Received 16 November 2023, accepted 5 January 2024, date of publication 23 January 2024, date of current version 31 January 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3357714*

RESEARCH ARTICLE

# Scalable Security Enforcement for Cyber Physical Systems

**ALEX BAIRD[1], ABHINANDAN PANDA[2], HAMMOND PEARCE[3], SRINIVAS PINISETTY[2], AND PARTHA ROOP[1], (Member, IEEE)**
[1]Department of Electrical, Computer and Software Engineering, The University of Auckland, Auckland 1010, New Zealand
[2]School of Electrical Sciences, Indian Institute of Technology (IIT) Bhubaneswar, Bhubaneswar 752050, India
[3]School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia

Corresponding author: Partha Roop (p.roop@auckland.ac.nz)

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 (IIT) Bhubaneswar Seed Grant SP093.

**ABSTRACT** The security of Cyber-Physical Systems (CPSs) is increasingly important as more and more of these systems are added to the Internet of Things (IoT). As we increase the complexity and connectivity of our smart systems, we likewise broaden their digital attack surface. Recorded attacks on CPSs have caused significant physical impacts making methods for mitigation of attacks of paramount importance. The use of runtime enforcement (RE) can prevent violation of security policies. Here, runtime enforcers intervene before the CPS is compromised. Two key challenges are presented: (1) for complex systems, methods for automatically composing multiple policies are lacking; and (2) runtime enforcers are themselves executed digitally—meaning they too could have potential security vulnerabilities. We present the first comprehensive runtime enforcement framework which addresses both challenges. It can compose a lot of security policies in parallel and synthesize these policies into the more trustworthy hardware layers of a system. This removes reliance on potentially vulnerable firmware and software layers. We demonstrate our approach with policies to mitigate a set of attacks on a Fused Filament Fabrication (FFF) 3D printer. The experimental results show linear growth in logic element and register usage as the number of policies increase. This compares favourably to the exponential state space explosion that occurs with the conventional approach of monolithic composition. Additionally, we find higher enforcer clock frequencies are possible with the proposed parallel approach compared to existing serial approaches.

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

## I. INTRODUCTION

Cyber-Physical Systems (CPSs) combine the physical and digital worlds, where embedded digital controllers interact with the physical world through environmental sensors and actuators [1]. Our modern world is reliant on their function—from when you turn your lights on in the morning, which requires electrical generation and distribution (smart grids), to having your smart coffee machine brew your morning coffee automatically based on your wake-up time, to your commute via car, bus, train, or e-bike, which is enhanced by integrated embedded systems for both their operation and in their manufacturing processes.

It is thus imperative that designers of CPSs take into account security when implementing their systems. This is not trivial, as illustrated by the range of high profile CPS attacks including the Stuxnet worm damaging Iranian centrifuges [2], the German Steel Mill attack, which prevented a blast furnace from shutting down and caused significant damage [3], and the ransomware attack on Colonial Pipeline, which caused serious disruption to gasoline supply in the United States and resulted in a multi-million dollar payout to the attackers [4].

However, attacks are not just limited to industrial plants, Internet of Things (IoT) devices have been compromised

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Yu.

and used to launch Distributed Denial of Service (DDoS) attacks [5], [6], weaknesses in over-the-counter drones have been demonstrated [7], and compromise of Additive Manufacturing (AM) can cause propeller defects which fail in flight [8].

In this work, we focus our attention on Fused Filament Fabrication (FFF) AM, known commonly as filament 3D printers. These devices have become increasingly ubiquitous, with a range of models available for hobbyist to commercial use. Filament 3D printers are CPSs as they sense the environment through sensors (for example, temperature), impact the environment through actuators (for example, heaters and motors) to create 3D objects, and are controlled by an embedded controller. These printers are subject to a number of threats, as illustrated in Figure 1.

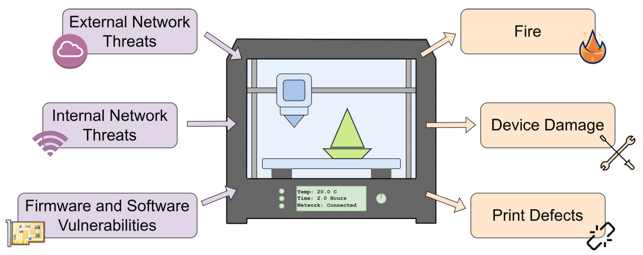


**FIGURE 1. Threats to the security of 3D printers and the potential impacts of successful attacks.**

For example, as 3D printers are connected to the internet for remote monitoring and control, they are exposed to internal and external network threats from malicious actors. Attackers could take advantage of security vulnerabilities in the printer's firmware or software to take control of the device. The attacker could then cause a range of damage potentially including: defects in printed objects, damage to the printer, and start a fire, which poses a significant safety risk. These threats and potential impacts follow a pattern shared by many CPS, where successful attacks have significant impact on the physical world.

Research in formal methods gives us a reliable mechanism for mitigating security vulnerabilities. The area of runtime verification (RV) [9], [10], [11], [12], [13] considers methods to generate runtime monitors which verify a set of policies while the system is running. If a policy is violated the monitor will raise an alarm. RV approaches are often applied where the systems are *black-box* (where the inputs and outputs of a system are known but internal behaviour and mechanics are not) or too complex for traditional model checking [14]. A key limitation of RV is that it fails to prevent the violation from occurring, and in security, violations cannot be tolerated. The area of runtime enforcement (RE) [15], [16], [17], [18], [19] extends these monitors to intervene before a violation occurs.

In this work, we consider FFF 3D printers as synchronous reactive cyber physical systems. Synchronous reactive systems are those which react to input stimuli and produce outputs continuously. Their function can be separated into *logical ticks* that consist of reading inputs, performing computation, and emitting outputs. Following the *synchrony hypothesis*, these *logical ticks* are considered as atomic events which occur infinitely faster than the environment produces input stimuli.

The system view of a synchronous reactive bidirectional enforcer is illustrated in Figure 2. The enforcer is placed between the environment (labelled *Env.*) and controller (labelled *Ctrl.*) such that it can observe and alter *Env.* inputs and *Ctrl.* outputs. For every *tick* of the controller, the enforcer first inspects *Env. Inputs I* and as necessary edits these to satisfy the set of policies ($\varphi$) before emitting *Safe and Secure Inputs I'* to the controller. The controller then executes it's reaction to these inputs and emits *Ctrl. Outputs O*. The output enforcer inspects and, as necessary to satisfy the policy ($\varphi$), edits these outputs to produce *Safe and Secure Outputs O'*, which are then exposed to the environment. This cycle repeats for every *tick*.

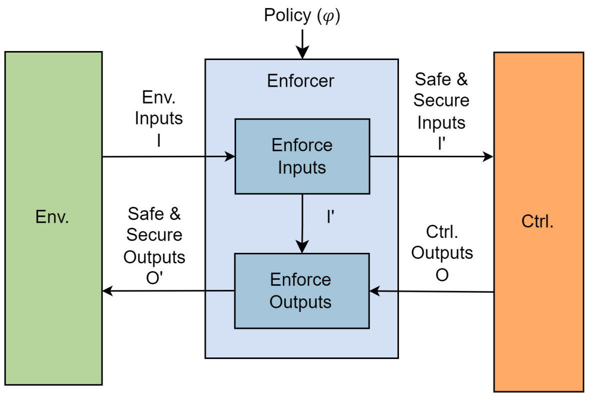


**FIGURE 2. System view of bidirectional reactive enforcement.**

The number of security policies increases as CPSs and the threat landscape become more complex. Therefore, the need to enforce multiple policies simultaneously has emerged. To simultaneously enforce multiple policies, there exist three methods of composition: monolithic, serial, and parallel.

The monolithic approach consists of taking the product of individual policies to produce a single large policy. This is then synthesised into a single enforcer. This approach has been shown to scale poorly due to state space explosion [20]. The serial approach, introduced for bidirectional synchronous reactive systems in [20], synthesises multiple enforcers that are executed sequentially. This overcomes the scalability issues of the monolithic approach, but is implemented in software, which assumes the firmware and software stack to be safe and secure. The myriad of security vulnerabilities and successful compromises of this stack challenge this assumption. This suggests enforcers executing on software platforms are at risk of being compromised by malicious actors exploting software vulnerabilities. This motivates our work to provide a high-trust method for compositional enforcement in bidirectional synchronous systems, as illustrated in Figure 2. We develop a generic framework that supports parallel composition of enforcers in hardware and can be applied to a range of CPSs. To demonstrate this framework, we consider a range of attacks on a FFF 3D printer and develop a range of policies, that we synthesise into enforcers, to defend against the attacks. These enforcers are synthesised to be executed simultaneously and then, through a merge block, a final set of signals, that satisfies all policies, is emitted.

*Contributions.* The contributions of this work are as follows:

- We propose a novel compositional framework for bidirectional RE that supports multiple policies with the parallel approach.
- We provide a tool which compiles multiple policies to a hardware description language for parallel compositions. This is an extension to the compiler *easy-rte-incremental* [20] which we call *easy-rte-hardware*.
- We consider an FFF 3D printer as our CPS case study for which we develop a set of attacks that may be carried out by a malicious actor. To mitigate these attacks we develop a set of defense policies. These policies are synthesised into hardware runtime enforcers with monolithic, serial, and parallel compositions.
- Our evaluation and analysis compares monolithic, serial, and parallel compositions of non-functional metrics as the number of policies increases. The results demonstrate exponential growth in compile time, compiled code size, synthesised logic elements, and synthesised registers for the monolithic approach. The serial and parallel approaches compare favourably with linear growth for these metrics.

*Outline.* In Section II, we introduce and discuss related work. In Section III, we introduce the preliminaries and notations for RE of synchronous systems. We recall, from existing work, the RE framework for synchronous programs in Section IV. In Section V, we discuss compositional approaches to enforcement, recalling, where appropriate, existing work in monolithic and serial composition before discussing limitations. This motivates our high-trust parallel composition, which we introduce in Section VI. In Section VII, we present the security of 3D printers, with a set of potential attacks and policies that mitigate them. We then explain the synthesis of hardware enforcers using monolithic, serial, and our proposed parallel framework in Section VIII. Results of this implementation are presented Section IX, where we compare monolithic, serial, and parallel composition of enforcers in hardware. In Section X, we discuss challenges, trade offs, and applications of the presented work. Finally, conclusions are drawn in Section XI.

## II. RELATED WORK

The generation of enforcers from properties is an existing field of research with varying approaches and applications. Schneider [15] proposes enforcers which delay execution with buffering when a sequence of events does not satisfy the security automata. Edit automata [16] allow enforcers to alter the input sequence by suppressing and/or inserting events. The methods in [18] enable the buffering of events, which are released once the sequence satisfies the required policy. These approaches consider only a single direction of communication, often from the controller to the environment, which limits properties to enforce only outputs.

Bi-directional runtime enforcement was added in Mandatory Result Automata (MRAs) [21] which can reason over communication between two parties. Pearce et al. [22] proposed bidirectional runtime enforcement for various cyber-physical threats in industrial applications with timed policies and, in previous work, we modelled bidirectional jamming, injection, and edit attacks to create enforcers that attack a simulated drone system.

In the security domain there are a number of scenarios where policies need to be composed. A prevalent example is firewall policies. There are many simultaneously active policies, such as allowing the flow of traffic between company IP addresses and port ranges, and blocking access to untrustworthy domains and IP addresses. Often these are supported informally by firewalls or device level security applications. Examples are the Fang [23] and Firmato tools [24] that allow the composition of security policies for firewall management. Other applications include developing, updating, and visualising access control policies, a subset of runtime enforceable policies, in [25]. Policies are not just limited to access and firewalls. In [26], a tool *polymer* is proposed for enforcing composable policies in java applications. Such tools and approaches are common but often informal, meaning they may not gurantee correctness.

In [27], Pinisetty and Tripakis investigated the compositionality of enforcers for a unidirectional RE architecture that allows the enforcer to buffer events (this is equivalent to delaying events). This work explored the synthesis of multiple enforcers, one for each policy, and then if composing these in series or parallel could satisfy all policies. However, buffering of events is not possible in reactive systems.

For reactive systems, the enforcer must react instantly, and so these approaches which allow combinations of halting or delaying are not adequate. However, enforcement frameworks developed in works such as [19], [20], and [28] are relevant.

Unidirectional reactive enforcers, termed *shields*, are introduced in [28], where safety properties are synthesised into *shields* which observe environment input and controller outputs. The *shields* then transform outputs as little as possible to ensure correctness by the safety properties.

Our work is based on [19], which introduces a framework for bidirectional RE, but does not consider composition of multiple policies, and [20], which introduces a framework for incrementally composing policies but considers only serial composition in software.

No existing RE framework considers parallel composition for reactive security enforcers in hardware. The contributions of this work address the risk of security vulnerabilities in software platforms, which undermine enforcer integrity, and simultaneously supports composing multiple policies for increasingly complex reactive systems.

### A. RELATIONSHIP TO PHYSICAL LAYER FAULT TOLERANCE

The hardware runtime enforcement we propose is similar in spirit to other approaches that use hardware to ensure a minimum quality of service such as physical layer fault tolerance.

Physical layer fault tolerance can be improved with a variety of approaches that use the principles of redundancy, error detetion, and correction. In [29] and [30], fault tolerance is considered for multi-agent distributed systems where concensus between agents must be reached without continious communication. Our work differs as a single agent (enforcer) is responsible satisfying multiple policies. This requires exammining bidirectional communication between the system plant and controller, rather than communication between agents in a multi-agent system.

However, there remains a largely unexplored area of runtime enforcement, called distributed enforcement, where policies can be applied to and between agents. Elements of control theory from multi-agent systems, like those in [29] and [30], may be leveraged for distributed enforcement, though this is beyond the scope of this work.

## III. PRELIMINARIES

In this section, we introduce the notations and the safety automata formalism used to define security 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 $\Sigma$ is a finite sequence $\sigma = a_1 \cdot a_2 \cdots a_n$ of members of $\Sigma$, and $\Sigma^*$ denotes the set of finite words over $\Sigma$. Considering a finite word $\sigma$, its length is denoted as $|\sigma|$. $\epsilon_\Sigma$ is used to denote the empty word over $\Sigma$ is denoted by $\epsilon_\Sigma$, or $\epsilon$ (when the context makes it evident). Given two words $\sigma$ and $\sigma'$, their *concatenation* is indicated as $\sigma \cdot \sigma'$. A word $\sigma'$ is a *prefix* of a word $\sigma$, represented as $\sigma' \preccurlyeq \sigma$, whenever a word $\sigma''$ is present such that $\sigma = \sigma' \cdot \sigma''$; $\sigma$ is called an *extension* of $\sigma'$.

A reactive system with 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 $\sigma$ 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 $\sigma$ 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 $\varphi$ (over $\Sigma$) 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 $\varphi$ 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. Security policies are formalized as safety automata, which we define next in this section.

Synchronous programming languages [31] are ideal for developing synchronous reactive systems. They express safety properties via observers [32], 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 $Q \setminus \{q_v\}$ exist. 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 $q_0 \xrightarrow{\sigma} q$.*

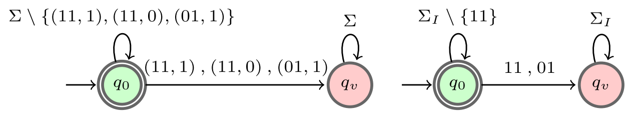
An SA $\mathcal{A} = (Q, q_0, q_v, \Sigma, \rightarrow)$ is *deterministic* if $\forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \implies (q' = q'')$. $\mathcal{A}$ is *complete* if $\forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q'$. A word $\sigma$ 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, $\varphi$ 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 $\varphi$ 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 $\varphi_I \subseteq \Sigma_I^*$.*

*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.*



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

**FIGURE 3. Safety Automaton (SA) (left) and its input SA (right).**[1]

*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 $\mathsf{editI}_{\varphi_I}$ (resp. $\mathsf{editO}_\varphi$), for editing input (resp. output) events (when required), as per the policy $\varphi_I$ (resp. $\varphi$).

- **$\mathsf{editI}_{\varphi_I}(\sigma_I)$**: Given $\sigma_I \in \Sigma_I^*$, $\mathsf{editI}_{\varphi_I}(\sigma_I)$ is the set of input events $x \in \Sigma_I$ s.t. the word obtained by concatenating $x$ after $\sigma_I$ satisfies policy $\varphi_I$. Formally,
$$\mathsf{editI}_{\varphi_I}(\sigma_I) = \{x \in \Sigma_I : \sigma_I \cdot x \models \varphi_I\}.$$
When we consider the SA $\mathcal{A}_{\varphi_I} = (Q, q_0, q_v, \Sigma_I, \rightarrow_I)$, the members in $\Sigma_I$ that allow to reach a state in $Q \setminus \{q_v\}$ from a state $q \in Q \setminus \{q_v\}$ is defined as:
$$\mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q) = \{x \in \Sigma_I : q \xrightarrow{x}_I q' \wedge q' \neq q_v\}.$$
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, $\mathsf{editI}_{\varphi_I}(\sigma_I) = \Sigma_I \setminus \{11\}$. Moreover, $q_0 \xrightarrow{10 \cdot 01}_I q_0$, and $\mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q_0) = \Sigma_I \setminus \{11\}$.
- **$\mathsf{randEditI}_{\mathcal{A}_{\varphi_I}}(q)$** If $\mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q)$ is non-empty, then $\mathsf{randEditI}_{\mathcal{A}_{\varphi_I}}(q)$ returns an element (chosen randomly) from $\mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q)$ and is undefined if $\mathsf{editI}_{\mathcal{A}_{\varphi_I}}(q)$ 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 $\Sigma_O$ s.t. the input-output word obtained by concatenating $\sigma$ followed by $(x,y)$ (i.e., $\sigma \cdot (x,y)$) satisfies policy $\varphi$. 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 $\varphi$, and an input event $x \in \Sigma_I$, the set of output events $y$ in $\Sigma_O$ 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 3. We have $\mathsf{editO}_{\mathcal{A}_\varphi}(q_0, 01) = \{0\}$.
- **$\mathsf{randEditO}_{\mathcal{A}_\varphi}(q, x)$** If $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is not empty, then $\mathsf{randEditO}_{\mathcal{A}_\varphi}(q, x)$ returns a random element from $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$, and if $\mathsf{editO}_{\mathcal{A}_\varphi}(q, x)$ is empty $\mathsf{randEditO}_{\mathcal{A}_\varphi}(q, x)$ is undefined.

## B. SELECT FUNCTIONS

In this section, we recall the $\mathsf{Select}$ functions, the $\mathsf{minD}$ function, and the incremental enforcement function. The incremental security enforcement schemes that we shall be discussing are defined using these $\mathsf{Select}$ 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 $\sigma_I$ with $x$ satisfies policy $\varphi_I$. 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

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

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 3b. 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, $q_0 \xrightarrow{00\cdot 01}_I q_0$, 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 $\sigma$ with $(x, y)$ satisfies policy $\varphi$. 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 $\varphi$, 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 $\varphi$, 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 $y$).

## IV. RUNTIME ENFORCEMENT WITH SAFETY AUTOMATA

To set the scene, we recall the runtime enforcement approach from [19] which presents how any given word $\sigma \in \Sigma^*$ is transformed to comply with the policy $\varphi$.

An enforcer for the policy $\varphi$ can only alter an input-output event when it's absolutely essential; it can't block, postpone, or suppress events.

An enforcer may be thought of as a function that modifies input-output words at a high level. An enforcement function for the policy $\varphi$ takes an input-output word over $\Sigma$ as input and produces an input-output word over $\Sigma$ that conforms to $\varphi$ as output.

We briefly recall the constraints that an enforcer for any given policy $\varphi$ should satisfy. Formal definitions of these constraints and more details are given in [19].

Constraints that should be satisfied by an enforcer for a given property $\varphi$:

Several constraints, such as Soundness, Transparency, Monotonicity, Instantainety, and Causality, must be met by an enforcer for a given property $\varphi$.

To be considered sound, the enforcer's output for each input word must always satisfy the property $\varphi$. In order to maintain transparency, the enforcer leaves the input event undisturbed when no changes are needed to comply with policy $\varphi$. The monotonicity condition means that the enforcer cannot undo what has already been transmitted as output. The instantainety constraint states that when the enforcer receives a new event, it must respond immediately and provide an output event instantaneously. The causality constraint specifies that, the enforcer has to first transform inputs from the environment in each step according to property $\varphi$, followed by reading and transforming the output.

*Remark 2 (Enforceability): Let $\varphi \subseteq \Sigma^*$ be a policy. We recall from [20] that $\varphi$ is* enforceable *iff an enforcer $E_\varphi$, for $\varphi$, satisfying all the constraints such as Soundness, Transparency, Monotonicity, Instantainety, and Causality exists. As discussed in [20], not all safety properties are enforceable. The conditions for enforceability (defining when a given policy is said to be enforceable) are also discussed in [20]. Informally, we can understand the SA defining the policy to satisfy the enforceability condition when every accepting state in the SA has one (or more) transition(s) to an accepting state.*

We now recall the definition of an enforcement function from [19] that satisfies the above discussed constraints. Every reaction of the system is an input-output event pair $(x, y)$, where $x \in \Sigma_I$ is the input, and $y \in \Sigma_O$ is the output. When an enforcer receives an input-output pair $(x, y)$ it immediately produces the transformed input-output pair $(x', y')$. The enforcer first processes the input $x$ to produce $x'$, and then the output $y$, to produce $y'$ and form the pair $(x', y')$. The enforcement function $E_\varphi$ consists of two subfunctions: input enforcement function $E_I$ and output enforcement function $E_O$. $E_I$ requires the environment input $x$ to produce transformed input $x'$. $E_O$ requires the transformed input $x'$ and the controller output $y$ (obtained by running the controller) to produce the transformed event pair $(x', y')$ which is appended to the output of the enforcer.

*Definition 4 (**Enforcement Function**): The enforcement function $E_\varphi : \Sigma^* \rightarrow \Sigma^*$ for a 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 $x' = \mathsf{randEditI}_{\varphi_I}(E_I(\sigma_I))$.*

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

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

*where* $y' = \mathsf{randEditO}_{\varphi}(E_O(\sigma_I, \sigma_O), x)$.

The function $E_\varphi$ takes a word over $\Sigma^*$ and outputs another word over $\Sigma^*$. For a word $\sigma \in \Sigma^*$ the projection of $\sigma$ on inputs is $\sigma_I \in \Sigma_I^*$, and the projection of $\sigma$ on outputs is $\sigma_O \in \Sigma_O^*$. The output of function $E_\varphi$ is defined through two functions, $E_I$ and $E_O$.

*Function $E_I$:* The input enforcement function $E_I$ takes, as input, a word projected on inputs $\sigma_I \in \Sigma_I^*$ and returns a word in $\Sigma_I^*$ 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 $\Sigma_I$ is read as input and $E_I(\sigma_I)$ is returned as output, there are two possible scenarios:

- If $E_I(\sigma_I)$ followed by the input $x$ satisfies input policy $\varphi_I$ then the input $x$ is concatenated to the previous output of function $E_I$: $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x$.
- Alternatively, $E_I(\sigma_I) \cdot x$ does not satisfy $\varphi_I$ and so input $x$ is transformed to input $x'$ using $\mathsf{randEditI}_{\varphi_I}(E_I(\sigma_I))$, which is appended to the previous output of function $E_I$: $E_I(\sigma_I \cdot x) = E_I(\sigma_I) \cdot x'$. Note that $\mathsf{randEditI}_{\varphi_I}(E_I(\sigma_I))$ returns $x' \in \Sigma_I$, such that $\varphi_I$ is satisfied by the $E_I(\sigma_I)$ followed by $x'$.

*Function $E_O$:* The output enforcement function $E_O$ takes, as input, an input word from $\Sigma_I^*$ and an output word from $\Sigma_O*$ and returns an input-output word in $\Sigma^*$, 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 $\epsilon$ 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 $E_O(\sigma_I, \sigma_O)$. If another input event $x$ and output event $y$ are observed, there are two possibilities:

- If $E_O(\sigma_I, \sigma_O)$ followed by $(x, y)$ satisfies $\varphi$, then $(x, y)$ is added to the previous output of $E_O$: $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y)$.
- Alternatively, $E_O(\sigma_I, \sigma_O) \cdot (x, y)$ does not satisfy $\varphi$. In this case, $\mathsf{randEditO}_{\varphi}(E_O(\sigma_I, \sigma_O), x)$ alters output $y$ to obtain $y'$, and the event $(x, y')$ is added to the previous output of the function $E_O$: $E_O(\sigma_I \cdot x, \sigma_O \cdot y) = E_O(\sigma_I, \sigma_O) \cdot (x, y')$. Note that $\mathsf{randEditO}_{\varphi}(E_O(\sigma_I, \sigma_O), x)$ outputs $y' \in \Sigma_O$ such that $\varphi$ is satisfied by $E_O(\sigma_I, \sigma_O)$ followed by $(x, y')$.

*Remark 3 (Functional definition satisfies constraints): In [19], it is proved that for any given policy $\varphi$ that is enforceable, the enforcer defined as function $E_\varphi$ (Definition 4) satisfies the Soundness, Transparency, Monotonicity, Instantainety, and Causality constraints.*

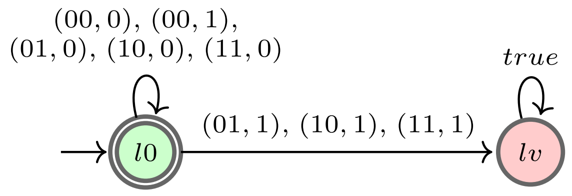


**FIGURE 4.** Example policy ***P***.

*Example 2 (Functional definition): For example, consider the policy P: "*MAX_CURRENT *and* HEATER *can't happen at the same time, and* MAX_TEMP *and* HEATER *can't happen at the same time" illustrated in Figure 4, where $I$ =* {MAX_CURRENT, MAX_TEMP} *and $O$ =* {HEATER}. *The output of functions $E_I$ and $E_O$ is illustrated in Table 1. The complete the input sequence $\sigma = (01, 0) \cdot (01, 1)$ (where $\sigma_I = 01 \cdot 01$ and $\sigma_O = 0 \cdot 1$) is processed gradually by the enforcer function. Initially, the input and output words are empty, $\epsilon_I$ and $\epsilon_O$ respectively, and so enforcer output is empty, $\epsilon$. The first event, when $\sigma$ is (01,0) (i.e.* MAX_TEMP*), it satisfies policy P, so is emitted without any edit. The second event (01,1) (i.e.* MAX_TEMP *and* HEATER*) ($\sigma = (01, 0) \cdot (01, 1)$) is acceptable to the input enforcer and so $E_I(\sigma_I)$ = 01 · 01. However, the* HEATER *output of 1 violates P, and so the enforcer transforms the* HEATER *signal to 0 to satisfy the policy. Thus, the enforcer outputs* $(01, 0) \cdot (01, 0)$.

**TABLE 1.** Functional definition example for policy ***P***.

| $\sigma_I$ | $\sigma_O$ | $E_I(\sigma_I)$ | $E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$ |
|---|---|---|---|
| $\epsilon_I$ | $\epsilon_O$ | $\epsilon_I$ | $(\epsilon_I, \epsilon_O) = \epsilon$ |
| 01 | 0 | 01 | $(01, 0)$ |
| $01 \cdot 01$ | $0 \cdot 1$ | $01 \cdot 01$ | $(01, 0) \cdot (01, \mathbf{0})$ |

## V. RUNTIME ENFORCEMENT WITH MULTIPLE POLICIES

In this section, we discuss monolithic, incremental, and parallel composition methods for enforcing a set of policies expressed as Safety Automaton (SA) in the 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 5a and Figure 5b define policies $S_1$ and $S_2$ respectively.*

### A. MONOLITHIC SECURITY ENFORCEMENT

The monolithic approach to the composition of a collection of (SA) policies takes the product of policies. We take the product of SA as defined in [20] as the intersection of policies. We can then synthesise one enforcer for the resulting policy (product of policies) if this resulting policy is enforceable (as discussed in Remark 2).

Specifically, given any two policies $\varphi_1$ and $\varphi_2$, to enforce both these policies, we first compute $\varphi = \varphi_1 \cap \varphi_2$ (by computing the product of SA for $\varphi_1$ and $\varphi_2$). Then if the resulting SA for $\varphi$ is enforceable, we synthesize an enforcer for $\varphi$ using the approach described in Section IV.

*Example 4 (Monolithic Approach): Consider policies $S_1$ and $S_2$ defined as SA illustrated in Figure 5. The product of these automata, defining $S_1 \cap S_2$, is shown in Figure 6. The policy $S_1 \cap S_2$ is enforceable as every accepting state has one (or more) transition(s) to an accepting state (See Remark 2). The behaviour of an enforcer for $\mathcal{A}_{S_1 \cap S_2}$ is illustrated in Table 2 when the input-output word* $(100, 1)\cdot(110, 1)\cdot(011, 0)$ *is processed incrementally.*

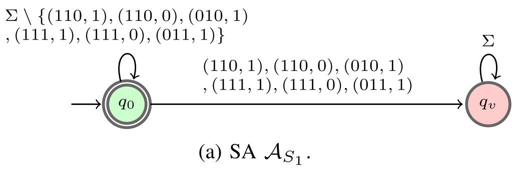

$\Sigma \setminus \{(110,1),(110,0),(010,1),(111,1),(111,0),(011,1)\}$
$(110,1),(110,0),(010,1),(111,1),(111,0),(011,1)$
$\Sigma$
$q_0$
$q_v$
(a) SA $\mathcal{A}_{S_1}$.
$\Sigma \setminus \{(011,0),(011,1),(111,0),(111,1)\}$
$(011,0),(011,1),(111,0),(111,1)$
$\Sigma$
$q_0$
$q_v$
(b) SA $\mathcal{A}_{S_2}$.

**FIGURE 5.** Safety automaton for $S_1$ and $S_2$.

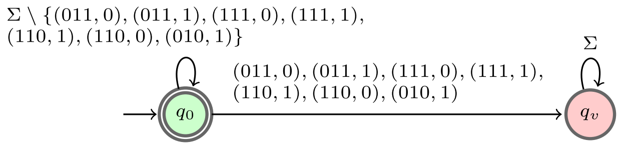

$\Sigma \setminus \{(011,0),(011,1),(111,0),(111,1),(110,1),(110,0),(010,1)\}$
$(011,0),(011,1),(111,0),(111,1),(110,1),(110,0),(010,1)$
$\Sigma$
$q_0$
$q_v$

**FIGURE 6.** $\mathcal{A}_{S_1 \cap S_2}$: Product of automaton $S_1$ and $S_2$.

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

| $\sigma_I$ | $\sigma_O$ | $E_I(\sigma_I)$ | $E_\varphi(\sigma) = E_O(E_I(\sigma_I), \sigma_O)$ |
|---|---|---|---|
| $\epsilon_I$ | $\epsilon_O$ | $\epsilon_I$ | $(\epsilon_I, \epsilon_O) = \epsilon$ |
| 100 | 1 | 100 | $(100, 1)$ |
| $100 \cdot 110$ | $1 \cdot 1$ | $100 \cdot 100$ | $(100,1) \cdot (100,1)$ |
| $100 \cdot 110 \cdot 011$ | $1 \cdot 1 \cdot 0$ | $100 \cdot 100 \cdot 001$ | $(100,1) \cdot (100,1) \cdot (001,0)$ |

As discussed in the problem description, we focus on how to improve trust and scalability of enforcers with hardware composition. In our framework we cannot use the existing Definition 4 to compose enforcers as, in this definition, an enforcer reads from the environment and edits any inputs to satisfy the policy it is defined for. The same occurs for controller output, the enforcer reads this and edits any outputs to satisfy the underlying policy. This does not support multiple policies as an edit made by one enforcer may not be compatible with the other enforcer. In our framework we require all enforcers acceptable solutions to be considered before an edit is selected. In the following section we recall incremental security enforcement [20] which addresses this concern.

## B. INCREMENTAL SECURITY ENFORCEMENT

In earlier work [20], the enforcement layer was incrementally expanded to defend against new threats. This work introduced a framework where each enforcer sequentially takes, as input, a set of possible *acceptable solutions*. This required a redefinition of the enforcement function from Definition 4 and the definition of *Select* functions. The *Select* functions produce subsets of possible *acceptable solutions* which satisfy the policy. This subset is reduced incrementally by each enforcer's *Select* function until a final set, which is acceptable to all policies, is produced. A final function, MinD, is then used to pick the edit action from the final set. This is repeated for both input and output.

*Definition 5 ([Incremental enforcement via serial composition): Given two properties $\varphi_1$ and $\varphi_2$ (where $\varphi_{1I}$ and $\varphi_{2I}$ 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:*

$$\begin{aligned} E_I(\epsilon_{\Sigma_I}) &= \epsilon_{\Sigma_I} \\ E_I(\sigma \cdot a) &= \sigma_I' \cdot MinD(a, \mathsf{SelectI}_{\varphi_2}(\sigma_I', \\ &\qquad (\mathsf{SelectI}_{\varphi_1}(\sigma_I', \Sigma_I)))) \end{aligned}$$

*where $\sigma_I' = E_I(\sigma)$.*

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

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

$$\text{where} \quad \begin{aligned} \sigma' &= E_O(\sigma_I, \sigma_O) \\ y' &= MinD(y, \mathsf{SelectO}_{\varphi_2}(\sigma', x, \\ &\qquad \mathsf{SelectO}_{\varphi_1}(\sigma', x, \Sigma_O))). \end{aligned}$$

As per the incremental composition in Definition 5, all possible inputs $\Sigma_I$ are passed to the input enforcers. The set obtained via SelectI satisfies all input policies, $\varphi_1$ and $\varphi_2$ in this instance.

When a new input event $a$ arrives, it is input to the MinD function along with the output from SelectI. MinD chooses (if required) a suitable element from the set which satisfies all input policies to emit to the controller. If the input $a$ satisfies the policies, it will be in the set from SelectI, and thus will be selected by MinD (this respects the *Transparency* requirement).

Similarly, all possible output events $\Sigma_O$ and the final input event $x$ are input to the output enforcers SelectO. The set obtained from this satisfies all output policies, $\varphi_1$ and $\varphi_2$ in this instance.

When an output event $y$ arrives from the controller, it is input to the MinD function along with the output from SelectO, the set of all possible events which satisfy all policies. MinD chooses (if required) a suitable element from the set which satisfies all output policies to emit to the environment. Similarly to the input enforcement, if $y$ satisfies all policies, it will be the set selected by MinD to respect the *Transparency* requirement.

## C. A NEW ENFORCEMENT APPROACH

We have now considered monolithic and incremental enforcement approaches to composing multiple policies. The problem of composing multiple enforcers is not straight forward. Multiple enforcers as defined in Definition 4 cannot simply be combined, as demonstrated in [20].

The incremental approach resolves this with the definition of *Select* functions that output all *acceptable solutions*.

[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 $\Sigma \setminus (0,0) = \{(0,1),(1,0),(1,1)\}$.

Then a merge function, like Rand (which picks an element randomly from the set), can determine the final output which satisfies all policies.

In our work we consider trust and scalability improvements in hardware composition, and so the incremental approach does not work either, for the following reasons:

- As the enforcers execute in parallel, they do not consider other enforcer's *acceptable solutions* as in the incremental approach. Instead parallel enforcers consider only current events (environment input(s) and/or controller output(s)), the internal automaton location, and any clocks. This prevents parallel enforcers from snooping on output from other enforcers. This supports hidden or confidential policies and their respective enforcers.
- As each enforcer produces a set of *acceptable solutions* simultaneously, these must be combined (via intersection) before an edit can be selected.
- From a practical perspective, passing sets of *acceptable solutions* (i.e. sets of sets) between hardware components does not scale. The initial set of *acceptable solutions* is $2^{(N_I+N_O)}$ where $N_I$ is the number of input events and $N_O$ is the number of output events. As such, there would be at minimum $2^{N_I+N_O}$ connections between each hardware component.

For these reasons, the existing enforcement functions are not suitable for parallel hardware enforcement and so we propose a new high-trust enforcement scheme in the following section.

## VI. HIGH-TRUST HARDWARE ENFORCEMENT

In this section we motivate our hardware implementation of the parallel enforcement strategy in hardware. Like previous efforts proposing runtime enforcement in hardware [22], [33], we also choose this strategy as pure-hardware systems are, by definition, more secure than those relying on software. Software-based systems, as they run atop a hardware platform, must consider software security, hardware security, and cross-domain security [34]. Pure hardware systems, on the other hand, have a far smaller attack surface since they only need to be concerned about the subset of the aspects of hardware security. Software systems, which rises in complexity from 'bare-metal' application code all the way through to multi-threaded and networked operating systems, introduce a wide attack surface depending on the nature of the application in question. Without such a software layer, an entire class of vulnerabilities are no longer applicable to pure hardware systems.

Still, there are risks in hardware applications, including in the supply chain, although we largely consider these out of scope for this work (refer to the surveys [35], [36] for problems and solutions posed in the broader hardware security literature). The major concern in this work regards functional bugs which may be exploitable by malicious adversaries. This is because we target hardware implementations which are immutable and without update functionality. The only risk, if your production facilities are trustworthy, would be in implementations that are faulty because these flaws would not be patchable. Ergo, it is essential that the hardware be implemented *correctly*—the procedures outlined in this section present this formally.

### A. PARALLEL HARDWARE ENFORCEMENT

To define a high-trust framework for multi-policy enforcement in hardware we can repurpose the input and output enforcement Select functions from incremental enforcement function. To compose multiple enforcers in parallel we need to redefine the enforcement function as illustrated in Figure 7:

- composing all input enforcement functions in parallel, such that each accepts input ($x$) and releases all satisfactory input combinations ($X_1$, $X_2$) to a merge block where input that satisfies all policies ($x'$) is selected and released to the controller.
- similarly, composing all output enforcement functions in parallel, such that each accepts final input ($x'$) and controller output ($y$) then releases all satisfactory output combinations ($Y_1$, $Y_2$) to a merge block where output that satisfies all policies ($y'$) is selected and released to the environment.

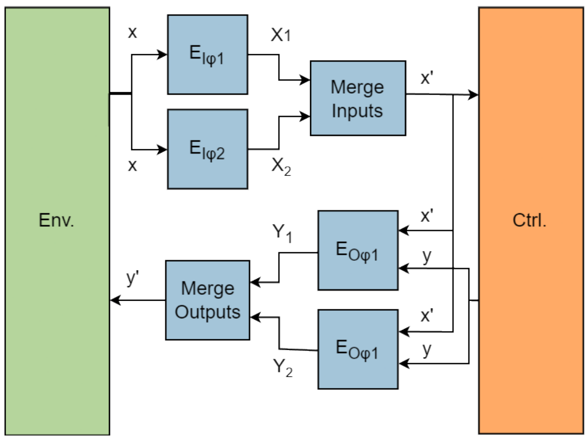


FIGURE 7. Parallel composition.

### B. PARALLEL ENFORCEMENT FUNCTION

We now define the parallel composition of enforcers using Select in Definition 6 below. We use Rand as our merge function to reduce the complexity when implementing the framework compared to MinD. The use of MinD would require a comparison between the original signal and every possible edit, causing an unscalable growth in the complexity of selecting an edit.

*Definition 6 (): Given two properties $\varphi_1$ and $\varphi_2$ (where $\varphi_{1I}$ and $\varphi_{2I}$ are their corresponding input properties), we define the enforcement function $E_{\varphi_1} || E_{\varphi_2} : \Sigma^* \rightarrow \Sigma^*$ as $E_O(E_I(\sigma_I), \sigma_O)$ where:*

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

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

$$E_I(\sigma_I \cdot a) = \begin{cases} \sigma_I' \cdot a & \textit{if } a \in \cap(\\ & \mathsf{SelectI}_{\varphi_1}(\sigma_I', \Sigma_I),\\ & \mathsf{SelectI}_{\varphi_2}(\sigma_I', \Sigma_I)) \\ \sigma_I' \cdot \mathsf{Rand}(\cap( & \textit{otherwise}\\ \mathsf{SelectI}_{\varphi_1}(\sigma_I', \Sigma_I), & \\ \mathsf{SelectI}_{\varphi_2}(\sigma_I', \Sigma_I))) & \end{cases}$$

*where $\sigma_I' = E_I(\sigma_I)$.*

$\cdot$ *$E_O : \Sigma_I^* \times \Sigma_O^* \to (\Sigma_I \times \Sigma_O)^*$ 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 $\sigma' = E_O(\sigma_I, \sigma_O)$,*

$$y' = \begin{cases} y & \textit{if } y \in \cap(\\ & \mathsf{SelectO}_{\varphi_1}(\sigma', x, \Sigma_O),\\ & \mathsf{SelectO}_{\varphi_2}(\sigma', x, \Sigma_O))\\ \mathsf{Rand}(\cap( & \\ \mathsf{SelectO}_{\varphi_1}(\sigma', x, \Sigma_O), & \textit{otherwise}\\ \mathsf{SelectO}_{\varphi_2}(\sigma', x, \Sigma_O))). & \end{cases}$$

$\cdot$ Rand *is a function that picks an element randomly for a set.*

Note the parallel composition of enforcers using *Select* functions always works. Given two properties, $\varphi_1$, $\varphi_2$, and also $\varphi_1 \cap \varphi_2$ are all enforceable, parallel composition of enforcers of $\varphi_1$ and $\varphi_2$ as per the above definition works. The final output obtained does satisfy $\varphi_1 \cap \varphi_2$.

Let us consider input enforcement to understand this (similar reasoning applies to output enforcement). As per parallel composition, defined in Definition 6, all input enforcers simultaneously consider all possible inputs, $\Sigma_I$. The set obtained (using the intersection of individual SelectI() output) is a valid one that satisfies all input properties ($\varphi_1$ and $\varphi_2$ in this case). This set is provided to randEdit which chooses, as required, a suitable edit.

Let us consider the following example to understand this further.

*Example 5 (Parallel Composition using Select): Let us again consider properties $S_1$ and $S_2$ illustrated in Figure 5. Both properties $S_1$ and $S_2$ are individually enforceable and the property $S_1 \cap S_2$ is also enforceable. When we compose input and output enforcers for these properties in parallel as per Definition 6, the final output obtained satisfies property $S_1 \cap S_2$. For example, as shown in Table 3, consider the word $(100, 1) \cdot (110, 1) \cdot (011, 0)$ to be processed. Whenever any input is given, the function* $\mathsf{randEditI}_{\varphi_1}$ *always selects a valid element to input to the output enforcement function always satisfying all the properties.*

*Theorem 2 (Parallel composition using* Select*):* Consider two policies $\varphi_1, \varphi_2$ defined as SA, and where $\varphi = \varphi_1 \cap \varphi_2$. If policy $\varphi$ is enforceable, then $E_{\varphi_1} || E_{\varphi_2}$ as per Definition 6 is an enforcer for $\varphi$ (satisfies the Soundness, Transparency, Monotonicity, Instantainety, and Causality constraints).

**TABLE 3.** Parallel composition scheme using Select.

| $\sigma_I$ | $\sigma_O$ | $X_1 = \mathsf{SelectI}_{\varphi_1}(\Sigma_I)$ | $X_1 = \mathsf{SelectI}_{\varphi_2}(\Sigma_I)$ |
|---|---|---|---|
| $100$ | $1$ | $(100, 101, 010, 001, 011)$ | $(100, 101, 010, 001, 110)$ |
| $100 \cdot 110$ | $1 \cdot 1$ | $(100, 101, 010, 001, 011)$ | $(100, 101, 010, 001, 110)$ |
| $100 \cdot 110 \cdot 011$ | $1 \cdot 1 \cdot 0$ | $(100, 101, 010, 001, 011)$ | $(100, 101, 010, 001, 110)$ |

| $X' = X_1 \cap X_2$ | $\sigma_I' = \mathsf{randEditI}_{\varphi_1}(\sigma_I, X')$ | SelectO() | $\mathsf{randEditI}_{\varphi_1}()$ |
|---|---|---|---|
| $(100, 101, 010, 001)$ | $100$ | $1$ | $(100, 1)$ |
| $(100, 101, 010, 001)$ | $100 \cdot 100$ | $1 \cdot 1$ | $(100, 1) \cdot (100, 1)$ |
| $(100, 101, 010, 001)$ | $100 \cdot 100 \cdot 001$ | $1 \cdot 1 \cdot 0$ | $(100, 1) \cdot (100, 1) \cdot (001, 0)$ |

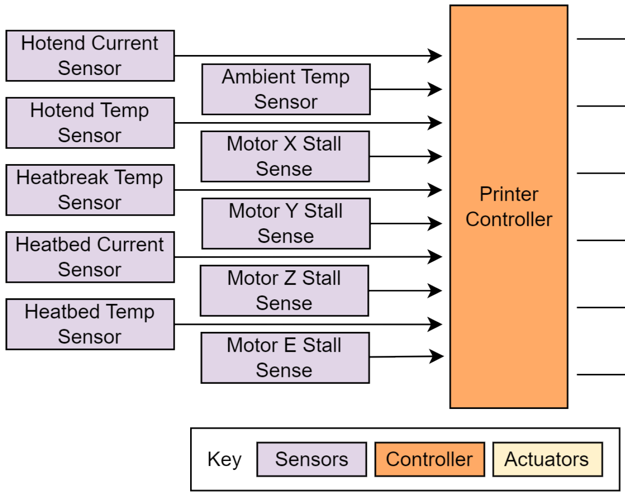
## VII. SECURITY OF 3D PRINTERS

Functionally, FFF 3D printers are a type of CPS. A filament is fed into an extruder and hotend at the filament material's melting temperature (commonly used plastics range between 200°C and 250°C). The hotend nozzle is then moved precisely in 3D space in combination with the extrusion of melted filament to build up, layer by layer, the desired 3D objects.

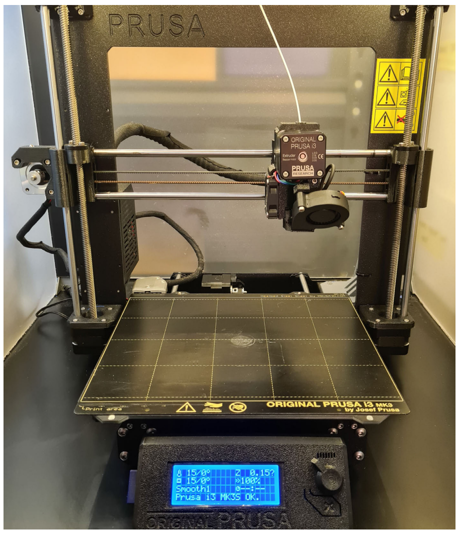
In recent years FFF printing has expanded rapidly. This includes many quickly produced low-cost models, that enable more hobbyists and consumers to begin printing at home, and high-end commercial machines producing parts for application in high-stake scenarios such as tissue and organ printing [37], automotive [38], and aerospace [39].

These systems, like all digital systems, have potential cybersecurity vulnerabilities [40]. Adversaries taking advantage of these vulnerabilities will be motivated by reasons falling into three broad threat categories for 3D printers [41]: (i) *Data Theft* (which contains the subset of *IP Theft*), (ii) *Sabotage* (where some—but not all—of the attack methods will violate the *integrity* of the design file or the manufactured part), and (iii) either unauthorized part manufacturing or reverse engineering of parts, both of which can depend on either theft of digital design files or their reproduction via *reverse engineering* from a physical part. Since hardware parts are manufactured for selling to customers, the reverse engineering and authentication of genuine products is a major challenge.

Attack vectors may fall into purely the digital realm (whereby printer software and/or print files are impacted), the physical realm (where attackers might seek to compromise physical aspects of the print process such as by damaging motors or sensors), or some combination thereof, including compromising the raw printer materials [42]. Further complicating matters, 3D printers are increasingly networked for increased efficiency and monitoring—such features are now commonly native to industry standard designs, and even when not native, may be added via the use of Open Source (OS) tools like OctoPrint [43]. This raises a host of standard cyber-security risks [44], similar to those faced by fast production low security IoT devices [45], [46]. Lateral movement with networks with untrusted devices (such as those fast production low security IoT devices!) compound the risk to a networked 3D printer.



(a) 3D Printer System Diagram



(b) A Prusa i3 MK3S[2] 3D Printer.

**FIGURE 8.** 3D Printer case study.

Demonstrations of AM vulnerabilities include a sabotage attack on 3D printed quadcopter propellers [8], which caused propeller failure during flight, and FLAW3D [47], a Trojan bootloader that caused a reduction in tensile strength by up to 50%. Given the potential for these sabotage-type attacks to compromise safety-critical design features (for instance, imagine an automotive part which has been subtly damaged to fail at high speeds) we focus primarily on how we can defend against sabotage-type attacks.

## A. 3D PRINTER MODEL

As illustrated in Figure 8, we consider a generalised and abstracted 3D Printer[5] with:

- A heated print bed and hotend, requiring two heaters
- Current sensors for each heater
- Temperature sensors for the hotend, heatbreak, heatbed, and ambient temperature.
- Stepper motors for each axis: X, Y, Z, and the extruder (E axis).

The following boolean controller inputs exist:

- MAX_TEMP_HOTEND, MAX_TEMP_HEATBREAK, MAX_TEMP_HEATBED, and MAX_TEMP_AMBIENT represent the maximum operating temperature for the hotend, heatbreak, heatbed, and ambient respectively.
- MAX_CURRENT_HOTEND and MAX_CURRENT_HEATBED represent the maximum current limits for the hotend and heatbed respectively.
- STALL_AXIS_X, STALL_AXIS_Y, STALL_AXIS_Z, and STALL_AXIS_E represent the back EMF based detection of stall[6] in each axis.
- RESET represents a user input reset to the startup state.

The following boolean controller outputs exist:

- EN_HEAT_HOTEND and EN_HEAT_HEATBED signals enable the heaters in the hotend and heatbed respectively.
- EN_MOTOR_X, EN_MOTOR_Y, EN_MOTOR_Z, and EN_MOTOR_E signals enable each axis motor or motors for the X, Y, Z, and E axes respectively.

## B. ATTACKS

We propose a set of actions an attacker may carry out:

- Instruct one or more heater(s) to heat (or operate) above the safe temperature. This could impact the quality of printed items, cause damage to hardware, or result in a fire. Explicitly we consider:
  **A1 Overheat Hotend**: Run the hotend heater above the safe maximum temperature
  **A2 Overheat Heatbreak**: Run the hotend heater when heatbreak is above the safe maximum temperature
  **A3 Overheat Heatbed**: Run the heatbed heater above the safe maximum temperature
  **A4 Overheat Ambient**: Run the hotend heater and/or heatbed heater above the safe maximum ambient temperature
- Operate one or more actuator(s) at or above maximums to exceed safe current limits. While hardware fuses are likely in most devices, replacing these takes time, and repeatedly blowing fuses could be considered as a denial of service attack. In the worst case, a hardware fuse may fail or blow slowly, causing damage to the printer. Explicitly we consider:
  **A5 Overcurrent Hotend**: Run the hotend heater above the safe maximum current
  **A6 Overcurrent Heatbed**: Run the heatbed heater above the safe maximum current

[4]The Prusa i3 MK3S+ website: https://www.prusa3d.com/category/original-prusa-i3-mk3s/

[5]Specifications are based on the Prusa i3 MK4: https://www.prusa3d.com/category/original-prusa-mk4/

[6]Available on stepper motor drivers such as the Trinamic 2130, termed StallGuard2™ [48]

- Drive stepper motors beyond their axis limits. Low impact consequences could be damaged axes, stepper motors, or printer frame, impacting print quality. The higher impact consequence is a fire from the stepper motor(s) overheating. Explicitly we consider:
  - **A7** **Stall X Axis**: Run the X axis stepper motor while stalling for more than one (1) second
  - **A8** **Stall Y Axis**: Run the Y axis stepper motor while stalling for more than one (1) second
  - **A9** **Stall Z Axis**: Run the Z axis stepper motors while stalling for more than one (1) second
  - **A10** **Stall E Axis**: Run the E axis stepper motor while stalling for more than one (1) second

### *C. MITIGATION WITH ENFORCEMENT*

We propose a set of policies described as Discrete Timed Automaton (DTA) to detect and mitigate each previously introduced attack. Policies are defined in three groups (thermal protection, current protection, and stepper motor protection). Policies within groups share structure, but different input and output (I/O). The policies are as follows:

- Thermal Protection
  - $\varphi_1$ The hotend heater should not be enabled when hotend temperature is at maximum (mitigates **A1 Overheat Hotend**)
  - $\varphi_2$ The hotend heater should not be enabled when heatbreak temperature is at maximum (mitigates **A2 Overheat Heatbreak**)
  - $\varphi_3$ The heatbed heater should not be enabled when heatbed temperature is at maximum (mitigates **A3 Overheat Heatbed**)
  - $\varphi_4$ Neither the hotend or heatbed heaters should be enabled when the ambient temperature is at maximum (mitigates **A4 Overheat Ambient**)
- Current Protection
  - $\varphi_5$ The hotend heater should not be enabled when at maximum current threshold (mitigates **A5 Overcurrent Hotend**)
  - $\varphi_6$ The heatbed heater should not be enabled when at maximum current threshold (mitigates **A6 Overcurrent Heatbed**)
- Stepper Motor Protection
  - $\varphi_7$ The X axis motors should not be enabled when stalling for longer than one (1) second (mitigates **A7 Stall X Axis**)
  - $\varphi_8$ The Y axis motors should not be enabled when stalling for longer than one (1) second (mitigates **A8 Stall Y Axis**)
  - $\varphi_9$ The Z axis motors should not be enabled when stalling for longer than one (1) second (mitigates **A9 Stall Z Axis**)
  - $\varphi_{10}$ The E (extruder) axis motor should not be enabled when stalling for longer than one (1) second (mitigates **A10 Stall E Axis**)

#### 1) POLICY $\varphi_1$ MITIGATING ATTACK **A1 OVERHEAT HOTEND**

The attack **A1 Overheat Hotend** runs the hotend (the component responsible for melting the input filament so that it can be extruded through the nozzle) heater by setting EN_HEAT_HOTEND to true when the hotend is at maximum temperature, indicated by the presence of input signal MAX_TEMP_HOTEND. This risks damaging an in-progress print, damaging the hotend, and causing a fire. Policy $\varphi_1$, as illustrated in Figure 9, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

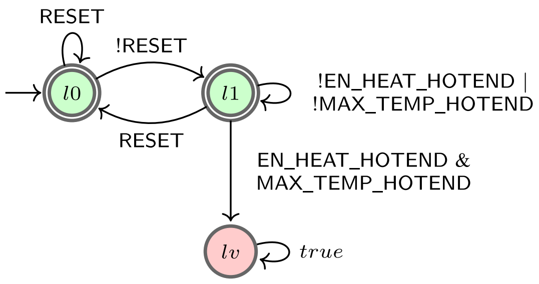


**FIGURE 9.** **Policy $\varphi_1$ which captures the hotend heater should not be enabled when hotend temperature is at maximum.**

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the hotend heater is not enabled (EN_HEAT_HOTEND signal absent) or if the maximum temperature of the hotend is not reached (MAX_TEMP_HOTEND signal absent).

If the hotend heater is enabled (EN_HEAT_HOTEND signal present) while the hotend is at maximum temperature (MAX_TEMP_HOTEND signal present) the policy transitions to a violation. The synthesised enforcer will, therefore, prevent violation by suppressing EN_HEAT_HOTEND, ensuring the policy remains in $l1$.

#### 2) POLICY $\varphi_2$ MITIGATING ATTACK **A2 OVERHEAT HEATBREAK**

The attack **A2 Overheat Heatbreak** runs the hotend heater by setting EN_HEAT_HOTEND to true when the heatbreak (a component, directly attached to the hotend, that is responsible for preventing heat transfer up the filament to prevent blockages) is at maximum temperature, indicated by the presence of input signal MAX_TEMP_HEATBREAK. This risks damaging an in-progress print with inconsistent extrusion, damaging the heatbreak and surrounding components, and causing a fire. Policy $\varphi_2$, as illustrated in Figure 10, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the hotend heater is not enabled (EN_HEAT_HOTEND signal absent) or if the maximum temperature of the heatbreak is not reached (MAX_TEMP_HEATBREAK signal absent).

If the hotend heater is enabled (EN_HEAT_HOTEND signal present) while the heatbreak is at maximum temperature (MAX_TEMP_HEATBREAK signal present)

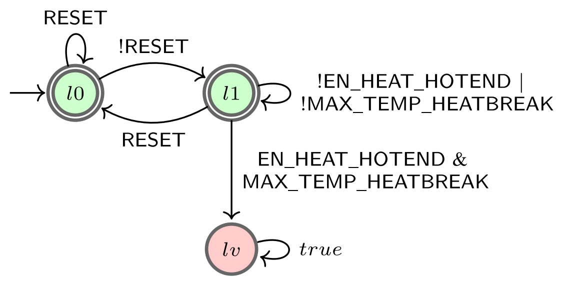


**FIGURE 10.** Policy $\varphi_2$ which captures the hotend heater should not be enabled when heatbreak temperature is at maximum.

the policy transitions to a violation. The synthesised enforcer will, therefore, prevent violation by suppressing EN_HEAT_HOTEND, ensuring the policy remains in $l1$.

### 3) POLICY $\varphi_3$ MITIGATING ATTACK **A3 OVERHEAT HEATBED**

The attack **A2 Overheat Heatbreak** runs the heatbed (base-plate which the 3D prints are extruded onto and gradually built on) heater by setting EN_HEAT_HEATBED to true when the heatbed is at maximum temperature, indicated by the presence of input signal MAX_TEMP_HEATBED. This risks damaging an in-progress print, damaging the heatbed and causing a fire. Policy $\boldsymbol{\varphi}_2$, as illustrated in Figure 11, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

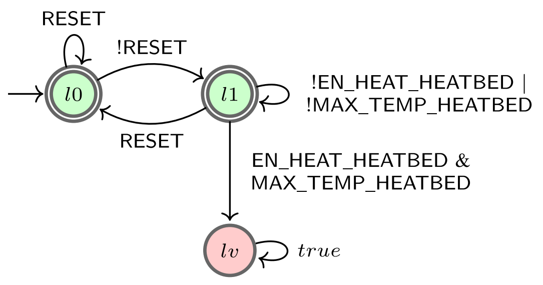


**FIGURE 11.** Policy $\varphi_3$ which captures the heatbed heater should not be enabled when heatbed temperature is at maximum.

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the heatbed heater is not enabled (EN_HEAT_HEATBED signal absent) or if the maximum temperature of the heatbed is not reached (MAX_TEMP_HEATBED signal absent).

If the heatbed heater is enabled (EN_HEAT_HEATBED signal present) while the heatbed is at maximum temperature (MAX_TEMP_HEATBED signal present) the policy transitions to a violation. The synthesised enforcer will, therefore, prevent violation by suppressing EN_HEAT_HEATBED, ensuring the policy remains in $l1$.

### 4) POLICY $\varphi_4$ MITIGATING ATTACK **A4 OVERHEAT AMBIENT**

The attack **A4 Overheat Ambient** runs the hotend heater by setting EN_HEAT_HOTEND to true and/or the heatbed heater by setting EN_HEAT_HEATBED to true when the ambient temperature at or above the printer's maximum operating temperature, indicated by the presence of input signal MAX_TEMP_AMBIENT. This risks reducing the life of printer components if operated beyond safe ambient temperature. Policy $\boldsymbol{\varphi}_4$, as illustrated in Figure 12, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

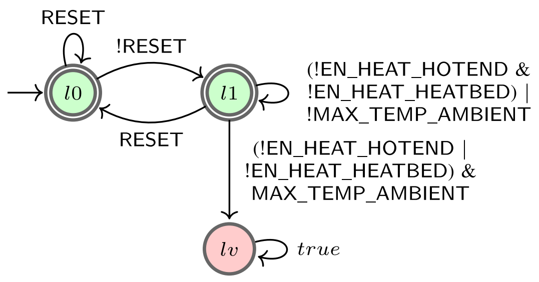


**FIGURE 12.** Policy $\varphi_4$ which captures that neither the hotend or heatbed heaters should be enabled when ambient temperature is at maximum.

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the hotend and heatbed heaters are not enabled (both EN_HEAT_HOTEND and EN_HEAT_HEATBED signals absent) or if the ambient temperature is below maximum (MAX_TEMP_AMBIENT signal absent).

If either or both hotend and heatbed heaters are enabled (EN_HEAT_HOTEND and EN_HEAT_HEATBED signals respectively) while the ambient temperature is maximum (MAX_TEMP_AMBIENT signal present) the policy transitions to a violation. The synthesised enforcer will, therefore, prevent violation by suppressing both EN_HEAT_HOTEND and EN_HEAT_HEATBEDsignals, ensuring the policy remains in $l1$.

### 5) POLICY $\varphi_5$ MITIGATING ATTACK **A5 OVERCURRENT HOTEND**

The attack **A5 Overcurrent Hotend** runs the hotend heater, by setting EN_HEAT_HOTEND to true, when at or above the maximum hotend current, indicated by the presence of input MAX_CURRENT_HOTEND. This risks wire insulation melting which could cause a fire, and reduces the life of the power supply if operated beyond maximum current. Policy $\boldsymbol{\varphi}_5$, as illustrated in Figure 13, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the hotend is not enabled (EN_HEAT_HOTEND signal absent) or if the hotend current is below maximum (MAX_CURRENT_HOTEND signal absent).

If the hotend heater is enabled (EN_HEAT_HOTEND signal present) while the hotend current is at or above the maximum (MAX_CURRENT_HOTEND signal present) the policy transitions to a violation. The synthesised

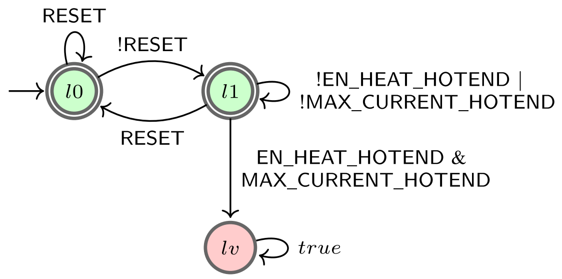


**FIGURE 13. Policy $\varphi_5$ which captures the hotend heater should not be enabled when at maximum current threshold.**

enforcer will, therefore, prevent violation by suppressing the EN_HEAT_HOTEND signal, ensuring the policy remains in $l1$.

### 6) POLICY $\varphi_6$ MITIGATING ATTACK **A6 OVERCURRENT HEATBED**

The attack **A6 Overcurrent Heatbed** runs the heatbed heater, by setting EN_HEAT_HEATBED to true, when at or above the maximum heatbed current, indicated by the presence of input MAX_CURRENT_HEATBED, this risks wire insulation melting which could cause a fire, and reduces the life of the power supply if operated beyond maximum current. Policy $\varphi_6$, as illustrated in Figure 14, mitigates this. The policy consists of two accepting locations ($l0$, $l1$) and one non-accepting violation location ($lv$).

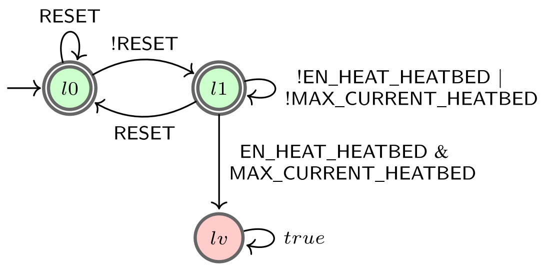


**FIGURE 14. Policy $\varphi_6$ which captures the heatbed heater should not be enabled when at maximum current threshold.**

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ if the hotend is not enabled (EN_HEAT_HEATBED signal absent) or if the hotend current is below maximum (MAX_CURRENT_HEATBED signal absent).

If the heatbed heater is enabled (EN_HEAT_HEATBED signal present) while the heatbed current is at or above the maximum (MAX_CURRENT_HEATBED signal present) the policy transitions to a violation. The synthesised enforcer will, therefore, prevent violation by suppressing the EN_HEAT_HEATBED signal, ensuring the policy remains in $l1$.

### 7) POLICY $\varphi_7$ MITIGATING ATTACK **A7 STALL X AXIS**

The attack **A7 Stall X Axis** continues to run the X axis stepper motor, by setting EN_MOTOR_X to true, when the motor is stalling, indicated by the presence of input STALL_AXIS_X. This risks damaging the stepper motor and the axis frame of the 3D printer. Policy $\varphi_7$, as illustrated in Figure 15, mitigates this. The policy consists of three accepting locations ($l0$, $l1$, $l2$), one non-accepting violation location ($lv$), and has one clock, $V_X$, responsible for timing the duration of a stall.

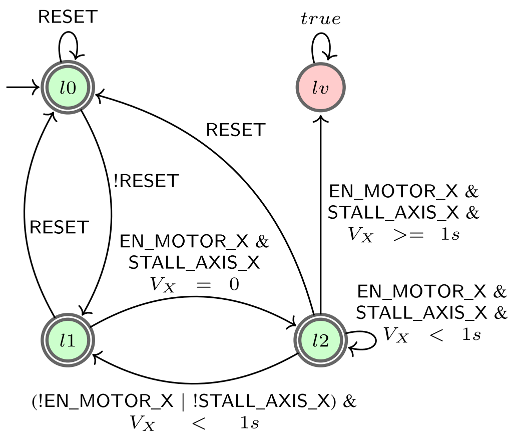


**FIGURE 15. Policy $\varphi_7$ which captures the X axis motors should not be enabled when stalling for longer than one (1) second.**

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ until the stepper motor is enabled and is sensed as stalling (both EN_MOTOR_X and STALL_AXIS_X signals present). If this occurs the policy transitions to $l2$ and the clock $V_X$ is reset.

The following transitions can be taken when the policy is in location $l2$:

- The motor is no longer enabled (EN_MOTOR_X absent) or no longer stalling (STALL_AXIS_X absent) and the clock ($V_X$) is less than one (1) second, then the policy transitions to $l1$.
- The motor is enabled (EN_MOTOR_X present) and stalling (STALL_AXIS_X present) while the clock ($V_X$) is less than one (1) second, then the policy remains in $l2$.
- The motor is enabled (EN_MOTOR_X present) and stalling (STALL_AXIS_X present) and the clock ($V_X$) is greater than or equal to one (1) second, then the policy transitions to $lv$. In this case the synthesised enforcer will prevent violation by suppressing EN_MOTOR_X.

### 8) POLICY $\varphi_8$ MITIGATING ATTACK **A8 STALL Y AXIS**

The attack **A8 Stall Y Axis** continues to run the Y axis stepper motor, by setting EN_MOTOR_Y to true, when the motor is stalling, indicated by the presence of input STALL_AXIS_Y, this risks damaging the stepper motor and the axis frame of the 3D printer. Policy $\varphi_8$, as illustrated in Figure 16, mitigates this. The policy consists of three accepting locations ($l0$, $l1$, $l2$), one non-accepting violation location ($lv$), and has one clock, $V_Y$, responsible for timing the duration of a stall. When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to

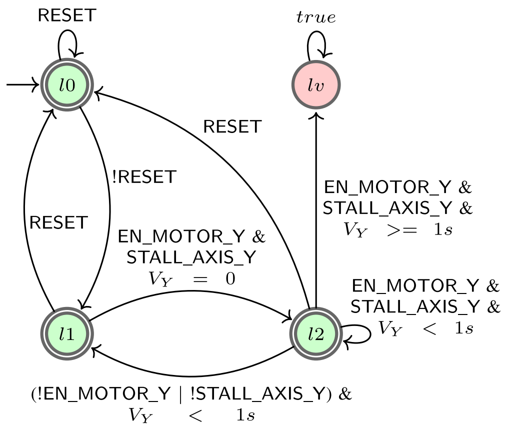

RESET
true
l0
lv
RESET
!RESET
RESET
EN_MOTOR_Y & STALL_AXIS_Y & $V_Y >= 1s$
EN_MOTOR_Y & STALL_AXIS_Y $V_Y = 0$
EN_MOTOR_Y & STALL_AXIS_Y & $V_Y < 1s$
l1
l2
(!EN_MOTOR_Y | !STALL_AXIS_Y) & $V_Y < 1s$

**FIGURE 16.** Policy $\varphi_8$ which captures that the Y axis motors should not be enabled when stalling for longer than one (1) second.

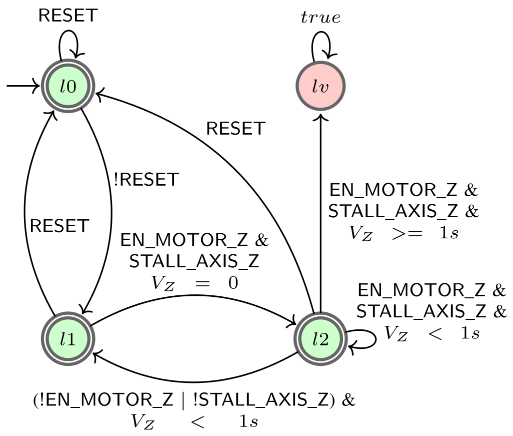

RESET
true
l0
lv
RESET
!RESET
RESET
EN_MOTOR_Z & STALL_AXIS_Z & $V_Z >= 1s$
EN_MOTOR_Z & STALL_AXIS_Z $V_Z = 0$
EN_MOTOR_Z & STALL_AXIS_Z & $V_Z < 1s$
l1
l2
(!EN_MOTOR_Z | !STALL_AXIS_Z) & $V_Z < 1s$

**FIGURE 17.** Policy $\varphi_9$ which captures the Z axis motors should not be enabled when stalling for longer than one (1) second.

$l1$. The policy remains in location $l1$ until the stepper motor is enabled and is sensed as stalling (both EN_MOTOR_Y and STALL_AXIS_Y signals present). If this occurs the policy transitions to $l2$ and the clock $V_Y$ is reset.

The following transitions can be taken when the policy is in location $l2$:

- The motor is no longer enabled (EN_MOTOR_Y absent) or no longer stalling (STALL_AXIS_Y absent) and the clock ($V_Y$) is less than one (1) second, then the policy transitions to $l1$.
- The motor is enabled (EN_MOTOR_Y present) and stalling (STALL_AXIS_Y present) while the clock ($V_Y$) is less than one (1) second, then the policy remains in $l2$.
- The motor is enabled (EN_MOTOR_Y present) and stalling (STALL_AXIS_Y present) and the clock ($V_Y$) is greater than or equal to one (1) second, then the policy transitions to $lv$. In this case the synthesised enforcer will prevent violation by suppressing EN_MOTOR_Y.

### 9) POLICY $\varphi_9$ MITIGATING ATTACK **A9 STALL Z AXIS**

The attack **A9 Stall Z Axis** continues to run the Z axis stepper motor, by setting EN_MOTOR_Z to true, when the motor is stalling, indicated by the presence of input STALL_AXIS_Z, this risks damaging the stepper motor and the axis frame of the 3D printer. Policy $\boldsymbol{\varphi}_9$, as illustrated in Figure 17, mitigates this. The policy consists of three accepting locations ($l0$, $l1$, $l2$), one non-accepting violation location ($lv$), and has one clock, $V_Z$, responsible for timing the duration of a stall. When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ until the stepper motor is enabled and is sensed as stalling (both EN_MOTOR_Z and STALL_AXIS_Z signals present). If this occurs the policy transitions to $l2$ and the clock $V_Z$ is reset.

The following transitions can be taken when the policy is in location $l2$:

- The motor is no longer enabled (EN_MOTOR_Z absent) or no longer stalling (STALL_AXIS_Z absent) and the clock ($V_Z$) is less than one (1) second, then the policy transitions to $l1$.
- The motor is enabled (EN_MOTOR_Z present) and stalling (STALL_AXIS_Z present) while the clock ($V_Z$) is less than one (1) second, then the policy remains in $l2$.
- The motor is enabled (EN_MOTOR_Z present) and stalling (STALL_AXIS_Z present) and the clock ($V_Z$) is greater than or equal to one (1) second, then the policy transitions to $lv$. In this case the synthesised enforcer will prevent violation by suppressing EN_MOTOR_Z.

### 10) POLICY $\varphi_{10}$ MITIGATING ATTACK **A10 STALL E AXIS**

The attack **A10 Stall E Axis** continues to run the E axis stepper motor, by setting EN_MOTOR_E to true, when the motor is stalling, indicated by the presence of input STALL_AXIS_E. This risks clogging the extruder, damaging the stepper motor and the extruder. Policy $\boldsymbol{\varphi}_{10}$, as illustrated in Figure 18, mitigates this. The policy consists of three accepting locations ($l0$, $l1$, $l2$), one non-accepting violation location ($lv$), and has one clock, $V_E$, responsible for timing the duration of a stall.

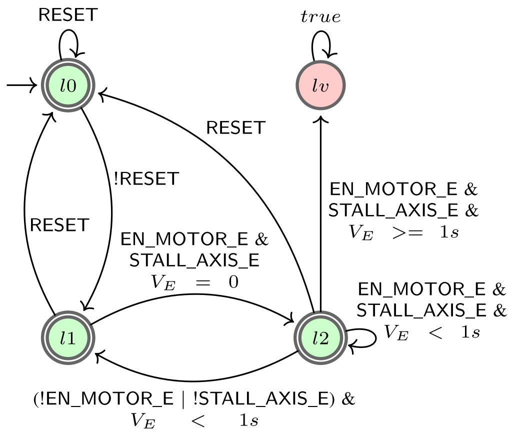

RESET
true
l0
lv
RESET
!RESET
RESET
EN_MOTOR_E & STALL_AXIS_E & $V_E >= 1s$
EN_MOTOR_E & STALL_AXIS_E $V_E = 0$
EN_MOTOR_E & STALL_AXIS_E & $V_E < 1s$
l1
l2
(!EN_MOTOR_E | !STALL_AXIS_E) & $V_E < 1s$

**FIGURE 18.** Policy $\varphi_{10}$ which captures the E (extruder) axis motor should not be enabled when stalling for longer than one (1) second.

When the input reset signal RESET is present, the policy remains in the initial location $l0$, otherwise, the policy transitions to $l1$. The policy remains in location $l1$ until the stepper motor is enabled and is sensed as stalling (both EN_MOTOR_E and STALL_AXIS_E signals present). If this occurs the policy transitions to $l2$ and the clock $V_E$ is reset.

The following transitions can be taken when the policy is in location $l2$:

- The motor is no longer enabled (EN_MOTOR_E absent) or no longer stalling (STALL_AXIS_E absent) and the clock ($V_E$) is less than one (1) second, then the policy transitions to $l1$.
- The motor is enabled (EN_MOTOR_E present) and stalling (STALL_AXIS_E present) while the clock ($V_E$) is less than one (1) second, then the policy remains in $l2$.
- The motor is enabled (EN_MOTOR_E present) and stalling (STALL_AXIS_E present) and the clock ($V_E$) is greater than or equal to one (1) second, then the policy transitions to $lv$. In this case the synthesised enforcer will prevent violation by suppressing EN_MOTOR_E.

## VIII. IMPLEMENTATION

To evaluate the proposed parallel enforcement method we introduce *easy-rte-hardware* an extension of *easy-rte* [22] and *easy-rte-incremental* [20]. The main contribution of this extended compiler is support for policies to be compiled into Verilog hardware components which are run in parallel. Additionally, it adds support for compilation of the monolithic compositions (from *easy-rte*) and incremental compositions (from *easy-rte-incremental*) to synthesisable Verilog. The source code for *easy-rte-hardware* is available online.[7]

### A. COMPILING PARALLEL HARDWARE ENFORCERS

The process for compilation, as illustrated in Figure 19, consists of four steps:

- First, the original *easy-rte* parser creates an intermediate XML file which describes each policy completely.
- Second, our main contribution to the compiler, the *easy-rte-hardware* parser, ingests the intermediate XML and computes the violation recovery table. This table consists of a recovery to each and every combination of violations that the policies have. This is explained further in the following section.
- Third, our modified *easy-rte* templater takes the new XML and produces synthesisable enforcer Verilog.
- Fourth, the Verilog file is passed to Quartus to synthesise hardware.

#### 1) VIOLATION RECOVERY TABLE

The following algorithm is pseudocode to produce the violation recovery table:

[7]The compiler *easy-rte-hardware*, a fork of *easy-rte*, can be accessed at https://github.com/PRETgroup/easy-rte-comp-hw. This includes support for the proposed parallel composition, but also for monolithic and incremental composition in hardware. The source code includes the 3D Printer case study and other examples which can be compiled and run.

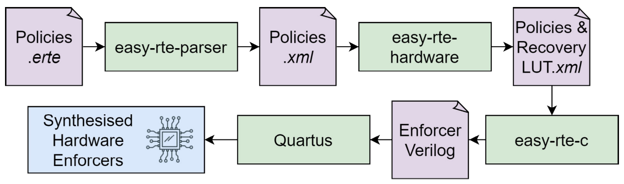


**FIGURE 19. Compiler process flow.**

```
1:  $xml = ParseInput()$
2:  // Get all possible violation combinations
3:  for all $policy in xml.policies$ do
4:      for all $violation, recoveries in policy.Violations$ do
5:          for all $row in violationTable$ do
6:              $row.Append(violation, recoveries)$
7:          end for;
8:      end for;
9:  end for;
10: // Determine solutions to each violation combination
11: for all $row in violationTable$ do
12:     $intersection = GetIntersection(row.recoveries)$
13:     $edit = GetRandEdit(intersection)$
14:     $solutions.Append(row.violations, edit)$
15: end for;
16: // Add the solution table to XML
17: $xmlOut = xml.Append(solutions)$
18: Write $xmlOut$
```

The resulting *xmlOut* is then passed as input to the *easy-rte* templater to produce synthesisable Verliog code. In the next section, we discuss the synthesised hardware.

### B. SYNTHESISED COMPOSITIONAL HARDWARE ENFORCERS

The synthesised enforcers are placed between the sensors and actuators (the environment) and the controller of the 3D printer, as illustrated in Figure 20. As illustrated, the enforcer is able to intercept and edit the controller's inputs and outputs as required.

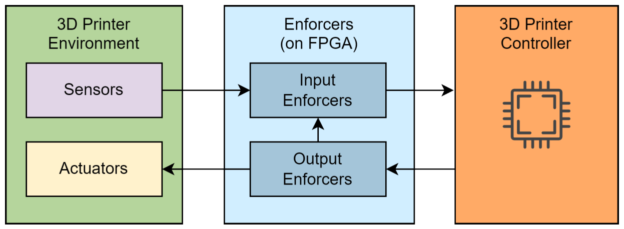


**FIGURE 20. 3D Printer secured with Compositional Enforcers.**

In practice, the enforcers could be deployed with custom hardware either on the printer controller's printed circuit board (PCB) or as a stand alone PCB. Alternatively, if new or altered security policies are expected, a field-programmable gate array (FPGA) could be used to maintain reconfigurability.

### 1) MONOLITHIC

Monolithic compositions result in a single hardware component regardless of the number of policies composed. This component is separated into four subparts (input edit, output edit, transition logic, and storage of state and clocks) as illustrated by Figure 21. The execution of monolithic enforcement follows the four state Finite State Machine (FSM) illustrated in Figure 22.

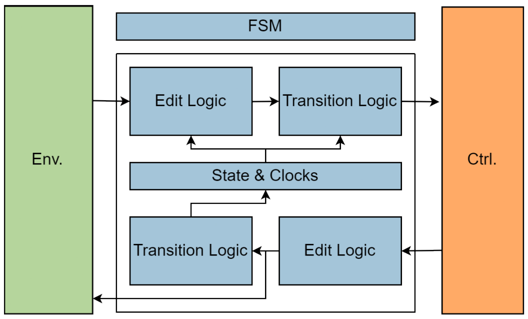


**FIGURE 21.** **Monolithic block diagram.**

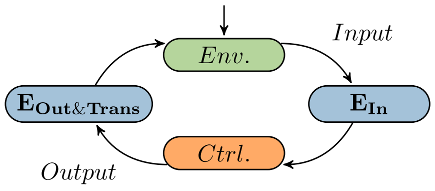


**FIGURE 22.** **Monolithic control FSM.**

Initially, the enforcer waits for the environment (in location *Env.*). When an environment input (*Input*) is produced the FSM transitions to input enforcement (location $\mathsf{E_{In}}$) and the input signals are passed to input edit logic which, if required to prevent a violation, the edits input. This input is then guaranteed to satisfy the input policies and is sent to the controller (FSM transitions to *Ctrl*).

Once the controller generates output (*Output*), the FSM transitions to the output edit logic ($\mathsf{E_{Out\&Trans}}$). Similarly to the input edit logic, any potential violation is edited to ensure the output policies are satisfied. The output is then passed to the transition logic block which determines and executes appropriate location transitions for the monolithic policy. As illustrated in the center of Figure 21, the state and clocks are stored such that each other component can read the current location and value of the clocks. These are updated only by the transition logic block. The output, which is guaranteed to satisfy both input and output policies, is then exposed to the environment as the FSM transitions to *Env.* where the process begins again when the next input event arrives.

### 2) INCREMENTAL

Incremental compositions result in multiple hardware components, each policy adds an input and output enforcement component. Explicitly, there are $2 + 2(N)$ hardware components, where $N$ is the number of policies. The hardware components in an incremental composition are illustrated in Figure 23. The FSM in Figure 24 controls sequential execution, as with the hardware components, the FSM expands as additional policies are added. The number of FSM states is $5 + 2N$, where $N$ is the number of policies.

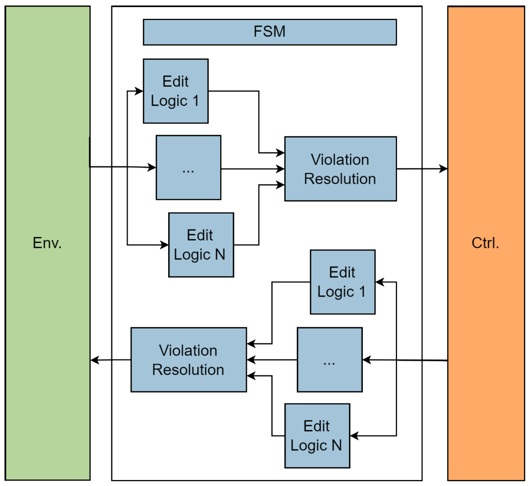


**FIGURE 23.** **Incremental block diagram.**

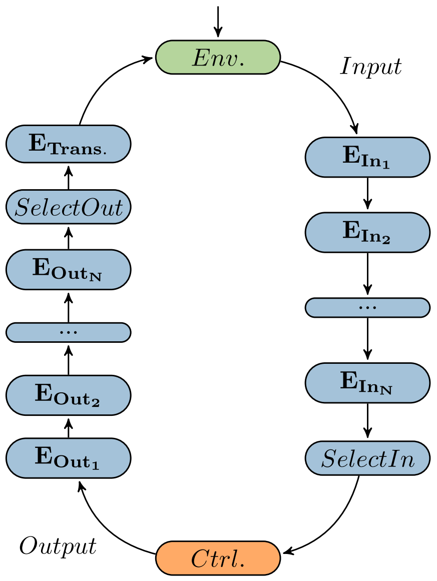


**FIGURE 24.** **Incremental control FSM.**

Regardless of the number of policies, the following pattern of execution occurs. First, the FSM is in location *Env.* waiting for the environment to produce an input event (*Input*) to transition to the first input enforcement location ($\mathsf{E_{In_1}}$) where the first input enforcer edit logic determines which, if any, violation recovery reference to emit (this recovery reference is determined at compile time as described in Section VIII-A1). Sequentially, each input enforcement edit logic is executed as the FSM proceeds through each remaining input enforcement location ($\mathsf{E_{In_2}}$, $\cdots$, $\mathsf{E_{In_N}}$). Then transitioning to input selection (location *SelectIn*) the violation resolution component takes each policy's recovery reference to determine if any edit action is required, and if so, select the appropriate edit. The input then satisfies all input policies and is exposed to the controller as the FSM transitions to *Ctrl*.

When the controller produces an output event (*Output*), similar to the input, each output edit logic component is executed as the FSM transitions through $\mathsf{E}_{\mathsf{Out}_1}$, $\mathsf{E}_{\mathsf{Out}_2}$, $\cdots$, $\mathsf{E}_{\mathsf{Out}_N}$. The violation resolution component then uses each violation recovery reference to determine if any edit action is required and select the appropriate edit. The output then satisfies all output policies and is exposed to the environment as the FSM transitions to *Env*. where the process begins again when the next input event arrives.

### 3) PARALLEL

Parallel compositions result in multiple hardware components, each policy adds an input and output enforcement component. Explicitly, there are $2 + 2(N)$ hardware components, where $N$ is the number of policies. The hardware components in an parallel composition are illustrated in Figure 25. The input enforcement component for each policy is responsible for, given the policy's location, determining and emitting the appropriate violation recovery reference based on the violation recovery table (explained in Section VIII-A1). The output enforcement components are, in addition to determining the appropriate violation recovery references, responsible for policy location transitions, storing the policy location and any clocks.

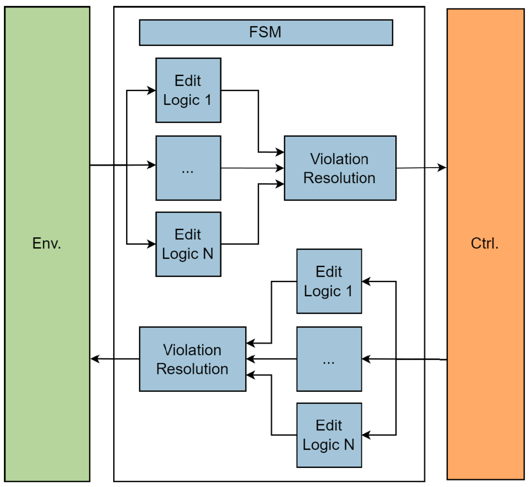


**FIGURE 25.** Parallel block diagram.

The FSM in Figure 26 controls execution, unlike the incremental control FSM, the number of states is fixed at seven. The FSM initially waits in location *Env*. for the environment to produce *Input* at which point the FSM transitions to the input enforcement location ($\mathsf{E}_{\mathsf{In}}$) where each input enforcement component is executed simultaneously to produce all violation recovery references. The FSM transitions to *SelectI* where recovery references are supplied to the violation recovery table which selects, as appropriate,

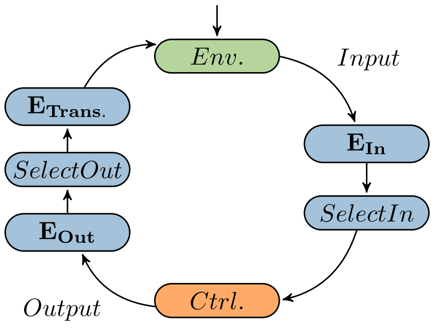


**FIGURE 26.** Parallel control FSM.

to edit the input signals. The input then satisfies all input policies and is exposed to the controller as the FSM transitions to *Ctrl*.

Once the controller produces *Output* the FSM transitions to $\mathsf{E}_{\mathsf{Out}}$ where all output enforcement components are executed simultaneously to produce their violation recovery references. Then transitioning to *SelectOut* the violation recovery table produces an appropriately edited set of input which is used to update the locations of each enforcer policy when the FSM transitions to $\mathsf{E}_{\mathsf{Trans.}}$. Finally, the output, which satisfies all output policies, is exposed to the environment as the FSM transitions to *Env*. where the process begins again when the next input event arrives.

### 4) UPDATING HARDWARE ENFORCERS

When the security landscape changes, policies may be added or altered. This requires recompilation of Verilog and resynthesis of some hardware blocks. The amount of recompilation and resyntehsis varies by method of composition.

Monolithic composition requires complete recompilation and resynthesis of hardware. This is the most time and computationally expensive method

Incremental composition requires compilation and synthesis of the new or altered policies, the FSM controller, and the violation resolution block.

Our proposed parallel approach requires compilation and synthesis of the new or altered policies, and the violation resolution block needs to be recompiled and synthesised. This means the the violation recovery block is resynthesised for any new or altered policies. However, no existing enforcer blocks need to be resynthesised.

## IX. RESULTS

To benchmark the proposed scalable hardware enforcers for the 3D printer we describe each policy from Section VII in *easy-rte* specification language. We use *easy-rte-hardware* to compile monolithic, incremental, and parallel compositions of increasing complexity to Verilog. We do this by adding mitigation policies gradually to each composition as follows:

1) $\varphi_1$
2) $\varphi_1$, and $\varphi_2$
3) $\varphi_1$, $\varphi_2$, and $\varphi_3$
4) $\varphi_1$, $\varphi_2$, $\varphi_3$, and $\varphi_4$

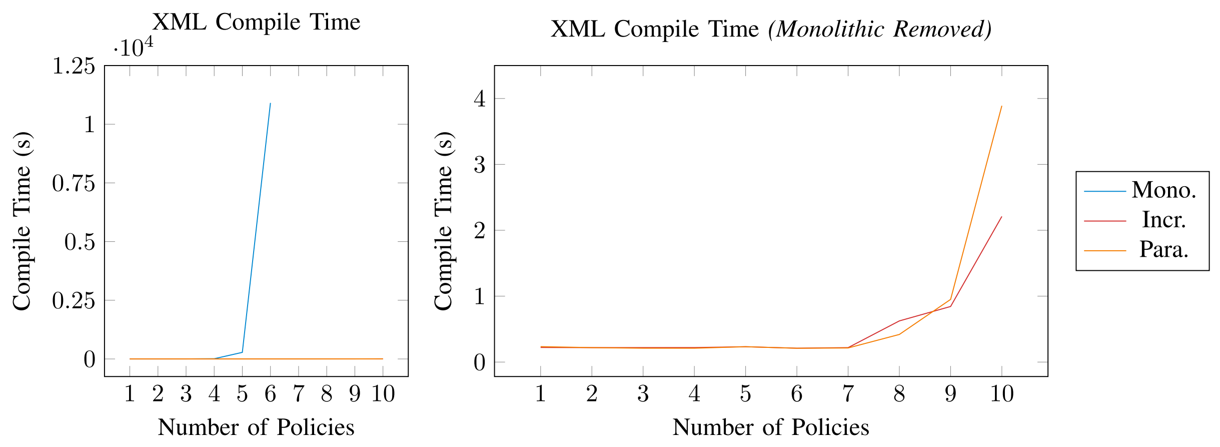


**FIGURE 27.** **Charts illustrating the growth in XML compilation time for Monolithic (Mono.), Incremental (Incr.), and Parallel (Para.). The left chart includes all data points. The right chart excludes the Monolithic results to allow comparison of Incremental and Parallel results. *Note the chart scales differ.***

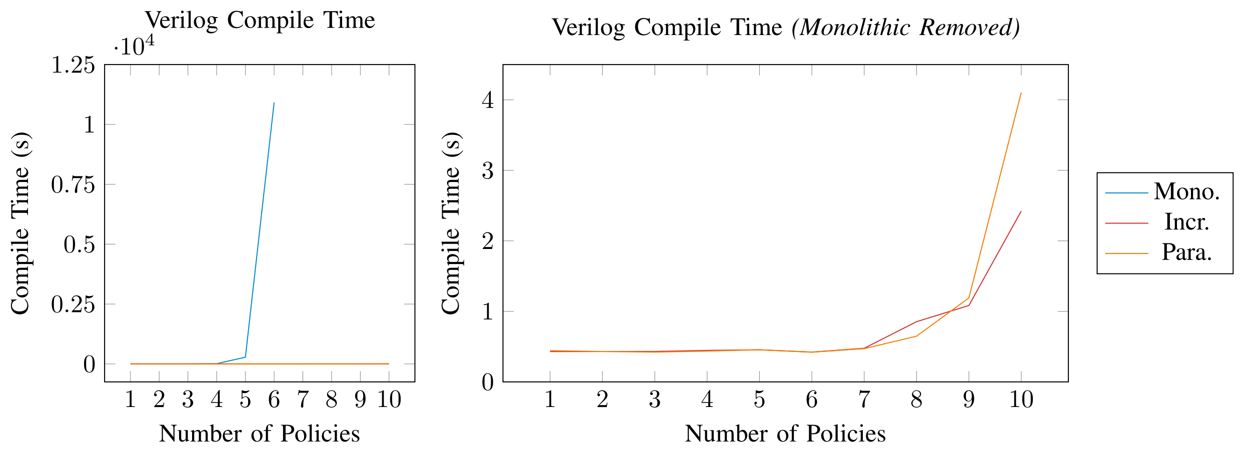


**FIGURE 28.** **Charts illustrating the growth in Verilog SV compilation time for Monolithic (Mono.), Incremental (Incr.), and Parallel (Para.). The left chart includes all data points. The right chart excludes the Monolithic results to allow comparison of Incremental and Parallel results. *Note the chart scales differ.***

5) $\varphi_1, \varphi_2, \varphi_3, \varphi_4$, and $\varphi_5$
6) $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5$, and $\varphi_6$
7) $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6$, and $\varphi_7$
8) $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7$, and $\varphi_8$
9) $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7, \varphi_8$, and $\varphi_9$
10) $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_6, \varphi_7, \varphi_8, \varphi_9$, and $\varphi_{10}$

As the complexity of the composition increases, the number of attacks mitigated increases, thus improving the security of the 3D printer.

We collect performance results for the compilation and hardware resource usage of each composed enforcer. All compilation is performed on a Windows 10 machine with an Intel i7-6700 processor running at 4GHz and with 32GB RAM. For hardware synthesis the command line interface for Quartus Prime Lite 22.1 is used with a Cyclone 5 (5CGXFC7C7F23C8) as the target device.

For compilation, we report on time for *easy-rte* to generate the XML policy description and the Verilog composition, and the time for Quartus to perform synthesis. We also report on the size of XML and Verilog (SV) files. For hardware resources, we report on logic elements, registers, and maximum clock frequency.

Throughout all results the monolithic compositions are completely up to a maximum of six policies. This is due to scalability issues in the *easy-rte* compiler for product automata. Compositions for seven and above policies were terminated after running for longer than two weeks on highly resourced virtual machines on the cloud. Improving *easy-rte* monolithic composition performance is beyond the scope of this work, but remains an area of exploration for future work.

## A. COMPILATION

### 1) COMPILE TIME

The XML compile time is reported in Figure 27. The left chart includes all results and clearly illustrates exponential growth in the monolithic XML compile time, with the six policy composition taking 10,908 seconds (just over 3 hours). The right chart removes monolithic compositions to better visualise the incremental and parallel results. The beginning

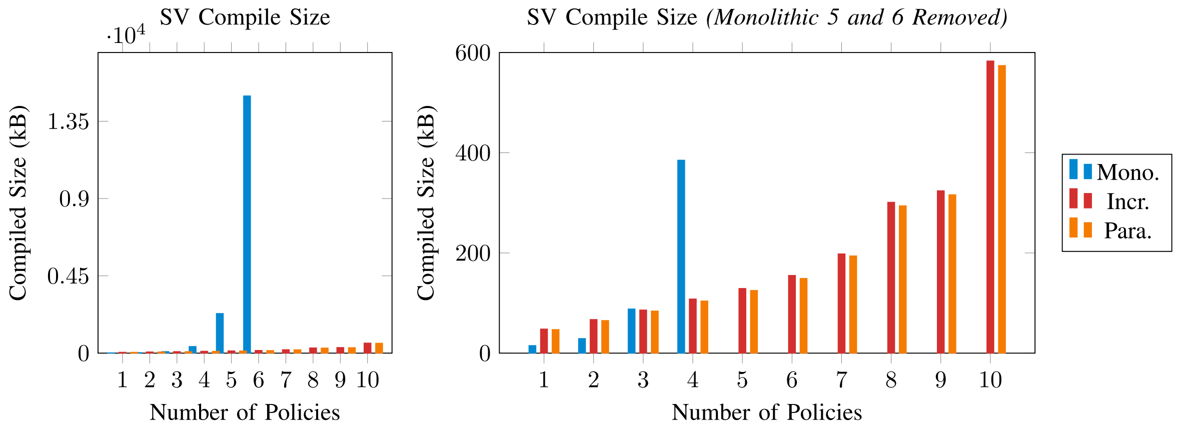


**FIGURE 29.** Charted Verilog SV compilation size for Monolithic (Mono.), Incremental (Incr.), and Parallel (Para.). The left chart includes all data points and the right chart removes the 5th and 6th compositions for Monolithic to improve clarity of Incremental and Parallel results. *Note the chart scales differ.*

of an exponential trend can be seen in compositions eight through ten. The tenth composition took 2.21 seconds for incremental and 3.89 seconds for parallel. Both incremental and parallel demonstrate significantly more scalable XML compile time than monolithic, however the developing exponential trend for parallel and incremental may limit scalability.

The Verilog compile time is reported in Figure 28. The left chart includes all results and, as with XML compile time, illustrates exponential growth in the monolithic compile time. The peak monolithic compile time, for six policies, is similar to XML compile time at 10,915 seconds (just over 3 hours). The right chart removes monolithic compositions to better visualise the incremental and parallel results. Similarly to the XML compile time an exponential trend is starting to be seen for the incremental and parallel compositions eight through ten. The tenth composition took 2.42 seconds for incremental and 4.10 seconds for parallel. Both incremental and parallel demonstrate significantly more scalable XML compile time than monolithic across the ten 3D printer policy compositions, however the developing exponential trend may limit scalability.

The Quartus synthesis time is reported in Figure 30. A clear exponential trend is illustrated by the monolithic compositions, with a peak of 365 seconds for six policies. Parallel and incremental illustrate linear trends with peaks of 152 seconds and 114 seconds respectively. This demonstrates parallel and incremental are significantly more scalable.

Compilation time results demonstrate parallel and incremental methods are more scalable than monolithic. Results from *easy-rte-hardware* show exponential trends for more complex compositions in parallel and incremental. This is likely due to the limited optimisation in the original *easy-rte* for larger policies. An example is memory managment when producing and reading XML files that do not fix into memory. Results from Quartus illustrate linear scale parallel and incremental time complexity which is encouraging, and suggest that with appropriate optimisation *easy-rte* could obtain similar trends.

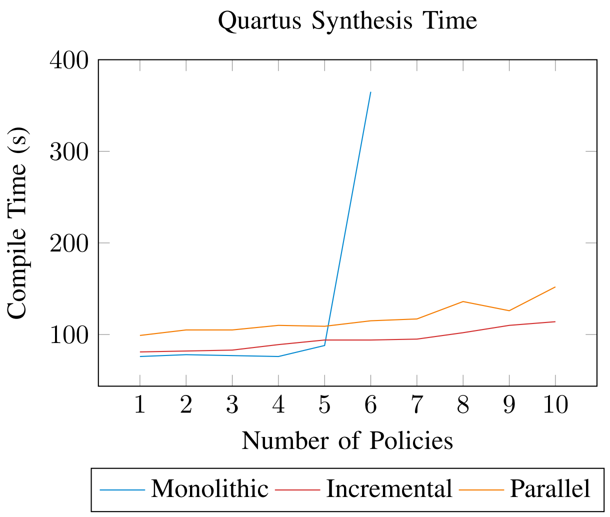


**FIGURE 30.** Time taken for quartus to synthesise hardware for each composition as the policy count increases.

### 2) COMPILE SIZE

The size of compiled XML files is reported in kilobytes (kBs) in Figure 31. Exponential trends are illustrated for each composition type, a steeper curve for monolithic with a peak XML size of 16,877 kB in the six policy composition, and a peak of 10,963 kB for incremental and parallel in the 10 policy compositions.

The size of compiled Verilog files is reported in kBs in Figure 29. The left chart includes all data and the right has monolithic compositions five and six removed to visualise the incremental and parallel trends more clearly. An exponential trend is illustrated for the monolithic Verilog files, with a peak of 14,971 kB for the six policy composition. Parallel and incremental results peak at 574 kB and 583 kB respectively.

Compile size trends show the proposed parallel and existing incremental methods are more scalable than monolithic. However, these approaches still show exponential trends in XML size. This suggests a XML structure specalised for compositions may further improve scalability.

### B. HARDWARE

#### 1) LOGIC ELEMENTS

The number of logic elements used, by the synthesised hardware components, is illustrated in Figure 32. The incremental and parallel trends are piecewise linear. The first six policies (thermal and current protection policies) are relatively simple compared to the final four (stepper motor protection policies). This is reflected in the steeper linear trend for compositions seven through to ten.

No significant monolithic trend is observed in the first six compositions, however the final composition of six policies consumed 27 logic elements compared to the previous three which all consumed seven. This hints that the expected exponential trend may be beginning. However, support for more complex monolithic compositions would be required to verify this.

These low total logic element counts demonstrate the desired low overhead of runtime enforcement hold for our proposed compositional runtime enforcement.

#### 2) REGISTERS

The number of registers in synthesised hardware is illustrated in Figure 33. The monolithic results show insignificant register synthesis with a peak of eight registers for the six policy composition, as they are only required to hold the current state of the enforcer. Additional results for this would be interesting to confirm the monolithic register count remains relatively low.

The incremental and parallel results reflect a piecewise linear trend consistent with the logic element results. Specifically, the increased complexity of policies added in compositions seven through ten results in a steeper linear growth in synthesised registers.

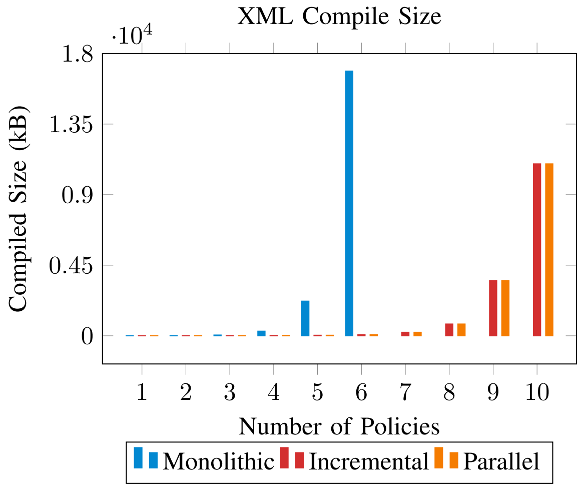


**FIGURE 31.** Size of compiled XML policy descriptions for each composition as the number of policies increases.

#### 3) MAXIMUM CLOCK FREQUENCY

The maximum clock frequency, in MHz, is reported in Figures 34 and 35. The chart in Figure 34 includes all results, in which the monolithic method shows a downward trend as policy complexity increases. However, Quartus optimisation improves clock frequency for compositions four and five.

The chart in Figure 35 has monolithic results removed to better illustrate the differences between incremental and parallel compositions. The incremental method shows a negative exponential decay trend in the maximum clock frequency with a minimum of 6.5 MHz for the ten policy composition. The parallel method shows a more inconsistent frequency between 75 and 80 MHz for compositions one to six. The frequency drops significantly to between 40 and 50 MHz when the more complex stepper motor policies are added in compositions seven through to ten.

The results here reflect a key difference between the incremental and parallel methods. As you increase the number of policies, the time taken to execute the incremental approach increases. In the parallel approach this is not the case.

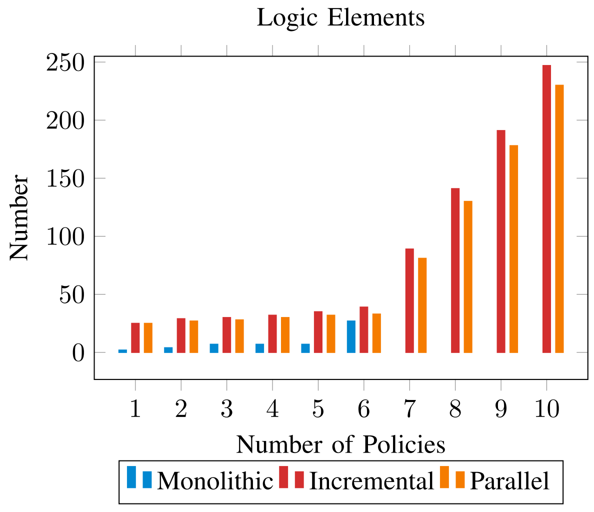


**FIGURE 32.** Number of logic elements synthesised for each composition as the number of policies increases.

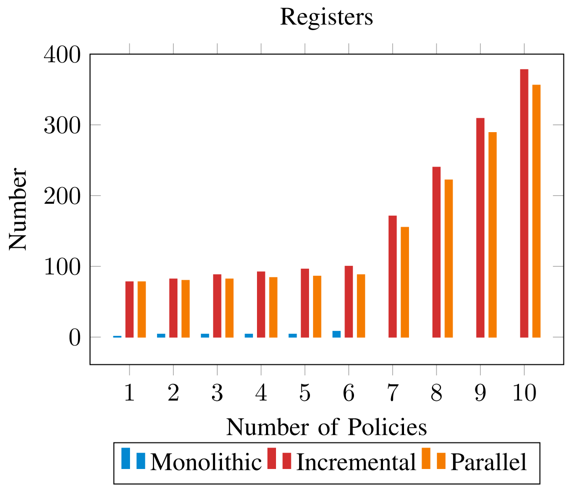


**FIGURE 33.** Number of registers synthesised for each composition as the number of policies increases.

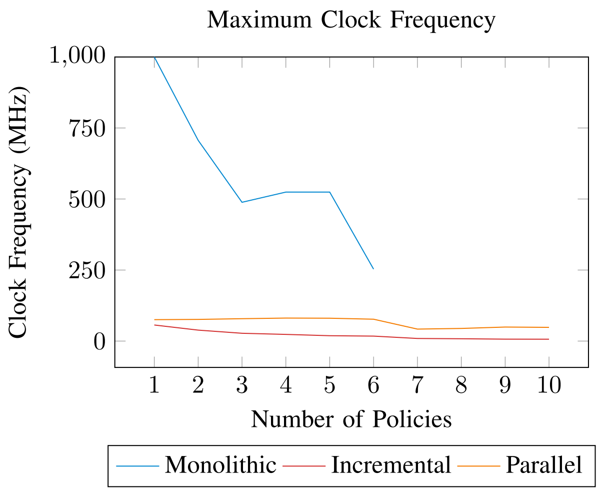


**FIGURE 34.** Maximum clock frequencies for monolithic, incremental, and parallel with includes all data points.

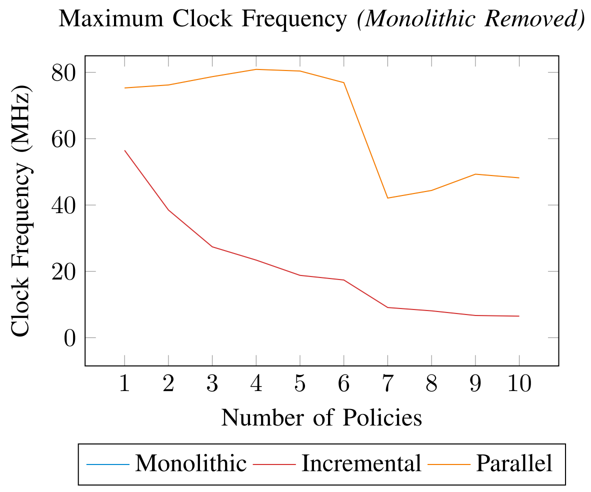


**FIGURE 35.** Maximum clock frequencies for incremental and parallel compositions.

## X. DISCUSSION

### A. PARALLEL HARDWARE OR INCREMENTAL SOFTWARE?

In this work, we have proposed a framework for multiple hardware enforcers to enforce simultaneously without suffering from the state space explosion of the monolithic approach (which uses the product of policies). In earlier work, we proposed multiple software enforcers to enforce sequentially, or incrementally. For a designer wishing to deploy enforcers, there is a trade-off between these approaches.

We have discussed hardware, relative to software, benefits from a reduced attack surface, which malicious actors may exploit (Section VI). Through our results we show the clock frequency of the proposed parallel approach is higher than the incremental approach. Therefore, for maximum speed and a reduced attack surface, parallel enforcers in hardware may be the designer's best choice.

Incremental software enforcers, however, benefit from reduced cost. This is due to software deployment on cheaper microcontrollers, rather than more expensive FPGAs or custom hardware required for hardware enforcers. This also impacts the ability for redeployment or updates to enforcers. The incremental approach supports adding new enforcers to the chain, which can then be uploaded to the microcontroller. The hardware approach requires partial re-synthesis (See Section VIII-B4 for more) and either upload to the FPGA or entirely new custom hardware. Therefore, in applications where policies are expected to change or grow reguarly, a software approach may be the designer's approach.

Ultimately, the requirements of the application and desired level of security determines which approach is most appropriate. We note the approach proposed in this work is CPS-agnostic. This means it can be applied to any system that can have policies modelled as DTA, with boolean inputs and outputs, and can include applications more broad than CPS.

### B. CHALLENGES

During the development of this framework a number of challenges were encountered, two of sigificance we now discuss.

A key requirement of the parallel approach was to resolve scalability challenges faced by the monolithic approach. Therefore, the hardware design to support parallel composition needed to avoid any exponential growth in resource consumption. This proved to be of significant challenge.

Conceptually, for each input/output event to be enforced, there is a set of possible inputs and outputs that are possible. Each enforcer has a subset of these events which it deems are satisfactory to the policy. These subsets need to be combined (intersection) and a final event needs to be selected. It was the intersection operation that was most challenging to optimise.

After a range of approaches were trialled, the violation recovery table design presented in Section VIII-A1 was selected as the most resource efficient method. This rested on the precalculation of intersections and a final look up table of recovery actions.

The other noteworthy challenge was producing results for the monolithic approach. As this approach is plagued by rapid exponential growth, the time taken to compile the enforcers quickly grew from minutes to hours. This was amplified by the existing monolithic compiler not having sufficient memory management to support larger compositions. Depsite compilation attempts on cloud based virtual machines with significant compute and memory capacity, some compositions had not completed after a week, and hence our monolithic results end after six combined policies.

Though beyond the scope of this paper, there is value in developing an improved monolithic compiler. This would expand comparison with our propsed approach.

## XI. CONCLUSION AND FUTURE WORK

The importance of safe and secure Cyber-Physical Systems (CPSs) is made clear by the significant human impact when they fail. Applying Runtime Enforcement to these

CPSs provides a method to ensure security policies are not violated during system execution. However, as the number of policies increases, methods to compose these policies becomes important.

In this work, we investigated parallel composition of bi-directional hardware enforcers for enforcing security policies for a FFF 3D printer. We use hardware enforcers as this reduces reliance on a tech stack (e.g. firmware, software, and operating system) with frequently discovered vulnerabilities. In CPSs where updates can be limited, a high trust execution platform is desirable.

We propose a novel bi-directional hardware RE framework that composes policies in parallel. This is the first work that addresses potential security vulnerabilities in software based enforcers and supports parallel policy composition for reactive systems. We provide a tool, *easy-rte-parallel*, which produces Verilog descriptions of these enforcers from a high-level policy description. We compare our proposed approach with monolithic and serial (incremental) methods to policy composition.

Our results demonstrate the expected state space explosion in the monolithic approach and more scalable increases in resource consumption by both serial (incremental) and our proposed parallel approach. Specifically, we show the consumption of resources is linear and proportional to the complexity of the composed policies. We also show higher clock frequencies compared to the serial (incremental) approach for the same number of composed policies. The results show our proposed approach is well suited where numerous security policies are required, like CPSs such as 3D printers.

In the future, we would like to demonstrate our hardware enforcers in a wider range of CPSs, study the possibility of combining parallel and serial (incremental) approaches, and distribution of enforcers.

## APPENDIX A

*proof of Theorem 2:*

We prove that given two policies $\varphi_1$, $\varphi_2$ defined as SA, and where $\varphi = \varphi_1 \cap \varphi_2$, if policy $\varphi$ is enforceable, then $E_{\varphi_1}||E_{\varphi_2}$ as per Definition 6 is an enforcer for $\varphi$ (satisfies the Soundness, Transparency, Monotonicity, Instantainety, and Causality constraints).

Let us first recall the constraints from [19] that an enforcer for any given policy $\varphi$ should satisfy. We have informally discussed about these constraints in Section IV, and more details and explanation about of Definition 7 is in [19].

*Definition 7 (Enforcer for $\varphi$): 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***

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

***Causality***

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

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 $\sigma$, i.e., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ satisfies the *(Snd)*, *(Tr)*, *(Mono)*, *(Inst)*, and *(Cau)* constraints. We have $E_{\varphi_1}||E_{\varphi_2}(\sigma) \in \varphi$ and $E_I(\sigma_I) \in \varphi_I$. Let us denote $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ using $\sigma'$, and $E_I(\sigma_I)$ using $\sigma'_I$.

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 $E_I(\sigma_I) \cdot x_{k+1} \in \varphi_I$. From the definition of $E_I$ in Def. 6, in this case, we have $x_{k+1} \in \cap(\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I), \mathsf{SelectI}_{\varphi_2}(\sigma'_I, \Sigma_I))$. Thus, $E_I(\sigma_I) \cdot x_{k+1} = \sigma'_I \cdot x_{k+1}$.
  From the Definition of $E_O$, in this case, we have $y_{k+1} \in \cap(\mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O), \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \Sigma_O))$. Thus, $E_O(\sigma_I \cdot x_{k+1}, \sigma_o \cdot y_{k+1}) = \sigma' \cdot (x_{k+1}, y_{k+1})$.
  Thus the output of the enforcer is $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$.

  About the *(Snd)* constraint, what was earlier released as output by the enforcer (before reading event $(x_{k+1}, y_{k+1})$) i.e., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ followed by the input-output event newly emitted as output $(x_{k+1}, y_{k+1})$ satisfies the property $\varphi_1 \cap \varphi_2$. Thus constraint *(Snd)* holds.

  The constraint *(Mono)* holds because $\sigma \preccurlyeq \sigma \cdot (x_{k+1}, y_{k+1})$, and $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ which is $\sigma'$ is a prefix of $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.

  Regarding constraint *(Inst)* from the induction hypothesis, we have for $\sigma$ of some length $k$, $|\sigma| = |E_{\varphi_1}||E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, 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., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ 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 $E_{\varphi_1}||E_{\varphi_2}$ as per Definition 6 up on reading $(x_{k+1}, y_{k+1})$ is $E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y_{k+1})$.

Regarding constraint *(Cau)*, from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x_{k+1} \in \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, \Sigma_I)$. Also, we have $y_{k+1} \in \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O)$, and $y_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \Sigma_O)$.
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.
  - $E_\varphi(\sigma) \cdot x_{k+1} \in \varphi_I$

    From the definition of $E_I$ in Def. 6, in this case, we have $x_{k+1} \in \cap(\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I), \mathsf{SelectI}_{\varphi_2}(\sigma'_I, \Sigma_I))$. Thus, $E_I(\sigma_I) \cdot x_{k+1} = \sigma'_I \cdot x_{k+1}$.
    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 $E_O$ and SelectO, i.e., $\mathsf{Rand}(\cap(\mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O), \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \Sigma_O))) = y'_{k+1}$.
    So the output of the enforcer is $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y'_{k+1})$.

    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., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$) 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 $E_{\varphi_1}||E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.

    Regarding constraint *(Inst)* from the induction hypothesis, we have for $\sigma$ of some length $k$, $|\sigma| = |E_{\varphi_1}||E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y'_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, 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., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ followed by the new input-output event read $(x_{k+1}, y_{k+1})$ does not satisfy the property $\varphi_1 \cap \varphi_2$. Thus, *(Tr)* holds trivially in this case. We already saw that the output event released by the enforcer $E_{\varphi_1}||E_{\varphi_2}$ as per Definition 6 up on reading $(x_{k+1}, y_{k+1})$ is $E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x_{k+1}, y'_{k+1})$.

    Regarding constraint *(Cau)*, from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x_{k+1} \in \Sigma_I = \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, \Sigma_I)$. Also we have, $y'_{k+1} \in Y \subseteq \Sigma_O = \mathsf{SelectO}_{\varphi_1}(\sigma', x_{k+1}, \Sigma_O)$, and $y'_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x_{k+1}, \Sigma_O)$.
    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 $E_I$ and SelectI, we have
    $\mathsf{Rand}(\cap(\mathsf{SelectI}_{\varphi_1}(\sigma'_I, \Sigma_I), \mathsf{SelectI}_{\varphi_2}(\sigma'_I, \Sigma_I))) = x'_{k+1}$
    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 $E_O$ and SelectO, we have $y_{k+1} \in \cap(\mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O), \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \Sigma_O))$.
      So the output of the enforcer is $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y_{k+1})$.

      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., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$) 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 $E_{\varphi_1}||E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.

      Regarding constraint *(Inst)* from the induction hypothesis, we have for $\sigma$ of some length $k$, $|\sigma| = |E_{\varphi_1}||E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint *(Inst)* holds.

      Constraint *(Tr)* holds trivially in this case since the output of the enforcer before reading $(x_{k+1}, y_{k+1})$ i.e., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ followed by the new input-output event read $(x_{k+1}, y_{k+1})$ does not satisfy the property $\varphi_1 \cap \varphi_2$.

      Regarding constraint *(Cau)*, from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x'_{k+1} \in \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also
      $x'_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, \Sigma_I)$. Also we have, $y_{k+1} \in \mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O)$, and $y_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \Sigma_O)$.
      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 $E_O$ and SelectO, we have $\mathsf{Rand}(\cap(\mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O), \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \Sigma_O))) = y'_{k+1}$.
      So the output of the enforcer is $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$.

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., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$) 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 $E_{\varphi_1}||E_{\varphi_2}(\sigma) \preccurlyeq E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))$.

Regarding constraint (*Inst*) from the induction hypothesis, we have for $\sigma$ of some length $k$, $|\sigma| = |E_{\varphi_1}||E_{\varphi_2}(\sigma)|$. We also have $E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1})) = E_{\varphi_1}||E_{\varphi_2}(\sigma) \cdot (x'_{k+1}, y'_{k+1})$. Thus, $|\sigma \cdot (x_{k+1}, y_{k+1})| = |E_{\varphi_1}||E_{\varphi_2}(\sigma \cdot (x_{k+1}, y_{k+1}))| = k + 1$, and constraint (*Inst*) holds.

Constraint (*Tr*) holds trivially in this case since the output of the enforcer before reading $(x_{k+1}, y_{k+1})$ i.e., $E_{\varphi_1}||E_{\varphi_2}(\sigma)$ followed by the new input-output event read $(x_{k+1}, y_{k+1})$ does not satisfy the property $\varphi_1 \cap \varphi_2$.

Regarding constraint (*Cau*), from the definitions of $\mathsf{SelectI}_{\varphi_I}$ and $\mathsf{SelectO}_{\varphi}$, we have $x'_{k+1} \in \mathsf{SelectI}_{\varphi_1 I}(\sigma'_I, \Sigma_I)$, and also $x'_{k+1} \in \mathsf{SelectI}_{\varphi_2 I}(\sigma'_I, \Sigma_I)$. Also we have, $y'_{k+1} \in \mathsf{SelectO}_{\varphi_1}(\sigma', x'_{k+1}, \Sigma_O)$, and $y'_{k+1} \in \mathsf{SelectO}_{\varphi_2}(\sigma', x'_{k+1}, \Sigma_O)$.

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. Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput. (ISORC)*, May 2008, pp. 1–7. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/427.html

[2] *To Kill a Centrifuge: A Technical Analysis of What Stuxnets Creators Tried to Achieve*, Langner Group, Hamburg, Germany, 2013.

[3] 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.

[4] CNN. (2021). *Colonial Pipeline Ceo Admits to Authorizing 4.4 Million Ransomware Payment*. [Online]. Available: https://edition.cnn.com/2021/05/19/politics/colonial-pipeline-ransom/index.html

[5] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[6] E. Bertino and N. Islam, "Botnets and Internet of Things security," *Computer*, vol. 50, no. 2, pp. 76–79, Feb. 2017.

[7] F. Trujano, B. Chan, G. Beams, and R. Rivera, "Security analysis of DJI phantom 3 standard," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. 1, 2016.

[8] S. Belikovetsky, M. Yampolskiy, J. Toh, J. Gatlin, and Y. Elovici, "dr0wned—Cyber-physical attack with additive manufacturing," in *Proc. WOOT*, 2017, pp. 1–15.

[9] (2001–2002). *Runtime Verification*. Accessed: Jul. 21, 2023. [Online]. Available: https://runtime-verification.github.io/

[10] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Proc. 14th Int. Symp. Formal Methods Formal Methods (FM)*, Hamilton, ON, Canada. Berlin, Germany: Springer, Aug. 2006, pp. 573–586.

[11] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL semantics for runtime verification," *J. Log. Comput.*, vol. 20, no. 3, pp. 651–674, Jun. 2010.

[12] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 1–64, Sep. 2011.

[13] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties," in *Proc. Int. Workshop Runtime Verification*. Berlin, Germany: Springer, 2009, pp. 40–59.

[14] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets*. Berlin, Germany: Springer, 1996, pp. 429–528.

[15] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, Feb. 2000.

[16] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, pp. 1–41, Jan. 2009.

[17] Y. Falcone, "You should better enforce than verify," in *Proc. 1st Int. Conf. Runtime verification (RV)*, in Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Berlin, Germany: Springer-Verlag, 2010, pp. 89–105.

[18] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: Composition, synthesis, and enforcement abilities," *Formal Methods Syst. Design*, vol. 38, no. 3, pp. 223–262, Jun. 2011.

[19] 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: ACM, Jul. 2017, pp. 80–89.

[20] A. Panda, A. Baird, S. Pinisetty, and P. Roop, "Incremental security enforcement for cyber-physical systems," *IEEE Access*, vol. 11, pp. 18475–18498, 2023.

[21] 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, Feb. 2015.

[22] 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.

[23] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Proc. IEEE Symp. Secur. Privacy. (S&P)*, May 2000, pp. 177–187.

[24] 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.

[25] 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. Human Factors Comput. Syst.*, Apr. 2008, pp. 1473–1482.

[26] 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.

[27] S. Pinisetty and S. Tripakis, "Compositional runtime enforcement," in *Proc. NASA Formal Methods Symp. (NFM)*, Minneapolis, MN, USA. Berlin, Germany: Springer, 2016, pp. 82–99.

[28] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: Runtime enforcement for reactive systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Scienc), vol. 9035. Berlin, Germany: Springer, 2015.

[29] Q. Hou and J. Dong, "Robust adaptive event-triggered fault-tolerant consensus control of multiagent systems with a positive minimum interevent time," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 53, no. 7, pp. 4003–4014, 2023.

[30] Q. Hou and J. Dong, "Cooperative fault-tolerant output regulation of linear heterogeneous multiagent systems via an adaptive dynamic event-triggered mechanism," *IEEE Trans. Cybern.*, vol. 53, no. 8, pp. 5299–5310, 2022.

[31] 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.

[32] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Algebraic Methodology and Software Technology (AMAST)*. London, U.K.: Springer, 1994, pp. 83–96.

[33] 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*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–9, doi: 10.1145/3359986.3361200.

[34] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into software-exploitable hardware bugs," in *Proc. 28th USENIX Secur. Symp. (USENIX Security)*. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[35] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An overview of hardware security and trust: Threats, countermeasures, and design tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 6, pp. 1010–1038, Jun. 2021.

[36] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proc. IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug. 2014.

[37] F. P. W. Melchels, M. A. N. Domingos, T. J. Klein, J. Malda, P. J. Bartolo, and D. W. Hutmacher, "Additive manufacturing of tissues and organs," *Prog. Polym. Sci.*, vol. 37, no. 8, pp. 1079–1104, Aug. 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0079670011001328

[38] U. Kalsoom, P. N. Nesterenko, and B. Paull, "Recent developments in 3D printable composite materials," *RSC Adv.*, vol. 6, no. 65, pp. 60355–60371, 2016.

[39] M. Kalender, S. E. Kiliç, S. Ersoy, Y. Bozkurt, and S. Salman, "Additive manufacturing and 3D printer technology in aerospace industry," in *Proc. 9th Int. Conf. Recent Adv. Space Technol. (RAST)*, Jun. 2019, pp. 689–694, doi: 10.1109/RAST.2019.8767881.

[40] D. Wu, A. Ren, W. Zhang, F. Fan, P. Liu, X. Fu, and J. Terpenny, "Cybersecurity for digital manufacturing," *J. Manuf. Syst.*, vol. 48, pp. 3–12, Jul. 2018.

[41] M. Yampolskiy, W. E. King, J. Gatlin, S. Belikovetsky, A. Brown, A. Skjellum, and Y. Elovici, "Security of additive manufacturing: Attack taxonomy and survey," *Additive Manuf.*, vol. 21, pp. 431–457, May 2018.

[42] N. Gupta, A. Tiwari, S. T. S. Bukkapatnam, and R. Karri, "Additive manufacturing cyber-physical system: Supply chain cybersecurity and risks," *IEEE Access*, vol. 8, pp. 47322–47333, 2020.

[43] G. Häußge. *OctoPrint*. Accessed: Jul. 21, 2023. [Online]. Available: https://octoprint.org/

[44] M. McCormack, S. Chandrasekaran, G. Liu, T. Yu, S. DeVincent Wolf, and V. Sekar, "Security analysis of networked 3D printers," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2020, pp. 118–125.

[45] M. B. Barcena and C. Wueest, "Insecurity in the Internet of Things," in *Proc. Secur. Response, Symantec* 2015, pp. 1–20.

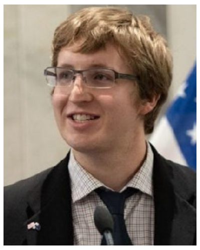
[46] M. O'Neill, "Insecurity by design: Today's IoT device security problem," *Engineering*, vol. 2, no. 1, pp. 48–49, Mar. 2016.

[47] H. Pearce, K. Yanamandra, N. Gupta, and R. Karri, "FLAW3D: A trojan-based cyber attack on the physical outcomes of additive manufacturing," *IEEE/ASME Trans. Mechatronics*, vol. 27, no. 6, pp. 5361–5370, Dec. 2022.

[48] TRINAMIC Motion Control GmbH Co. KG. (2023). *MC2130 Datasheet*. [Online]. Available: https://www.trinamic.com

**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.

**ABHINANDAN PANDA** received the M.Tech. degree in information and communication technologies from the Indian Institute of Technology Kharagpur, India, in 2014. Currently, he is pursuing the Ph.D. 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.



**HAMMOND PEARCE** received the B.E. degree (Hons.) in computer systems engineering and the Ph.D. degree in computer systems engineering from The University of Auckland, New Zealand, in 2016 and 2020, respectively. He is currently a Lecturer with the School of Computer Science and Engineering, University of New South Wales. Previously, he was with the Department of Electrical and Computer Engineering, NYU, and NYU Center for Cybersecurity as a Research Assistant Professor. His research interests include cybersecurity and hardware and embedded systems design, and the intersection of AI and industrial informatics in this area.

**SRINIVAS PINISETTY** 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 the TU/e for two years. For his master's thesis project, he worked with ASML, Veldhoven, The Netherlands, in 2009, and he worked as a Software Design Engineer Trainee with 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 was a Postdoctoral Researcher with the University of Aalto, Espoo, Finland, and later with the University of Gothenburg–Chalmers, Gothenburg, Sweden. His research interests include formal methods, and 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.