is enforceable iff the condition (*EnfCo*) holds which is as follows: $\forall q \in Q, q \neq q_v \implies \exists (x, y) \in \Sigma : q \xrightarrow{(x,y)} q' \wedge q' \neq q_v$.

Let policy φ_1 is represented by SA $\mathcal{A}_{\varphi_1} = (Q^1, q_0^1, q_v^1, \Sigma, \rightarrow_1)$ and policy φ_2 is defined by SA $\mathcal{A}_{\varphi_2} = (Q^2, q_0^2, q_v^2, \Sigma, \rightarrow_2)$.

The policy $\varphi_1 \cap \varphi_2$ is defined as the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2} = (Q, q_0, q_v, \Sigma, \rightarrow)$ where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, and the transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ with $((q^1, q^2), a, (q'^1, q'^2)) \in \rightarrow$ if $(q^1, a, q'^1) \in \rightarrow^1$ and $(q^2, a, q'^2) \in \rightarrow_2$.

Note that in the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ representing $\varphi_1 \cap \varphi_2$, all the locations in $(Q^1 \times q_v^2) \cup (q_v^1 \times Q^2)$ are trap locations. All such locations can be merged into a single violating location labeled as q_v.

Let us suppose that one of the policies (say φ_1[9]) among φ_1 and φ_2 is non-enforceable. Since φ_1 is non-enforceable, there exists a location $q^1 \in Q^1 \setminus \{q_v^1\}$ such that all the outgoing transitions from q^1 go to q_v^1. That is, $\exists q^1 \in Q^1 : (q^1 \neq q_v^1) \wedge (\forall (x, y) \in \Sigma : q^1 \xrightarrow{(x,y)} q_v^1)$.

Thus, in the product SA $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ representing $\varphi_1 \cap \varphi_2$ will have a location such that all the outgoing transitions from it go to violating location, i.e., $\exists (q^1, q^2) \in Q : (q^1, q^2) \neq q_v \wedge \forall (x, y) \in \Sigma : (q^1, q^2) \xrightarrow{(x,y)} q_v$. Thus, the policy $\varphi_1 \cap \varphi_2$ is also non-enforceable.

If the policy φ_2 is non-enforceable, following the above analogies, the policy $\varphi_1 \cap \varphi_2$ will be non-enforceable. Hence Theorem 1 holds. □

proof of Theorem 2 We prove that given two policies φ_1, φ_2 defined as SA, and where $\varphi = \varphi_1 \cap \varphi_2$, if policy φ is enforceable, then $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 is an enforcer for φ (satisfies all the constraints (*Snd*), (*Tr*), (*Mono*), (*Inst*), and (*Cau*) constraints as per Definition 4).

Let us prove this theorem using induction on the length of the input sequence $\sigma \in \Sigma^*$.

Induction basis. Theorem 2 holds for $\sigma = (\epsilon_{\Sigma_I}, \epsilon_{\Sigma_O})$ since the function will not release any input-output event as output and thus $E_O(E_I(\epsilon_{\Sigma_I}), \epsilon_{\Sigma_O}) = \epsilon_\Sigma$.

Induction step. Assume that for every $\sigma = (x_1, y_1) \cdots (x_k, y_k) \in \Sigma^*$ of some length $k \in \mathbb{N}$, let $E_\varphi(\sigma) = (x'_1, y'_1) \cdots (x'_k, y'_k) \in \Sigma^*$, and Theorem 2 holds for σ, i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ satisfies the (*Snd*), (*Tr*), (*Mono*), (*Inst*), and (*Cau*) constraints. We have $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \in \varphi$ and $E_I(\sigma_I) \in \varphi_I$. Let us denote $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ using σ', and $E_I(\sigma_I)$ using σ'_I.

We now prove that for any event $(x_{k+1}, y_{k+1}) \in \Sigma$, Theorem 2 holds for $\sigma \cdot (x_{k+1}, y_{k+1})$, where $x_{k+1} \in \Sigma_I$ is the input event, and $y_{k+1} \in \Sigma_O$ is the output event. We have the following two possible cases based on whether $E_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \in \varphi_1 \cap \varphi_2$.

- $E_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \in \varphi_1 \cap \varphi_2$.
  Using Lemma 1, we also have $E_I(\sigma_I) \cdot x_{k+1} \in \varphi_I$. Thus, from the Definitions of *MinD* and SelectI, we have

  $$MinD(x_{k+1}, \mathsf{SelectI}_{\varphi_2}(\sigma'_I, (\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I)))) = x_{k+1}$$

  .
  From the definition of *MinD* and SelectO, we have $MinD(y_{k+1}, \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O))) = y_{k+1}$.
  So the output of the enforcer is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$.
  Regarding constraint (*Snd*), in this case, what has been already released as output by the enforcer earlier before reading event (x_{k+1}, y_{k+1}) (i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$) followed by the new input-output event released as output (x_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$, and thus constraint (*Snd*) holds.
  Regarding constraint (*Mono*), it holds since $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$ and also $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.
  Regarding constraint (*Inst*) from the induction hypothesis, we have for σ of some length k, $|\sigma| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint (*Inst*) holds.
  Constraint (*Tr*) holds in this case since the output of the enforcer before reading (x_{k+1}, y_{k+1}) i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ followed by the new input-output event read (x_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$ and we already saw that the output event released by the enforcer $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 up on reading (x_{k+1}, y_{k+1}) is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$.
  Regarding constraint (*Cau*), from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_\varphi$, we have $x_{k+1} \in X = \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, X)$. Also, we have $y_{k+1} \in Y = \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O)$, and $y_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, Y)$.
  Theorem 2 thus holds for $\sigma \cdot (x_{k+1}, y_{k+1})$ in this case.
- $E_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \notin \varphi_1 \cap \varphi_2$.
  In this case, we have two sub-cases.
  - $E_\varphi(\sigma) \cdot x_{k+1} \in \varphi_I$
    Thus, from the Definitions of *MinD* and SelectI, we have

    $$MinD(x_{k+1}, \mathsf{SelectI}_{\varphi_2}(\sigma'_I, (\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I)))) = x_{k+1}.$$

    Now, since $E_\varphi(\sigma) \cdot (x_{k+1}, y_{k+1}) \notin \varphi_1 \cap \varphi_2$, we can have $y'_{k+1} \in \sigma_O$ from the Definitions of *MinD* and SelectO, i.e.,
    $MinD(y_{k+1}, \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O))) = y'_{k+1}$.
    So the output of the enforcer is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y'_{k+1})$.
    Regarding constraint (*Snd*), in this case, what has been already released as output by the enforcer earlier before reading event (x_{k+1}, y_{k+1}) (i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$) followed by the new input-output event released as output (x_{k+1}, y'_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$, and thus constraint (*Snd*) holds.
    Regarding constraint (*Mono*), it holds since $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$ and also $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.
    Regarding constraint (*Inst*) from the induction hypothesis, we have for σ of some length k, $|\sigma| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y'_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint (*Inst*) holds.
    Constraint (*Tr*) holds in this case since the output of the enforcer before reading (x_{k+1}, y_{k+1}) i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ followed by the new input-output event read

[9] Similar reasoning applies when we consider that φ_2 is non-enforceable.

(x_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$ and we already saw that the output event released by the enforcer $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 up on reading (x_{k+1}, y_{k+1}) is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)\cdot(x_{k+1}, y'_{k+1})$. Regarding constraint (*Cau*), from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x_{k+1} \in X \subseteq \Sigma_I = \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, X)$. Also we have, $y'_{k+1} \in Y \subseteq \Sigma_O = \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O)$, and $y'_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, Y)$.
Theorem 2 thus holds for $\sigma \cdot (x_{k+1}, y_{k+1})$ in this case.

– $E_{\varphi}(\sigma) \cdot x_{k+1} \notin \varphi_I$
Thus, from the Definitions of *MinD* and $\mathsf{SelectI}$, we have

$$MinD(x_{k+1}, \mathsf{SelectI}_{\varphi_2}(\sigma'_I, (\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I)))) = x'_{k+1}.$$

Now, we have two sub-cases.

∗ $E_{\varphi}(\sigma) \cdot (x'_{k+1}, y_{k+1}) \in \varphi_1 \cap \varphi_2$.
From the definition of *MinD* and $\mathsf{SelectO}$, we have $MinD(y_{k+1}, \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O))) = y_{k+1}$.
So the output of the enforcer is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y_{k+1})$.
Regarding constraint (*Snd*), in this case, what has been already released as output by the enforcer earlier before reading event (x_{k+1}, y_{k+1}) (i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$) followed by the new input-output event released as output (x'_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$, and thus constraint (*Snd*) holds.
Regarding constraint (*Mono*), it holds since $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$ and also $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.
Regarding constraint (*Inst*) from the induction hypothesis, we have for σ of some length k, $|\sigma| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint (*Inst*) holds.
Constraint (*Tr*) holds in this case since the output of the enforcer before reading (x_{k+1}, y_{k+1}) i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ followed by the new input-output event read (x_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$ and we already saw that the output event released by the enforcer $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 up on reading (x_{k+1}, y_{k+1}) is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$.
Regarding constraint (*Cau*), from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x'_{k+1} \in X = \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x'_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, X)$. Also we have, $y_{k+1} \in Y = \mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O)$, and $y_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, Y)$.
Theorem 2 thus holds for $\sigma \cdot (x_{k+1}, y_{k+1})$ in this case.

∗ $E_{\varphi}(\sigma) \cdot (x'_{k+1}, y_{k+1}) \notin \varphi_1 \cap \varphi_2$.
In this case, from the Definitions of *MinD* and $\mathsf{SelectO}$, we can have $MinD(y_{k+1}, \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O))) = y'_{k+1}$.
So the output of the enforcer is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$.
Regarding constraint (*Snd*), in this case, what has been already released as output by the enforcer earlier before reading event (x_{k+1}, y_{k+1}) (i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$) followed by the new input-output event released as output (x'_{k+1}, y'_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$, and thus constraint (*Snd*) holds.
Regarding constraint (*Mono*), it holds since $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$ and also $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.
Regarding constraint (*Inst*) from the induction hypothesis, we have for σ of some length k, $|\sigma| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint (*Inst*) holds.
Constraint (*Tr*) holds in this case since the output of the enforcer before reading (x_{k+1}, y_{k+1}) i.e., $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma)$ followed by the new input-output event read (x_{k+1}, y_{k+1}) satisfies the property $\varphi_1 \cap \varphi_2$ and we already saw that the output event released by the enforcer $E_{\varphi_1} \Rrightarrow E_{\varphi_2}$ as per Definition 8 up on reading (x_{k+1}, y_{k+1}) is $E_{\varphi_1} \Rrightarrow E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$.
Regarding constraint (*Cau*), from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x'_{k+1} \in X \subseteq \Sigma_I = \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x'_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, X)$. Also we have, $y'_{k+1} \in Y \subseteq \Sigma_O = \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O)$, and $y'_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, Y)$.
Theorem 2 thus holds for $\sigma \cdot (x_{k+1}, y_{k+1})$ in this case.

Hence Theorem 2 holds for $\sigma \cdot (x_{k+1}, y_{k+1})$. □

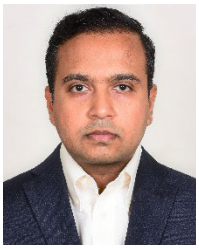
REFERENCES

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. 11th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput. (ISORC)*, May 2008, pp. 363–369.
[2] G. Loukas, *Cyber-Physical Attacks: A Growing Invisible Threat*. London, U.K.: Butterworth, 2015.
[3] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security—A survey," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1802–1831, Dec. 2017.
[4] *Predatory Sparrow Massively Disrupts Steel Factories While Keeping Workers Safe*. Accessed: Aug. 5, 2022. [Online]. Available: https://blog.malwarebytes.com/hacking-2/2022/07/predatory-sparrow-massively-disrupts-steel-factories-while-keeping-workers-safe/
[5] R. Langner, "To kill a centrifuge: A technical analysis of what stuxnet's creators tried to achieve," Langner Group, Munich, Germany, Tech. Rep., 2013.

[6] J. Slay and M. Miller, "Lessons learned from the Maroochy water breach," in *Proc. Int. Conf. Crit. Infrastruct. Protection*. New York, NY, USA: Springer, 2008, pp. 73–82.
[7] R. M. Lee, M. J. Assante, and T. Conway, "German steel mill cyber attack," *Ind. Control Syst.*, vol. 30, no. 62, pp. 1–15, 2014.
[8] J.-P. Yaacoub, H. Noura, O. Salman, and A. Chehab, "Security analysis of drones systems: Attacks, limitations, and recommendations," *Internet Things*, vol. 11, Sep. 2020, Art. no. 100218.
[9] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.
[10] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: Composition, synthesis, and enforcement abilities," *Formal Methods Syst. Des.*, vol. 38, no. 3, pp. 223–262, 2011.
[11] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, pp. 19:1–19:41, Jan. 2009.
[12] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, and O. N. Timo, "Runtime enforcement of timed properties revisited," *Formal Methods Syst. Des.*, vol. 45, no. 3, pp. 381–422, 2014.
[13] M. Ngo, F. Massacci, D. Milushev, and F. Piessens, "Runtime enforcement of security policies on black box reactive programs," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2015, pp. 43–54.
[14] H. Pearce, M. M. Y. Kuo, P. S. Roop, and S. Pinisetty, "Securing implantable medical devices with runtime enforcement hardware," in *Proc. 17th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design*, Oct. 2019, pp. 1–9.
[15] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Logic Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.
[16] G. Pola and M. D. Di Benedetto, "Control of cyber-physical-systems with logic specifications: A formal methods approach," *Annu. Rev. Control*, vol. 47, pp. 178–192, 2019.
[17] D. He, S. Chan, and M. Guizani, "Drone-assisted public safety networks: The security aspect," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 218–223, Aug. 2017.
[18] M. Yampolskiy, P. Horvath, X. D. Koutsoukos, Y. Xue, and J. Sztipanovits, "Taxonomy for description of cross-domain attacks on CPS," in *Proc. 2nd ACM Int. Conf. High Confidence Netw. Syst.*, Apr. 2013, pp. 135–142.
[19] S. Pinisetty and S. Tripakis, "Compositional runtime enforcement," in *Proc. NASA Formal Methods Symp. (NFM)*. Minneapolis, MN, USA: Springer, 2016, pp. 82–99.
[20] S. Pinisetty, A. Pradhan, P. Roop, and S. Tripakis, "Compositional runtime enforcement revisited," *Formal Methods Syst. Des.*, vol. 59, pp. 205–252, Oct. 2022.
[21] S. Pinisetty and S. Tripakis, "Compositional runtime enforcement," in *NASA Formal Methods*, S. Rayadurgam and O. Tkachuk, Eds. Cham, Switzerland: Springer, 2016, pp. 82–99.
[22] S. Pinisetty, P. S. Roop, S. Smyth, S. Tripakis, and R. V. Hanxleden, "Runtime enforcement of reactive systems using synchronous enforcers," in *Proc. 24th ACM SIGSOFT Int. SPIN Symp. Model Checking Softw.*, Santa Barbara, CA, USA, Jul. 2017, pp. 80–89.
[23] H. Pearce, S. Pinisetty, P. S. Roop, M. M. Y. Kuo, and A. Ukil, "Smart I/O modules for mitigating cyber-physical attacks on industrial control systems," *IEEE Trans. Ind. Informat.*, vol. 16, no. 7, pp. 4659–4669, Jul. 2020.
[24] S. Adepu, S. Shrivastava, and A. Mathur, "Argus: An orthogonal defense framework to protect public infrastructure against cyber-physical attacks," *IEEE Internet Comput.*, vol. 20, no. 5, pp. 38–45, Sep. 2016.
[25] S. Adepu and A. Mathur, "From design to invariants: Detecting attacks on cyber physical systems," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur. Companion (QRS-C)*, Jul. 2017, pp. 533–540.
[26] A. O. de Sá, L. F. R. da Costa Carmo, and R. C. S. Machado, "Covert attacks in cyber-physical control systems," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1641–1651, Aug. 2017.
[27] O. A. Beg, T. T. Johnson, and A. Davoudi, "Detection of false-data injection attacks in cyber-physical DC microgrids," *IEEE Trans. Ind. Informat.*, vol. 13, no. 5, pp. 2693–2703, Oct. 2017.
[28] Q. Sun, K. Zhang, and Y. Shi, "Resilient model predictive control of cyber–physical systems under DoS attacks," *IEEE Trans. Ind. Informat.*, vol. 16, no. 7, pp. 4920–4927, Jul. 2020.
[29] F. Farivar, M. S. Haghighi, A. Jolfaei, and M. Alazab, "Artificial intelligence for detection, estimation, and compensation of malicious attacks in nonlinear cyber-physical systems and industrial IoT," *IEEE Trans. Ind. Informat.*, vol. 16, no. 4, pp. 2716–2725, Apr. 2020.
[30] S. Kim, Y. Won, I.-H. Park, Y. Eun, and K.-J. Park, "Cyber-physical vulnerability analysis of communication-based train control," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6353–6362, Aug. 2019.
[31] F. Li, Y. Shi, A. Shinde, J. Ye, and W.-Z. Song, "Enhanced cyber-physical security in Internet of Things through energy auditing," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 5224–5231, Jun. 2019.
[32] E. Dolzhenko, J. Ligatti, and S. Reddy, "Modeling runtime enforcement with mandatory results automata," *Int. J. Inf. Secur.*, vol. 14, no. 1, pp. 47–60, 2015.
[33] A. Baird, H. Pearce, S. Pinisetty, and P. Roop, "Runtime interchange of enforcers for adaptive attacks: A security analysis framework for drones," in *Proc. 20th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design (MEMOCODE)*, Oct. 2022, pp. 1–11.
[34] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: Runtime enforcement for reactive systems," in *Proc. TACAS*, in Lecture Notes in Computer Science, vol. 9035. Berlin, Germany: Springer, 2015.
[35] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, pp. 1–43, May 2009.
[36] D. Lomsak and J. Ligatti, "PoliSeer: A tool for managing complex security policies," *J. Inf. Process.*, vol. 19, pp. 292–306, Jul. 2011.
[37] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong, "Expandable grids for visualizing and authoring computer security policies," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, Apr. 2008, pp. 1473–1482.
[38] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Proc. IEEE Symp. Secur. Privacy*, May 2000, pp. 177–187.
[39] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," in *Proc. IEEE Symp. Secur. Privacy*, May 1999, pp. 17–31.
[40] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proc. IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.
[41] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Algebraic Methodology and Software Technology (AMAST'93)*. London, U.K.: Springer, 1994, pp. 83–96.
[42] D. Hambling. (2021). *This Record-Breaking Shanghai Drone Display is a Show of Technological Strength*. [Online]. Available: https://www.forbes.com/sites/davidhambling/2021/04/06/why-this-record-breaking-drone-display-in-shanghai-is-a-show-of-technological-strength/?sh=903dfd82d534
[43] R. Kellermann, T. Biehle, and L. Fischer, "Drones for parcel and passenger transportation: A literature review," *Transp. Res. Interdiscipl. Perspect.*, vol. 4, Mar. 2020, Art. no. 100088.
[44] *Wing*. Accessed: Aug. 5, 2022. [Online]. Available: https://wing.com/
[45] O. Westerlund and R. Asif, "Drone hacking with Raspberry-Pi 3 and WiFi pineapple: Security and privacy threats for the Internet-of-Things," in *Proc. 1st Int. Conf. Unmanned Vehicle Syst.-Oman (UVS)*, Feb. 2019, pp. 1–10.
[46] A. H. Abunada, A. Y. Osman, A. Khandakar, M. E. H. Chowdhury, T. Khattab, and F. Touati, "Design and implementation of a RF based anti-drone system," in *Proc. IEEE Int. Conf. Informat., IoT, Enabling Technol. (ICIoT)*, Feb. 2020, pp. 35–42.
[47] P. Valianti, S. Papaioannou, P. Kolios, and G. Ellinas, "Multi-agent coordinated close-in jamming for disabling a rogue drone," *IEEE Trans. Mobile Comput.*, vol. 21, no. 10, pp. 3700–3717, Oct. 2022.
[48] A. Belous and V. Saladukha, "Hardware trojans in electronic devices," in *Viruses, Hardware and Software Trojans*. Cham, Switzerland: Springer, 2020, pp. 209–275.
[49] EMAX. *EMAX MT2213 935 kV Multicopter Brushless Motor*. Accessed: Aug. 5, 2022. [Online]. Available: https:/emaxmodel.com/products/emax-mt2213-935kv-multicopter-brushless-motor

ABHINANDAN PANDA received the M.Tech. degree in information and communication technologies from the Indian Institute of Technology Kharagpur, India, in 2014. He is currently pursuing the Ph.D. degree in computer systems engineering with the Indian Institute of Technology Bhubaneswar, India. His research interests include the theory of computation, formal methods, runtime verification, and enforcement and its application in the security of cyber-physical systems, and health monitoring.

ALEX BAIRD received the B.E. degree (Hons.) in computer systems engineering from The University of Auckland, Auckland, New Zealand, in 2019, where he is currently pursuing the Ph.D. degree in computer systems engineering. His research interests include the design, safety, and security of cyber-physical systems using formal methods, particularly runtime verification and enforcement.



SRINIVAS PINISETTY (Member, IEEE) received the master's degree in computer science from the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands, in 2009, and the Ph.D. degree in computer science from INRIA, University of Rennes 1, Rennes, France, in January 2015. He continued as a P.D.Eng. Trainee with TU/e for two years. For his master's thesis project, he worked with ASML, Veldhoven, The Netherlands, in 2009, and as a Software Design Engineer Trainee with the Océ Technologies, Venlo, The Netherlands, in 2011. He is currently an Assistant Professor with the School of Electrical Sciences, Indian Institute of Technology (IIT) Bhubaneswar, Bhubaneswar, India. Prior to joining IIT Bhubaneswar, he worked as a Postdoctoral Researcher with the University of Aalto, Espoo, Finland, and later with the University of Gothenburg and the Chalmers University of Technology, Gothenburg, Sweden. His research interests include formal methods, software engineering in general, and runtime verification and enforcement in particular.

PARTHA ROOP (Member, IEEE) received the B.E. degree in computer science and engineering from the College of Engineering, Anna University, Chennai, India, in 1989, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 1993, and the Ph.D. degree in computer science (software engineering) from the University of New South Wales, Sydney, NSW, Australia, in 2001. He is currently a Professor with the Department of Electrical, Computer and Software Engineering, The University of Auckland, Auckland, New Zealand. His research interests include the design and validation of cyber-physical systems using formal methods, including in digital health and artificial intelligence (AI) applications in cyber-physical systems.