PatrIoT: Policy Assisted Resilient Programmable IoT System

Moosa Yahyazadeh[1], Syed Rafiul Hussain[2], Endadul Hoque[3], and Omar Chowdhury[1]

[1] The University of Iowa, Iowa City, IA 52242, USA
{moosa-yahyazadeh,omar-chowdhury}@uiowa.edu
[2] The Pennsylvania State University, University Park, PA 16802, USA
hussain1@psu.edu
[3] Syracuse University, Syracuse, NY 13244, USA
enhoque@syr.edu

Abstract. This paper presents PatrIoT, which efficiently monitors the behavior of a programmable IoT system at runtime and suppresses contemplated actions that violate a given declarative policy. Policies in PatrIoT are specified in effectively propositional, past metric temporal logic and capture the system's expected temporal invariants whose violation can break its desired security, privacy, and safety guarantees. PatrIoT has been instantiated for not only an industrial IoT system (EVA ICS) but also for two home representative automation platforms: one proprietary (SmartThings) and another open-source (OpenHAB). Our empirical evaluation shows that, while imposing only a moderate runtime overhead, PatrIoT can effectively detect policy violations.

Keywords: Runtime monitoring · IoT systems · Policy enforcement.

1 Introduction

Programmable IoT systems, that have seen deployments in regular households as well as advanced manufacturing plants, enable one to carry out *specialized automation functions* by instructing a group of (COTS) actuators to perform different tasks based on sensor values, events, and business logic. For supporting diverse automation tasks, these systems allow one to develop and deploy *automation apps/units* whose complexity can range from simple if-then-else rules to complex machine learning based programs. As a simple example, an IoT app in a metal melting plant can command a lifter-robot (*i.e.*, actuator) to load scrap metal on a conveyor belt only when the weight sensor reads more than 100 lbs.

These automation apps that *(i)* can be possibly obtained from unvetted sources, *(ii)* can be malicious, *(iii)* may have logical bugs, or *(iv)* may interact in unanticipated ways with other apps, can render the system to unexpected states. Such unexpected state transitions can halt a production line, create safety-hazards, or violate security and privacy guarantees of such systems. For instance, an IoT app in a melting plant can instruct the lifter-robot to load scrap metal on an already loaded conveyor belt, severely damaging it and creating a safety hazard. *This paper focuses on developing a runtime policy enforcement approach for ensuring that a system under monitoring does not reach such unexpected states.*

A majority of the existing work relies on static analysis of apps to identify such undesired behavior [39,24,37,17,30,20] but suffers from one of the following limitations: *(i)* false positives due to conservative analysis; *(ii)* false negatives due to single app analysis; or *(iii)* scalability issues due to state-space explosion. Existing runtime enforcement based approaches [18,41,28], on the contrary, have one of the following limitations: *(i)* requires constructing the whole state-space of the global IoT system statically which is infeasible for large systems; *(ii)* cannot enforce rich temporal policies; *(iii)* policy specification is based on the transitions on the global state-space which is extremely inconvenient.

In this paper, we present PatrIoT which monitors the contemplated actions of programmable IoT apps at runtime and denies them only when they violate a declarative policy. The policy language of PatrIoT can refer to any past events/actions/system states, impose explicit-time constraints between any past events/states, and contain predicates over quantitative aspects of the system execution (*e.g.*, number of times an event has occurred in the last 10 seconds). Technically, this language can be automatically translated to a first-order, past-time metric temporal logic (MTL) with some aggregation functions (e.g., count, mean). The first-order logic portion of it, modulo aggregation functions, is restricted to a fragment of the BernaysSchönfinkel class (or, effectively proposition logic or EPR). This conscious choice allows us not only to express a rich set of policies but also to enforce our policies efficiently by drawing inspirations from existing runtime monitoring algorithms [12,13,14,35,36,15]. Unlike first-order MTL, for enforcing our policies, it is sufficient to store only truth values of sub-formulas with auxiliary structures of the immediate previous state instead of any past variable substitutions.

To show PatrIoT's generality, we instantiated it for 3 representative IoT systems, namely, EVA ICS [4] in the context of Industrial IoT systems, and also Samsung SmartThings [7] and OpenHAB [5], in the context of home automation. For instantiating PatrIoT, we resort to inline reference monitoring in which we automatically instrument each app by guarding high-level APIs for performing actions with a call to the policy decision function. We develop automatic instrumentation approach for each of the platforms. In addition, as automation apps can run concurrently, one has to also use appropriate synchronization mechanisms (e.g., mutex) to avoid any inconsistencies during state update of the reference monitor. We needed to design such a synchronization mechanism for SmartThings (e.g., mutex) as its programming interface did not provide any.

In an empirical evaluation with two case studies, one based on metal melting plant and another based on home automation, we observed that PatrIoT was able to mitigate all the undesired state transitions while incurring an average of 137 milliseconds of runtime overhead. We have also designed PatrIoT as a standalone library which other platforms can use to perform policy checking.

To summarize, the paper has the following technical contributions:

– We propose PatrIoT, which monitors the execution of an IoT system at runtime and prevents it from entering an undesired state by denying actions that violate a policy.
– We instantiate PatrIoT for three representative programmable IoT platforms, namely, EVA ICS [4], SmartThings [7], and OpenHAB [5]. Our evaluation with these instantiations show that they are not only effective in identifying non-compliant actions but also efficient in terms of runtime overhead.

2 Scope and Motivation

Most of the IoT ecosystems, despite being complex, share a similar architecture. Figure 1 shows a general IoT system consisting of a wide range of devices (*e.g.*, robot arm, surveillance cameras, smart lights) and a programmable IoT platform, which serves as a centralized backend of the system. The backend typically exposes some programming interfaces for IoT apps (or a system control panel) to automate the system (or directly control the devices, respectively).

The backend is a (logically) centralized server, dedicated for storage and computation, is responsible to synchronize the physical world with the cyber world. It can be physically co-located or placed in a remote cloud. Nevertheless, it provides a human machine interface to enable remote control and monitoring. Nowadays, most IoT platforms expose programming interfaces for programmers to develop flexible and customized IoT apps. The app execution engine coordinates the execution of all IoT apps in which the automated operations of the IoT system occur as they directly guide (and possibly, alter) the physical processes.

All actions taking place in an IoT system form its behavior. The actions in IoT apps are based on the automation tasks in that system. Given an IoT app's business logic, commanding an action might not only depend on the devices' events for which the IoT app (issuing the action) is registered but also hinge on the current context in the system. The current context for an action is formed by the snapshot of the system state right before taking the action. As the complexity of an IoT system (including its automation tasks) scales and grows, the dependency between actions and the system state becomes more tangled. Hence, to capture the expected behavior, this complexity of the dependency between action and the system state needs to be taken into account.

Policy. The expected behavior of an IoT system can be captured via a set of expressions, called *policies*. All policies must be invariably maintained by the system at runtime while the IoT apps are active and operating. A policy is maintained if it is satisfied by an impending action of any IoT app given the current execution trace. Thus, if an action respects every policy, then the system is considered to comply with the policies and allows the action to be executed. In case of a violation, an action must be denied to prevent potential repercussions.

Threats. Any unwanted behavior, caused by single IoT app or *unintended interplays* among multiple IoT apps, can impose security, privacy, or safety threats to the IoT system. The security threats exist if an unwanted action impairs an

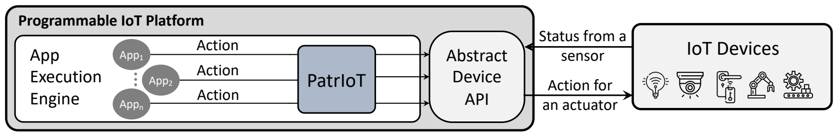

Fig. 1. A programmable IoT platform with PatrIoT installed. Without PatrIoT, every `Action` would be directly forwarded to the device API.

IoT system by changing its state such that it cannot perform its intended functions (*i.e.*, targeting system integrity) or gives an unauthorized access to the protected data (*i.e.*, violating data confidentiality). Depending on the criticality level of the IoT apps, such unwanted actions can also threaten the safety of the system (*e.g.*, failing to turn on the fire sprinkler of the factory floor when there is a fire). Not to mention that allowing some of those action can also violate the privacy (*e.g.*, posting a tweet by a medicine-reminder app unbeknown to the user). An IoT system can face such threat due to a number of vulnerabilities including cross-app interference, race conditions due to concurrent executions of apps, a lack of fine-grained access control mechanisms in the underlying platforms, and semantic bugs in apps. As an example, let us let us consider cross-app interference vulnerabilities specialized for programmable IoT systems.

Since IoT app's action can change a device state and/or a physical processes (*i.e.*, the environment), it can generate additional trigger(s) which in turn can execute another IoT app. Such interplay among IoT apps can be either *explicit* or *implicit* [20]. In an explicit interplay, the outcome of an action directly triggers other IoT app. For instance, running "*if the weight sensor detects some weights then turn on the conveyor belt*" explicitly triggers "*if the conveyor belt is rolling then turn on the object detector*". Contrarily, in an implicit interplay, the outcome of an action changes some attributes of the physical environment which can consequently trigger other IoT app. For example, "*if water temperature is greater than 200 °F then open the outgoing valve of the cooler to drain hot water*" implicitly triggers "*if water level is low then open the incoming valve to refill the cooler with cold water*".

Attacker Model. In this paper, we assume an attacker is capable of launching the unwanted behavior by *(i)* developing the malicious IoT apps exploiting the above vulnerabilities or *(ii)* simply misusing the existing faulty IoT apps, where the latter does not necessarily need to involve the attacker, yet is unwittingly introduced by a developer. For smart home based systems, a malicious app can creep in to the system through *unvetted* marketplaces from which users often obtain IoT apps. Contrarily, IoT systems like IIoT may not have open marketplaces, but they are prone to — (i) insiders who can carry and install malicious apps, and (ii) untrusted third-party developers. Any undesired situation resulted from compromised IoT devices, by exploiting vulnerabilities in firmware or communication protocols, and even the IoT backend itself is beyond our scope.

Motivating example. Consider a smart building where access restrictions are imposed on the entry of several rooms. An IoT integrated reader installed near the entry unlocks the door when a presented credential (*e.g.*, smart card) is authenticated. This one-way security entrance operation is one of the most popular and simplest access control solutions for smart buildings in which once someone is authorized entering a room, then they can easily exit the room because the door is unlocked from inside. In this situation, an unwanted behavior could occur when an unauthorized person sneaks into the room through a window or ventilation pipe and then they can freely open the door and enter into the building. A security measure preventing this undesirable situation is to check the following temporal policy before unlocking the door from inside:

Allow unlatching the door lock
only if the **number** of granted door access requests is
greater than the **number** of times door unlatched.

That is, unlatching the door from inside is allowed whenever someone entered into the room before.

3 Overview of PATRIOT

In this section, we present an abstract programmable IoT system model, our problem definition, and PatrIoT's architecture.

3.1 Abstract Model

A programmable IoT system $\mathcal{I}$ is viewed as a labeled transition system defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{V}, \mathcal{T} \rangle$, where $\mathcal{S}$ is a non-empty, finite set of states, $\mathcal{A}$ is a finite set of high-level activities in the IoT system, $\mathcal{P}$ refers to a finite set of all possible IoT apps supported by the underlying platform, $\mathcal{V}$ refers to a finite but arbitrary set of *typed* variables, and $\mathcal{T}$ is the transition relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ regulates how $\mathcal{I}$ changes its states while reacting to the activities. A state is total map that assigns each variable $v \in \mathcal{V}$ a value from an appropriate domain.

We consider $\mathcal{T}$ to be a *deterministic* and *left-total* relation (*i.e.*, no dead states). For all states $s_b, s_e \in \mathcal{S}$ and activity $a \in \mathcal{A}$, if $\langle s_b, a, s_e \rangle \in \mathcal{T}$ (alternatively, $s_b \overset{a}{\to} s_e$), then it suggests that when the system is in state s_b and an activity a is performed, then $\mathcal{I}$ will move to the state s_e. Given a state $s_b \in \mathcal{S}$ such that $s_b = [v_{\text{valve1}} \mapsto \mathsf{close}, \ldots]$ and an activity $a = \langle \mathit{caused\_by} \mapsto \mathsf{app}_1, \mathit{target\_device} \mapsto \mathsf{valve1}, \mathit{activity\_value} \mapsto \mathsf{open} \rangle$ by an app (called $\mathsf{app}_1 \in \mathcal{P}$), then $\mathcal{I}$ will transition to a state s_e, where $s_e = [v_{\text{valve1}} \mapsto \mathsf{open}, \ldots]$.

3.2 Problem Definition

Given a programmable IoT system $\mathcal{I}$, let σ be a finite execution trace of it. A trace σ is a finite sequence of states $\sigma = \langle s_0, s_1, s_2, \ldots, s_{n-1} \rangle$. We use $|\sigma|$ to denote the length of the trace σ. For each trace σ, we require that there exists an activity $a \in \mathcal{A}$ such that $s_i \overset{a}{\to} s_{i+1} \in \mathcal{T}$ where $0 \leq i < |\sigma| - 1$. Given $\mathcal{I}$ and a policy Ψ, in this paper, we want to ensure that each *action* activity performed by an $\mathsf{app} \in \mathcal{P}$ is compliant with Ψ, formalized as below. Given a programmable IoT system $\mathcal{I}$, a policy Ψ, a valid trace $\sigma = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ of $\mathcal{I}$, and an action activity a_c, the *policy compliance problem* is to decide whether $\sigma' = \langle s_0, s_1, \ldots, s_{n-1}, s_n \rangle$ is compliant with Ψ where $s_{n-1} \overset{a_c}{\to} s_n \in \mathcal{T}$.

3.3 Architecture of PATRIOT

PatrIoT's reference monitor has the following three main components: *Policy Enforcement Point* (PEP), *Policy Decision Point* (PDP), and *Policy Information Point* (PIP). The PEP intercepts each action contemplated by `apps` installed in a programmable IoT system. It then consults the PDP to decide whether the action is compliant with the given policies. If the action is compliant, it is allowed to be carried out; otherwise, the action is denied. The PDP implements the *policy decision function* which takes as input an action, and returns Allow or Deny. It essentially solves the *policy compliance problem.* The PIP stores all the relevant information regarding the policy (*e.g.*, policy statements) and system information (*e.g.*, device status) that are necessary for policy decision.

4 Design of PatrIoT

In this section, we present the syntax and semantics of PatrIoT's policy language as well as PatrIoT's solution to policy compliance problem.

4.1 Policy Language Syntax

Note that, in what follows, we discuss the abstract syntax of our language. A policy consists of one or more *policy blocks.* Each policy block starts with a **`Policy`** keyword followed by an identifier. The body of each block can have one or more *policy statements* of the following form. Policy blocks are introduced to allow modularity; grouping similar policy statements under a named block.

```
allow/deny <target_clause>
      [only if <condition_clause>]
      [except <condition_clause>]
```

In a policy statement, we can use either **`allow`** or **`deny`** keyword to identify allow or deny statements, respectively. The *allow* (or, *deny*) statements capture the conditions under which certain actions are permitted (or, discarded, respectively). The `<target_clause>` is an expression that captures information about the actions for which the policy statement is applicable. It has the form of a non-temporal condition (i.e., Ψ) shown below. We allow a special wildcard keyword, **`everything`**, in place of `<target_clause>`, to denote all possible actions. The optional **`only if`** portion contains a `<condition_clause>` that expresses a logical condition under which the action, identified by the `<target_clause>`, will be allowed (or, denied). The optional **`except`** portion contains a `<condition_clause>` that captures the exceptional cases under which the restriction expressed by the `<condition_clause>` in **`only if`** portion can be disregarded.

$$\begin{aligned}
\textsf{(Term)}\, t &::= v \mid c \mid f(\Phi) \\
\textsf{(Non-temporal Condition)}\Psi &::= \textbf{true} \mid \textbf{false} \mid \mathsf{P}(t_1, t_2) \mid \textbf{not}\,\Psi_1 \mid \Psi_1 \textbf{or}\, \Psi_2 \mid \Psi_1 \textbf{and}\, \Psi_2 \\
\textsf{(Temporal Condition)}\Phi &::= \Psi \mid \mathsf{Since}_{[\ell,r]}(\Phi_1, \Phi_2) \mid \mathsf{Lastly}_{[\ell,r]}(\Phi_1) \mid \mathsf{Once}_{[\ell,r]}(\Phi_1) \mid \\
&\quad\ \textbf{not}\,\Phi_1 \mid \Phi_1 \textbf{or}\, \Phi_2 \mid \Phi_1 \textbf{and}\, \Phi_2 \\
(\langle \textsf{condition\_clause} \rangle) &::= \Phi | \Psi
\end{aligned}$$

A condition clause can be either be a temporal condition or a non-temporal condition. A non-temporal formula (i.e., Ψ) can be **true**, **false**, a predicate or their logical combinations. We use **and**, **or**, and **not** to denote the logical conjunction, disjunction, and negation, respectively. We only consider binary *predicates* P where one of its arguments is a constant. A *term* is either a variable v, a constant c, or an aggregation function f. Currently, we have the following standard predicates: $>$, $\geq$, $=$, $\neq$, $\leq$, $<$. Examples of predicates could be $\mathsf{Temperature} \geq 78$ and $\mathsf{Humidity} < 30$. We currently allow the Count aggregation function.

The condition clause can use three standard *past* temporal operators $\mathsf{Since}_{[\ell,r]}(\cdot,\cdot)$, $\mathsf{Lastly}_{[\ell,r]}(\cdot)$, and $\mathsf{Once}_{[\ell,r]}(\cdot)$. To enable condition clause to refer to explicit time differences between different state values, each of the temporal operators can optionally take an additional time interval $[\ell,\, r]$, where ℓ and r denote the lowerbound and upperbound such that $\ell, r \in \mathbb{R}^+ \cup \{0, \infty\}$ and $\ell \leq r$. If ℓ is 0, it points to the current state. r can be ∞ to allow temporal operators to refer to arbitrarily in the past. Using ℓ and r, we can adjust a time window on which a temporal operator can be applied.

Examples. Having understood the basics of policy language syntax, we can formally specify the policy given in motivating example 2 as follows:

```
POLICY Motivating_Example_1:
  ALLOW action_command = unlatch and action_device = door
  ONLY IF COUNT(ONCE(state(door_reader) = access_granted)) >
          COUNT(ONCE(state(door) = unlatched))
```

4.2 Policy Language Semantics

We provide the semantics of policy language by converting any given policy to a quantifier-free, first-order metric temporal logic (QF-MTL) formula.

A policy Ψ consists of a sequence of policy statements $\langle ps_1, ps_2, \ldots, ps_n\rangle$. Given that each policy statement ps_i can be converted into a QF-MTL formula φ_i, the QF-MTL equivalent of Ψ denoted with φ can be logically viewed as combining φ_i with logical conjunctions (*i.e.*, $\varphi \equiv \bigwedge_{i=1}^{n} \varphi_i$). Conceptually, this is similar to using the "Deny overrides Allow" conflict resolution mechanism to combine compliance verdicts of the different policy statements. In this mechanism, an action is thus allowed only if all the policy statements have the Allow verdict (i.e., evaluates to true) for that action. Note that, if an action falsifies the <`target_clause`> component of each rule, then it is trivially allowed. It is also possible to easily extend the language to support other conflict resolution mechanisms (e.g., "Allow overrides Deny", "first applicable policy statement").

Our discussion of formal semantics will thus be complete as long as we describe how to convert each policy statement ps_i to its equivalent QF-MTL formula. In our presentation, $\varphi_{\text{applicable\_action}}$, $\varphi_{\text{condition}}$, and $\varphi_{\text{exception}}$ are meta-variables representing corresponding QF-MTL formulas capturing the applicable action, condition, and exception of a statement, respectively. We interpret the allow statement as the following QF-MTL formula: $\varphi_{\text{applicable\_action}} \Rightarrow \varphi_{\text{condition}} \wedge \neg\varphi_{\text{exception}}$. In the similar vein, we interpret the deny statement as the following QF-MTL formula: $\varphi_{\text{applicable\_action}} \Rightarrow \neg\varphi_{\text{condition}} \vee \varphi_{\text{exception}}$.

In case either the optional condition or exception block is missing, we consider them to be logical TRUE (*i.e.*, $\varphi_{\text{condition}} = \text{TRUE}$) or FALSE (*i.e.*, $\varphi_{\text{exception}} = \text{FALSE}$), respectively. When the condition block contains **`everything`**, we consider $\varphi_{\text{applicable\_action}} = \text{TRUE}$. Otherwise, obtaining $\varphi_{\text{applicable\_action}}$, $\varphi_{\text{condition}}$, and $\varphi_{\text{exception}}$ from policy syntax are straightforward. Each syntactic element are replaced by its logical equivalent (*e.g.*, **`not`** with $\neg$, **`or`** with $\vee$, and **`and`** with $\wedge$). Similarly, the temporal operators will be replaced by their usual equivalent. For a given temporal formula Φ, a trace σ, and a position i in it (i.e., $0 \leq i < \mid \Phi \mid$), we write $\sigma, i \models \Phi$ if and only if Φ evaluates to true in the i^{th} position of σ [6].

4.3 Policy Decision Function

PATRIoT's policy decision function (Δ) takes as input an attempted action a_c, the current execution trace $\sigma = \langle s_0, s_1, \ldots, s_{n-1} \rangle$, and a policy Ψ (whose QF-MTL equivalent is φ), then decides whether a_c is compliant with Ψ. In case a_c is compliant (i.e., φ evaluates to true), Δ returns ALLOW; it returns DENY, otherwise. PATRIoT's decision function checks whether $\sigma', n \models^? \varphi$ where $\sigma' = \langle s_0, s_1, \ldots, s_{n-1}, s_n \rangle$ and $s_{n-1} \xrightarrow{a_c} s_n \in \mathcal{T}$. For checking $\sigma', n \models^? \varphi$, we rely on standard runtime monitoring algorithms from the literature [11,13,14,22,35,36,15].

5 Implementation

To demonstrate the generality of PATRIoT, we have instantiated it for both industrial IoT (EVA ICS [4]) and smart-home systems (SmartThings [7] and OpenHAB [5]). Although these systems share similar design principles with respect to other IoT platforms, each presents unique challenges for PATRIoT instantiation. Since EVA ICS, SmartThings, and OpenHAB do not provide native APIs support for policy enforcement, we hooked PATRIoT in these platforms automation unit execution engine (shown in Figure 2) such that all PATRIoT's necessary components are realized by code snippets (automatically generated by an accompanying toolchain, which we call as the *instrumentor*). That is, the auto-generated code can be deployed inside the IoT platform alongside the apps logics. This makes PATRIoT self-contained since it does not require any custom service from the target platforms and can enforce policy compliance by the platform's app execution engine.

As shown in Figure 2, PATRIoT's instrumentor takes the policies and IoT apps as inputs and automatically generates instrumented-, ready-to-be-deployed-apps as outputs. The instrumentor is written as a Python script and internally uses its own parser (generated by ANTLR [1]) to parse policy language syntax (step ❶). Once parsed, its semantics will be encoded as a part of Policy Decision Point (PDP) in the platform's supporting language (step ❷). The PDP code is also accompanied by all the necessary codes retrieving information about the system/devices states. The instrumentor also parses the apps source code to find the actions of interest (*i.e.*, the function calls sending command). These actions are then guarded with an `if` block, predicated on a function call to PDP by

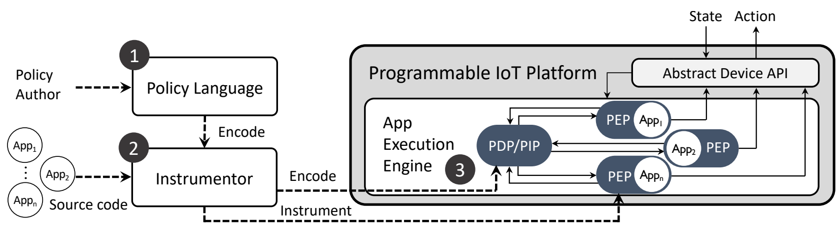

Fig. 2. The flowchart of automation tool to deploy PatrIoT with the necessary components inside the target programmable IoT platform.

passing the request context as its arguments. With that in place, PatrIoT can enforce the policies given the decision result from PDP at runtime. That is, the requested action is either allowed to be taken or simply needs to be dropped.

Note that, there are two main aspects need to be considered in the design of PDP. First, it should be deployed at a place reachable by every guarded action in the apps inside the execution engine, whilst preserving a global view of the entire system state. Second, each function call to PDP for each guarded action needs be synchronized. This is due to the fact that multiple action commands from different apps can be called roughly at the same time, which might cause some state change while the current thread running inside PDP has already read its old value. Therefore, once the decision has been made in PDP, it will be no longer valid since its premise might have changed. This is a well-known concurrency issue called Time-of-check Time-of-use (TOCTOU) race condition [3] which can be addressed by some synchronization mechanism. Note that, allowing only sequential actions is not restrictive, as eventually they will be serialized in the network modem; however, it may affect the performance of the system. Once the instrumentation is finished, the generated code can be run inside the automation execution engine of a platform (step ❸), which automatically prevents the system from entering into an undesirable state during its execution.

5.1 Platform-specific Implementation Details

In the following subsections, we discuss the important details of the instrumentation in our targeted IoT platforms.

EVA ICS EVA ICS is an IoT platform for the automated system development in both industrial and home environment [4]. The automation units in EVA ICS is called *macros* which mainly supports Python as its scripting language.
PEP. EVA ICS provides uniform APIs (e.g., `action()`, `action_toggle()`) for performing actions in a macro. Calling these functions with appropriate arguments result in a global state change. For instance, `action('zone1/lifter_robot', status=1)` will activate `lifter_robot` residing in `zone1`. Our instrumentor uses its own custom parser to spot these actions in macros' source code. Once identified, its arguments (*e.g.*, `'zone1/lifter_robot'` and `1`) are extracted to be then passed to PDP as part of the request context. The identified

action is then guarded with an `if` block, whose condition is a function call to PDP to which the pre-extracted information such as the target device and command name are passed as its arguments. EVA ICS also provides native support for locking mechanism via `lock()` and `unlock()` functions, which can be used to address synchronization issue discussed earlier.

PDP/PIP. EVA ICS runs macros whose source codes reside in a specific folder in the system. If any piece of code needs to be shared among them, it has to be stored in `common.py` and also located in the same folder with the apps. Given that it can be readily accessed by the other macros and any code inside of it can see the entire system states, `common.py` file is our target to store the policy decision function encoding the semantics of policy language policies. There are other platform-supported features used to capture the request context inside the PDP function (*e.g.*, `_00` variable for obtaining current macro's full identifier and `_source` variable to access the item generated the event).

SmartThings SmartThings is a cloud-based smart-home platform with a proprietary back-end that can provide automations among SmartThings-powered IoT devices. The automation units in SmartThings are called *SmartApps* which support a restricted subset of the groovy language [7].

PEP. SmartThings uses a variety of methods for performing an action. For example, it uses `on()` method of an object with `capability.switch` to turn it on (*e.g.*, `light1.on()`). Given a pre-compiled action list, the instrumentor parses the source code of SmartApps to find those actions and then guarding them with PEP-related statements. Since SmartApps are groovy-based programs, we use groovy meta-programming feature [2] which allows traversing the Abstract Syntax Tree (AST) of a SmartApp. To this end, we use a groovy script that uses `ASTTransformation` class to write a custom `ASTNode` visitor to spot each method call in the pre-compiled action list and then replacing it with a ternary operator such that in the condition portion it checks whether the function call to PDP is evaluated to be true. In the true branch, it performs the guarded action while in the false branch it logs that the action is denied. The `ASTNode` visitor also extracts the necessary request context and passes them to PDP as arguments. After visitor's pass, our groovy script translates the AST back to the source code and spits it out as the instrumented SmartApp.

PDP/PIP. Recall that, PDP needs to have a global view of the entire system. In SmartThings, this can be achieved through the Parent-Child SmartApps relationship structure in which the PDP is defined as a function inside a parent SmartApp, so-called policy manager, while the previously instrumented SmartApps are considered as its children. This setup not only features all SmartApps to call the same PDP function but also enables the PDP to access the state of all devices used by the SmartApps. Unfortunately, SmartThings does not provide any built-in synchronization primitives. To address concurrency issues, we have built PATRIoT lock management server which is **(i) RESTful**: SmartApps requests for a lock by a simple HTTP post (*e.g.*, http://⟨domain : port⟩/locks/PatriotLock). The server notifies them whether lock is acquired or not by a HTTP response code (*e.g.*, 201: The lock is acquired;

408: lock is not acquired). SmartApps release the lock by a HTTP delete; **(ii) Queue-based**: It simply uses FIFO approach to give the lock to the oldest request. The further requests have to wait for the one who acquired the lock to release it and then the older request can acquire it (if the request has not gotten timed-out); **(iii) Starving-free**: The created lock has a lifetime in seconds which server starts counting down from each request which successfully acquires the lock. So, if the client does not release the lock before the lock lifetime, server simply releases the lock; and **(iv) Secure**: It uses HTTPS for each request and release. Each request/release is authenticated (using a pre-shared key) and replay protected; therefore, a malicious entity cannot make illegal lock request/release.

OpenHAB OpenHAB is an open-source smart-home platform that can be deployed locally. The automation units in OpenHAB are called *rules* and are written in a domain specific language (DSL).
PEP. OpenHAB has a well-structured category of actions that can be used inside a rule to perform an action. Our instrumentor uses that to select the actions of interest. For instance, `sendCommand()` and `postUpdate()` are methods for sending a command to an item and updating an item's status, respectively. Since rules in OpenHAB are written in a DSL, we developed a custom parser to instrument a rule's source code. The synchronization issue in OpenHAB is handled using mutex lock provided by the platform.
PDP/PIP. In OpenHAB, rules written in the same file can share global variables and utilize the common functions. Given that, the instrumentor merge all rules into the single rule file and encode PDP as a common function.

6 Evaluation

In this section, we evaluate our instantiations of PatrIoT. The main goal of our evaluation is to demonstrate PatrIoT effectiveness in maintaining the user expectations and its efficiency in terms of runtime overhead on a host platform.

6.1 Effectiveness

To showcase the effectiveness of PatrIoT in each platform, we built a testbed containing several IoT devices and constructed different scenarios in which some undesired action(s) can occur.

EVA ICS

Testbed. The testbed is similar to a realistic production line of a chemical plant aiming to combine metal blocks with two other compounds in a furnace. Table 1 and Table 2 show the item list and simplified apps used for this testbed, respectively. To setup the testbed for EVA ICS, we deployed EVA ICS-3.2.4 [4] on a machine powered by an Intel Core i7-6700 3.40GHz CPU and with 32GB RAM.

Table 1. Item list used for EVA ICS chemical plant testbed

compound_valve01	liquid_level_indicator01 (lli01)
compound_valve02	presence_sensor01 (ps01)
conveyor_belt	presence_sensor02 (ps02)
drain_valve	quality_control_sensor01 (qc01)
lifter_robot	temperature_sensor01 (ts01)
lifting_arm	weight_sensor01 (ws01)
mixing_robot	weight_sensor02 (ws02)
stopper	processed_sensor
water_valve	

Table 2. Automation units used in EVA ICS testbed

ID	**AU Description**
AU_1	$\xrightarrow{\text{ws01 sensed}}$ activate lifter_robot
AU_2	$\xrightarrow{\text{ws02 sensed}}$ activate stopper
AU_3	$\xrightarrow{\text{ps01 sensed}}$ activate mixing_robot
AU_4	$\xrightarrow{\text{mixing\_robot activated}}$ if lli01 off: activate lifting_arm; open compound_valve01; open compound_valve02;
AU_5	$\xrightarrow{\text{ts01 on}}$ open water_valve
AU_6	$\xrightarrow{\text{qc01 passed}}$ if ts01 off: open drain_valve; deactivate mixing_-robot;
AU_7	$\xrightarrow{\text{mixing\_robot deactivated}}$ deactivate stopper
AU_8	$\xrightarrow{\text{ps02 sensed}}$ set processed;

The testbed production line is designed to receive a metal block at a time and deliver it to the furnace in order to be combined with two other compounds and then drain the mixture into another production line (not covered here) and start the whole process again for a new batch. In this periodic operation, each element in the system takes a fixed amount of time to perform its task and deliver its output to the next element. The design of the system guarantees that the overall time needed for each iteration is significantly longer than the total time needed by each element in isolation. Each iteration starts when a pallet carrying a metal block has been placed on a weight sensor `ws01`, which activates AU_1. The robot then lifts the pallet and places it on the conveyor belt. The conveyor belt's weight sensor `ws02` notifies the stopper in the middle-end of the line about the incoming pallet via AU_2. Once arrived, the stopper engages the pallet and detaches it from conveyor belt. The presence sensor `ps01` then detects package arrival and AU_3 activates the mixing robot. According to AU_4, the mixing robot then checks whether the liquid level indicator `lli01` inside the furnace is off and then activates the lifting arm to take the metal block off the pallet and puts it inside the furnace. It then opens the compound valves 1 and 2, respectively. As the result of some chemical process, the temperature of the mixture goes up until the temperature sensor `ts01` trips. In that case AU_5 opens the water valve to cool down the mixture. Once cooled, it reaches to the point making the quality control sensor `qc01` to indicate "passed" signal. Then, AU_6 gets executed and monitors `ts01` to double check whether it is safe to open the drain valve and reset the mixing robot. Deactivating the mixing robot then releases the stopper (AU_7) which puts the pallet back on the conveyor belt letting it to go towards the end of the line where presence sensor `ps02` resides. Once `ps02` tripped, it signals the batch process has completed (AU_8) and pallet will be removed automatically from the conveyor belt.

Although this production line works fine for most situations, there are some undesired cases that can cause fatal physical damages such as *conveyor belt blockage*, *liquid overflow*, and *ruining the batch*.

Scenario #1 - Conveyor belt blockage. There are several reasons that can cause multiple pallets to be on conveyor belt at the same time making the conveyor belt to break because of its weight tolerance limitation or crashing a few pallets into each other. Having multiple pallets at the same time can happen if the the previous pallet got stuck on an obstacle next to conveyor belt or malfunctioning stopper. Therefore, after passing the iteration time limit, our simulation has shown that multiple pallets can be placed on the conveyor belt causing the blockage. This undesirable situation has been prevented by PatrIoT using the policy in the Listing 1.1 which says "*deny activating the lifter robot only if since the last time it was activated, no batch has been processed*".

Listing 1.1. policy language Policy P1 to address conveyor belt blockage

```
POLICY P1:
  DENY action_command = on and action_device = g1/lifter_robot
  ONLY IF LASTLY(SINCE(state(unit:g1/lifter_robot) = on, value(g1/processed) = 0))
```

Scenario #2 - Liquid overflow. The production line is also susceptible to furnace overflow causing severe physical damages. According to our simulation, this undesirable situation can occur if, for example, the drain valve is faulty such that it cannot completely drain the previous batch and some significant amount of mixture liquid has remained in the furnace, which is still below the `lli01` threshold. Therefore, once the next batch comes in (including the metal block, compounds 1 and 2, and the water) it will cause liquid overflow. PatrIoT prevents such undesirable situation via the following policies which simply checks the liquid level before taking any actions causing adding a substance to the furnace. This scenario with policies (`P2-P5`) in place could still create a conveyor belt blockage which will be prevented by the policy `P1`.

Listing 1.2. policy language Policies to address liquid overflow

```
POLICY P2:
  DENY action_command = on and action_device = g1/lifting_arm
  ONLY IF value(sensor:g1/lli01) = on

POLICY P3:
  DENY action_command = on and action_device = g1/compound_valve01
  ONLY IF value(sensor:g1/lli01) = on

POLICY P4:
  DENY action_command = on and action_device = g1/compound_valve02
  ONLY IF value(sensor:g1/lli01) = on

POLICY P5:
  DENY action_command = on and action_device = g1/water_valve
  ONLY IF value(sensor:g1/lli01) = on
```

Scenario #3 - Ruining the batch. This scenario is based on the requirement which warrants that water will be added to the mixture only after compounds 1 and 2 have been poured. If the metal block is placed in the furnace containing water, that batch will be considered as ruined. This situation can happen in our testbed when for some reason (*e.g.*, `ts01`) water valve opens before mixing robot is activated in an iteration. PatrIoT uses the following policy to avoid this situation. The policy "*allows activating the mixing robot only if since the last time it was activated, water valve has not been opened*".

Listing 1.3. policy language Policy P1 to address conveyor belt blockage

```
POLICY P6:
  ALLOW action_command = on and action_device = g1/mixing_robot
  ONLY IF LASTLY(SINCE(state(unit:g1/mixing_robot) = off, state(unit:g1/water_valve) =
        off))
```

SmartThings and OpenHAB

Testbeds. To perform our evaluation on the smart-home platforms we built two testbeds for SmartThings and OpenHAB in which we leveraged 48 IoT devices for our setup [6].

In order to setup the testbed for OpenHAB, we deployed OpenHAB 2.4 [5] on a Raspberry Pi 3 Model B+ and created our virtual devices inside the platform. Virtual devices in OpenHAB can be controlled and monitored via a web-based interfaced provided by the platform. For SmartThings, we used SmartThings web-based IDE to create these virtual devices. In order to control and monitor the status of these devices, SmartThings provides a companion mobile app, so-called SmartThings, which we used for this experiment.

IoT Apps. For our experiment, we used 122 SmartApps for SmartThings and 20 rules for OpenHAB which we collected from SmartThings official repository [8], IoTBench [16], and the developer community forums. All rules have been manually investigated to establish the ground-truth and understand their semantics and intentions. Unlike static analysis approaches, PatrIoT provides runtime protection; therefore, to evaluate its effectiveness, all these rules need to be executed in different scenarios and through the pre-established ground-truth one should validate whether PatrIoT can maintain user-specified policies.

Policies. In our experiment, we used 33 policies which are listed in Table 3. These policies are specified after studying the literature and acquiring the necessary knowledge by manually investigating the rules. Although the English description of these policies are provided for exposition, the policy language representation of these policies can be found in [6]. Table 3 also shows the main goal of each policy. For instance, **P1** restricts sending SMS by the rules to only those that the user expects to do so. In that case, all other apps trying to send SMS, whether maliciously or not, will be blocked by the policy. This policy aims to protect the user against any privacy violation. Policy **P27** also protects the expensive appliances (*e.g.*, water pump) from any damage that might be caused by repeatedly turning them on and off which can be as a result of a loop of actions because of a semantic bug or a malicious intent.

In order to evaluate the effectiveness of PatrIoT we created three sample scenarios to illustrate that the specified policies are maintained by the system at runtime.

- *Privacy.* In this scenario, the main focus is on Policies **P1** and **P2**. Given these policies, PatrIoT only allowed user's authorized apps to send SMS while denying the action for other apps. For instance, a malicious app pretending to strobe the alarm when CO2 is detected while maliciously sending an advertisement before strobing the alarm. However, PatrIoT is able to successfully block that advertisement by sms at runtime.
- *Overprivilege.* This scenario mainly focuses on Policies such as **P3** and **P14** and PatrIoT only allowed the authorized apps to unlock the door and open the garage. Those malicious apps such as the one that monitors the battery level of the lock but sneakily detects that nobody is at home and then unlocks the door, are successfully blocked.
- *Interplay.* The focus of this scenarios is on Policies that can protect users from some hidden- unwanted actions such as **P5**. Based on fire-sprinkler app and dry-the-wetspot app, actions might interfere and as a result water valve gets closed while there is still fire to contain. Given the policy PatrIoT successfully blocked unwanted closing of water valve.

6.2 Efficiency

In order to measure the runtime overhead of PatrIoT incurred on EVA ICS, SmartThings, and OpenHAB, we calculate the computation time of executing each automation unit *with* and *without* PatrIoT in place and then compare them to each other. Figure 3 illustrates the runtime overhead incurred by PatrIoT in different platforms, which is on average 8.96%, 9.44%, and 11.52% for EVA ICS, OpenHAB, and SmartThings, respectively. This runtime overhead depends on: *(i)* the number of actions happening at the same time; *(ii)* the complexity of the policies related to an impending action; and (*iii)* synchronization mechanism support for a platform. To have a fair evaluation, we carefully established our testbeds to closely reflect real-world IoT setups. For the application we had, although this overhead is acceptable, this could be an interesting future work to discover tighter overhead threshold for different applications. Figure 4 also shows the portion of the PatrIoT overhead incurred by locking mechanism. Among these platforms, SmartThings has the most locking overhead since it uses our external https-based lock manager. The locking overhead can be reduced, if we have native synchronization support form SmartThings.

7 Related Work

Prior efforts in IoT security are broadly focused on devices [33,27,31,38,10,23], communication and authentication protocols [34,42,25,43]. There are significant efforts focusing on unexpected behavior on programmable IoT systems [24,20,32,26,37,21,28,30,9,40,17,19,18,29,44], which is also the focus of this paper. Broadly speaking, there are two main approaches to address this issue: static and runtime monitoring approaches. The static approaches [24,37,20,17,30] are mostly pre-deployment techniques used for either: *(i)* further investigation like

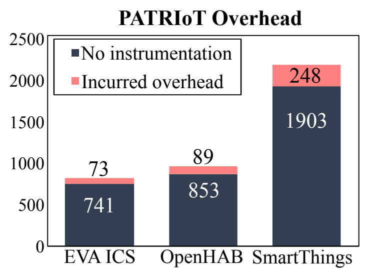

Fig. 3. Runtime overhead (in milliseconds) incurred by PatrIoT because of the instrumentation.

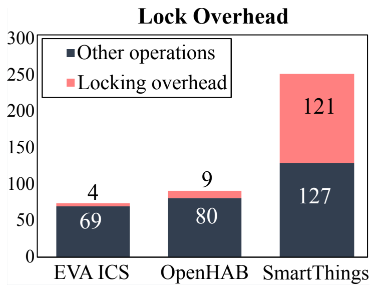

Fig. 4. Runtime overhead (in milliseconds) incurred by PatrIoT's synchronization mechanism.

taint analysis to find out how private data is consumed by the apps (*i.e.*, whether or not there is any unwanted operation performed on the data); *(ii)* verifying that it satisfies a set of properties described as its correct behavior; or *(iii)* rectification aiming to fix mistakes in writing the trigger-action rules in it. Runtime monitoring based approaches, on the other hand, aim to provide post-deployment solutions to prevent unsafe/undesired operations at runtime [28,41,18]. These prior efforts, however, either require human intervention or cannot support rich temporal policies. Extensive work has been done in developing efficient runtime monitors using different types of logic [12,13,14,22,35,36,15]. Prior works on IoT security, however, have not leveraged these rich policy languages and do not take advantage of the developments in the runtime verification community. Apart from these works, there have been efforts repairing or synthesizing rules based on given properties [44] to make sure IoT apps behave as expected.

8 Discussion

PatrIoT policy expressiveness. In contrast to existing IoT policy languages [28,41,18], PatrIoT policies have a formal semantics, are more expressive, and can specify existing IoT policies in the literature. With respect to MFOTL, however, PatrIoT policies neither support quantification nor arbitrary function symbols. Although such restrictions consciously limit the expressive power of the language to applicable to general policies, they are not only necessary for efficient monitoring but also sufficient to express existing IoT policies.
Authoring PatrIoT policies. For PatrIoT's effectiveness, it is crucial for the users to be able to write the correct and consistent policies. To be practically deployable, one would require to consider the usability of the language as well as tool support for identifying inconsistent policies. To limit the scope of this paper, we focus on the technical foundations of PatrIoT and considering deployment issues are subjects of future work.
Extending PatrIoT applicability. Current instantiation of PatrIoT assumes a centralized IoT architecture. However, one can envision extending PatrIoT for

decentralized IoT architectures. For this extension, however, one would require the decomposition of global policies into local policies that are to be enforced on the IoT devices themselves. Additionally, the device-centric policy enforcement mechanisms would need to communicate to ensure the consistency of the global system and policy states. This is a subject of future work.
Performance overhead of PatrIoT. We observed that PatrIoT on-average incurs < 100 milliseconds of runtime overhead in systems whose app programming interface have native support for synchronization mechanism (e.g., mutex). For Samsung SmartThings, PatrIoT on-average induces an overhead of 248 milliseconds of which 48% is due to its web-based locking mechanism. Such overheads are tolerable in a non-safety-critical system such as a home automation system. For real-time systems, however, this overhead needs to be decreased. One possible solution to this high overhead is to realize PatrIoT by incorporating it in the back-end.
Limitation. Currently, PatrIoT can only regulate actions contemplated by IoT apps. Actions triggered by third-party service (e.g., IFTTT) or user interaction with the companion mobile app cannot be regulated by PatrIoT. To mitigate this limitation, one would require installing PatrIoT in the backend. Installing PatrIoT in the backend may require app instrumentation for collecting context information. Extending PatrIoT with such support is future work.

9 Conclusion

We presented PatrIoT, a runtime monitor that dynamically ensures that actions performed by IoT apps installed in an IoT system do not violate desired policies crucial for assuring the security, privacy, and safety of the users and system. To express policies, PatrIoT provides a platform-independent policy language policy language that can effectively capture system invariants as well as different temporal behaviors including explicit timing restrictions and counting operator. For compliance checking, PatrIoT uses an existing, efficient dynamic programming algorithm which encodes the relevant information in the system's execution history into summary structures that can be quickly looked up during policy checking. Finally, we evaluated PatrIoT's generality, efficacy, and efficiency by instantiating it for three popular open-source IoT platforms We tested 33 policies against 122 SmartThings, 10 policies against 20 OpenHAB, and 6 policies against 8 EVA ICS automation units. The performance overhead induced by PatrIoT is as low as 248 ms for SmartThings, 89 ms for OpenHAB, and 73 ms for EVA ICS, demonstrating the efficiency of our proposed framework.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments and suggestions. This work was supported by DARPA CASE program award N66001-18-C-4006. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or DARPA.

Table 3. List of policies used in the evaluation

ID	Policy description	Main goal
P1	Allow sending SMS only if it is requested by flood-alert or energy-alerts or humidity-alert or mail-arrived or medicine-reminder or presence-change-text or ready-for-rain or laundry-monitor apps.	Privacy
P2	Deny all http request.	Privacy, Security
P3	Allow unlocking the front door only if it is requested by enhanced-auto-lock-door app.	Security
P4	Allow water valve to be closed only if the fire-sprinkler was not on in the last 5 hours.	Safety
P5	Allow water valve to be closed only if water leak sensor sensed wet within 1 minute.	Safety
P6	Allow any light to be turned on only if the system is not on vacation mode.	Safety, Energy saving
P7	Deny surveillance camera to get turned off except user is at home	Security
P8	Allow light to be switched on only if user is at home.	Safety, Energy saving
P9	Allow hallway light to get turned on only if the hallway motion sensor has tripped within 20 sec.	Energy saving
P10	Allow siren to go off only if lastly smoke/co2 detector detects smoke/co2 with 1 minute, or flood sensor sensed wet within 1 minute, or motion is sensed within 1 minute while user is not at home.	Convenience
P11	Deny turning on the coffee machine only if the user is not at home.	Safety
P12	Deny turning off the refrigerator or TV only if it is requested by energy-saver app.	Convenience
P13	Allow light to be turned off only if lastly it was on within 30 sec.	Damage protection
P14	Deny opening the garage door except it is requested by garage-door-opener app.	Security
P15	Deny unlock the front door only if user is not at home or the system is in sleep mode except smoke detector detects smoke within the last 60 sec.	Security, Safety
P16	Allow AC to be switched on only if the heater is off.	Convenience, Energy saving
P17	Allow heater to be switched on only if the AC is off.	Convenience, Energy saving
P18	Allow living room window to be opened only if both heater and AC are off.	Energy saving
P19	Allow living room Window to be opened only if lastly motion detector tripped within 60 sec.	Energy saving
P20	Allow living room window to be closed only if lastly it was open within 30 sec.	Damage protection
P21	Allow opening the interior door only if the exterior door is closed and lastly pod is empty.	Security
P22	Deny irrigation to go off only if since moisture sensor sensed wet, it has been dry within 2 days.	Energy saving
P23	Allow setting the mode to away only if my presence is not present.	Security, Safety
P24	Allow setting the mode to home only if my presence is present.	Security, Safety
P25	Allow turning off coffee machine only if lastly it was on within 30 sec.	Damage protection
P26	Allow turning off TV only if lastly it was on within 30 sec.	Damage protection
P27	Allow turning off water pump only if lastly it was on within 120 sec.	Damage protection
P28	Allow turning off water pump only if the basement moisture sensor senses dry.	Safety
P29	Allow turning off thermostat only if lastly it was on within 120 sec.	Damage protection
P30	Allow changing light level only if user is at home.	Security, Energy saving
P31	Deny turning on TV only if it is after midnight.	Energy saving
P32	Allow window to be opened only if the system is not on vacation mode.	Safety, Energy saving
P33	Allow window to be opened only if user is at home.	Safety, Energy saving

References

1. Antlr. `https://www.antlr.org`, accessed: Feb 16, 2019
2. Apache Groovy – runtime and compile-time metaprogramming. `http://groovy-lang.org/metaprogramming.html`, accessed: Sep 13, 2019
3. CWE-367 – time-of-check time-of-use (toctou) race condition. `https://cwe.mitre.org/data/definitions/367.html`, accessed: Sep 13, 2019
4. EVA ICS. `https://www.eva-ics.com`, accessed: Sep 13, 2019
5. openHAB – a vendor and technology agnostic open source automation software for your home. `https://www.openhab.org`, accessed: Feb 16, 2019
6. PatrIoT. `https://github.com/yahyazadeh/patriot.git`, accessed: Aug 16, 2020
7. Smartthings. `https://www.smartthings.com/`, accessed: Feb 16, 2019
8. SmartThings Public GitHub Repo. `https://github.com/SmartThingsCommunity/SmartThingsPublic`, accessed: Feb 17, 2019
9. Alrawi, O., Lever, C., Antonakakis, M., Monrose, F.: Sok: Security evaluation of home-based iot deployments. In: (S&P). IEEE (2019)
10. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al.: Understanding the mirai botnet. In: USENIX Security Symposium. pp. 1092–1110 (2017)
11. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. (FMSD) **46**(3), 262–285 (2015)
12. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: (CAV). pp. 1–18. Springer (2010)
13. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. (JACM) **62**(2), 1–45 (2015)
14. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. (TOSEM) **20**(4), 1–64 (2011)
15. Calzavara, S., Focardi, R., Maffei, M., Schneidewind, C., Squarcina, M., Tempesta, M.: {WPSE}: Fortifying web protocols via browser-side security monitoring. In: USENIX Security Symposium. pp. 1493–1510 (2018)
16. Celik, Z.B., Babun, L., Sikder, A.K., Aksu, H., Tan, G., McDaniel, P., Uluagac, A.S.: Sensitive information tracking in commodity iot. In: USENIX Security Symposium. pp. 1687–1704 (2018)
17. Celik, Z.B., McDaniel, P., Tan, G.: Soteria: Automated iot safety and security analysis. In: (ATC). pp. 147–158. USENIX (2018)
18. Celik, Z.B., Tan, G., McDaniel, P.: IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In: (NDSS) (2019)
19. Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W.C., Sun, M., Yang, R., Zhang, K.: Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In: (NDSS) (2018)
20. Chi, H., Zeng, Q., Du, X., Yu, J.: Cross-app interference threats in smart homes: Categorization, detection and handling. CoRR **abs/1808.02125** (2018)
21. Ding, W., Hu, H.: On the safety of iot device physical interaction control. In: (CCS). pp. 832–846. ACM (2018)
22. Du, X., Liu, Y., Tiu, A.: Trace-length independent runtime monitoring of quantitative policies in ltl. In: (FM). pp. 231–247. Springer (2015)
23. Edwards, S., Profetis, I.: Hajime: Analysis of a decentralized internet worm for iot devices. Rapidity Networks **16** (2016)
24. Fernandes, E., Jung, J., Prakash, A.: Security analysis of emerging smart home applications. In: (S&P). vol. 00, pp. 636–654. IEEE (May 2016)

25. Gong, N.Z., Ozen, A., Wu, Y., Cao, X., Shin, R., Song, D., Jin, H., Bao, X.: Piano: Proximity-based user authentication on voice-powered internet-of-things devices. In: (ICDCS). pp. 2212–2219. IEEE (2017)
26. He, W., Golla, M., Padhi, R., Ofek, J., Dürmuth, M., Fernandes, E., Ur, B.: Rethinking access control and authentication for the home internet of things (iot). In: (USENIX Security). pp. 255–272 (2018)
27. Ho, G., Leung, D., Mishra, P., Hosseini, A., Song, D., Wagner, D.: Smart locks: Lessons for securing commodity internet of things devices. In: (ASIACCS). pp. 461–472. ACM (2016)
28. Jia, Y.J., Chen, Q.A., Wang, S., Rahmati, A., Fernandes, E., Mao, Z.M., Prakash, A.: ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In: (NDSS) (2017)
29. Lee, S., Choi, J., Kim, J., Cho, B., Lee, S., Kim, H., Kim, J.: Fact: Functionality-centric access control system for iot programming frameworks. In: (SACMAT). pp. 43–54. ACM (2017)
30. Nguyen, D.T., Song, C., Qian, Z., Krishnamurthy, S.V., Colbert, E.J., McDaniel, P.: Iotsan: fortifying the safety of iot systems. In: (CoNEXT). pp. 191–203. ACM (2018)
31. Notra, S., Siddiqi, M., Gharakheili, H.H., Sivaraman, V., Boreli, R.: An experimental study of security and privacy risks with emerging household appliances. In: (CNS). pp. 79–84. IEEE (2014)
32. Rahmati, A., Fernandes, E., Eykholt, K., Prakash, A.: Tyche: A risk-based permission model for smart homes. In: (SecDev). pp. 29–36. IEEE (2018)
33. Ronen, E., Shamir, A.: Extended functionality attacks on iot devices: The case of smart lights. In: (EuroS&P). pp. 3–12. IEEE (2016)
34. Ronen, E., Shamir, A., Weingarten, A.O., OFlynn, C.: Iot goes nuclear: Creating a zigbee chain reaction. In: (S&P) (2017)
35. Rosu, G., Havelund, K.: Synthesizing dynamic programming algorithms from linear temporal logic formulae (2001)
36. Soewito, B., Vespa, L., Mahajan, A., Weng, N., Wang, H.: Self-addressable memory-based fsm: a scalable intrusion detection engine. IEEE network **23**(1), 14–21 (2009)
37. Tian, Y., Zhang, N., Lin, Y.H., Wang, X., Ur, B., Guo, X., Tague, P.: Smartauth: User-centered authorization for the internet of things. In: (USENIX Security) (2017)
38. Ur, B., Jung, J., Schechter, S.: The current state of access control for smart devices in homes. In: (HUPS) (2013)
39. Wang, Q., Datta, P., Yang, W., Liu, S., Bates, A., Gunter, C.A.: Charting the atack surface of trigger-action iot platforms. In: (CCS) (2019)
40. Wang, Q., Hassan, W.U., Bates, A., Gunter, C.: Fear and logging in the internet of things. In: ISOC NDSS (2018)
41. Yahyazadeh, M., Podder, P., Hoque, E., Chowdhury, O.: Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In: (SACMAT). pp. 61–72. ACM (2019)
42. Yu, T., Sekar, V., Seshan, S., Agarwal, Y., Xu, C.: Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In: (HotNets) (2015)
43. Zhang, J., Wang, Z., Yang, Z., Zhang, Q.: Proximity based iot device authentication. In: (INFOCOM). pp. 1–9. IEEE (2017)
44. Zhang, L., He, W., Martinez, J., Brackenbury, N., Lu, S., Ur, B.: Autotap: Synthesizing and repairing trigger-action programs using ltl properties. In: ICSE (2019)