IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT

Z. Berkay Celik, Gang Tan, and Patrick McDaniel
Pennsylvania State University
{zbc102, gtan, mcdaniel}@cse.psu.edu

***Abstract*—Broadly defined as the Internet of Things (IoT), the growth of commodity devices that integrate physical processes with digital connectivity has changed the way we live, play, and work. To date, the traditional approach to securing IoT has treated devices individually. However, in practice, it has been recently shown that the interactions among devices are often the real cause of safety and security violations. In this paper, we present IoTGuard, a dynamic, policy-based enforcement system for IoT, which protects users from unsafe and insecure device states by monitoring the behavior of IoT and trigger-action platform apps. IoTGuard operates in three phases: (a) implementation of a code instrumentor that adds extra logic to an app's source code to collect app's information at runtime, (b) storing the apps' information in a dynamic model that represents the runtime execution behavior of apps, and (c) identifying IoT safety and security policies, and enforcing relevant policies on the dynamic model of individual apps or sets of interacting apps. We demonstrate IoTGuard on 20 flawed apps and find that IoTGuard correctly enforces 12 of the 12 policy violations. In addition, we evaluate IoTGuard on 35 SmartThings IoT and 30 IFTTT trigger-action platform market apps executed in a simulated smart home. IoTGuard enforces 11 unique policies and blocks 16 states in six (17.1%) SmartThings and five (16.6%) IFTTT apps. IoTGuard imposes only 17.3% runtime overhead on an app and 19.8% for five interacting apps. Through this effort, we introduce a rigorously grounded system for enforcing correct operation of IoT devices through systematically identified IoT policies, demonstrating the effectiveness and value of monitoring IoT apps with tools such as IoTGuard.**

I. Introduction

IoT devices used in smart homes, industrial automation, agriculture, and transportation have become a fundamental part of modern society. Such devices enable our living space to be more autonomous, adaptive, efficient, and convenient. However, concerns have also been raised about the security and privacy of these digitally augmented spaces [9]–[11], [16], [17]. These environments necessarily have access to functions that if abused would put the user security at risk. e.g., unlocking doors when users are not at home, or creating unsafe or damaging conditions by turning off the heat at winter. Recently, it has been shown that the interactions between devices are an increasing cause of safety and security violations [11], [12], [14], [39]. In practice, IoT apps interact through a common device or some common abstract event (such as the home, away or sleeping modes) when they are co-installed in an environment. These interactions lead to unsafe and undesired device states through apps' joint behavior. For example, an app that opens the water valve to activate fire sprinklers when there is a fire interacts with another app that shuts off the water valve when it detects water leaks. In this case, the joint behavior of the otherwise-safe apps leaves users at risk from fire.

Another trend is that increasingly trigger-action platforms such as IFTTT [26], Zapier [53], and Microsoft Flow [35] are used to bridge the divide between physical (e.g., IoT devices) and digital (e.g., e-mail services and social media platforms) processes. These platforms allow users to write rules that connect the events and actions of IoT devices with the events and actions of digital services. For example, a rule turns on the light when the user receives an email, and similarly, another rule logs the user's presence to a spreadsheet file when the front door is unlocked. This inter-tangled environment expands the interactions among devices to online services [10], [47]; for example, an IoT app that subscribes to the switch "turn-on" event interacts with a trigger-action platform rule that "turns on" the switch when the user is tagged in a photo on Facebook.

Most attempts to date in IoT security and privacy aim to improve perimeter defenses that harden the IoT infrastructure against attacks using firewalls [30], intrusion detection [54], access control policies [22], and software patches [32]. Yet, perimeter security measures do not enforce safe behavior of physical processes in IoT systems. For example, a firewall rule does little to guarantee that the door is locked when the user is not home. Furthermore, past analyses of IoT devices and environments have focused on securing an IoT app through source code analysis. For instance, some systems infer an app's context to enforce permissions based on that context through runtime prompts [28] or asking users for authorization through an interface [49], and others apply static model checking to find property violations [11]. Unfortunately, current dynamic approaches are insufficient to identify and ultimately enforce violations in multi-app environments, and static approaches lack precision and enforce only a limited set of policies.

In this paper, we present IoTGuard, a dynamic enforcement system for the usage of the most sensitive resource in an IoT system, the physical devices themselves. IoTGuard directly blocks unsafe and undesired states in an individual app and multi-app environments. To achieve this, an app is instrumented with an assertion of the code blocks to work with IoTGuard. Here, IoTGuard models the app's lifecycle and adds code to obtain an app's events, actions, and predicates that guard each action. The instrumented app then executes when a subscribed event occurs. The app transmits its information (e.g., events

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23326
www.ndss-symposium.org

and actions) to IoTGuard before it executes actions. The apps information is stored in a dynamic model that consists of transitions and states. The dynamic model represents the runtime execution behavior of an individual app if an app does not interact with other apps, and the unified behavior of the apps when the apps interact. From this, IoTGuard evaluates the (unified) dynamic model of an app against a set of systematically developed IoT policies. A policy is a system artifact that represents the physical behavioral specifications of users' expectations about the safe and secure behavior of an IoT system. If an app's action fails to pass a policy, IoTGuard enforces the policy violation by notifying an app with a reject message; otherwise a pass message. The instrumented app's action is conditioned on the security service's response; thus an app's actions that violate a policy are blocked or allowed depending on the response.

We present two studies evaluating IoTGuard. In a first study, we evaluated the effectiveness of IoTGuard on 15 SmartThings IoT apps and five IFTTT trigger-action platform apps. These apps include a flaw or malicious behavior that violates policies when used in isolation and when used together in multi-app environments. IoTGuard correctly identified all policy violations. The second study is a horizontal market study in which we evaluated 35 SmartThings and 30 IFTTT market vetted apps in a simulated smart home, which includes 29 devices with a total of 20 device types. IoTGuard enforced eleven unique policies in five SmartThings and six IFTTT apps. The experiments also demonstrated that IoTGuard enforces policies without significant overhead; it incurs only 17.3% runtime overhead on an individual app and 19.8% for five apps interacting with each other. In summary, we make the following contributions:

- We introduce IoTGuard, a dynamic system for policy enforcement on IoT devices. IoTGuard adds extra logic to an app's source code to collect its information in a dynamic model and enforces safety and security policies in an app and multi-app environments.
- We validate IoTGuard on a corpus of 20 hand-crafted flawed apps (15 SmartThings and five IFTTT apps) and expose safety and security violations in an app and interacting apps. Furthermore, we evaluate IoTGuard on 65 market-vetted apps (35 SmartThings and 30 IFTTT apps) executed in a simulated smart home and reveal how violations are enforced.
- We evaluate performance of IoTGuard on SmartThings and IFTTT apps, showing that policy enforcement incurs on average a runtime overhead of 17.3% for an individual app and 19.8% for five interacting apps.

II. Background

A. IoT Platforms

IoT systems integrate physical processes with digital connectivity. Regardless of their purpose and complexity, IoT systems often structure their architecture from bottom to top with devices, connectivity protocols, and IoT programming platforms. Such systems often use a hub as a centralized gateway connecting devices in a physical environment, use the cloud to synchronize device states and provide interfaces for remote control and monitoring. The devices are equipped with embedded sensors and actuators that enable interaction with a physical environment. Sensors collect physical states and send events to other devices, the hub, or the cloud. Events are processed and used to actuate the devices. For example, a presence sensor detects a presence event and communicates with the light switch (actuator) that turns on the lights. We note that a mobile phone or even a coffee machine can be a sensor as long as it can sense its environment. Protocols are used to establish communication between heterogeneous devices and network endpoints. These protocols are selected according to the constraints of environments such as low power or lossless connection. IoT programming platforms are responsible for delivering app-specific services by managing devices and their interactions. They also enable crucial functions such as data collection, control, and interoperability. In recent years, several IoT programming platforms have emerged in a wide range of domains: Apple's HomeKit [5], OpenHAB [41], Samsung's SmartThings [43] for smart home, Android Sensor API [2], Google Fit for wearables [20], ThingWorx [42] for industrial applications, and FarmBeats [50] for agriculture. These platforms offer web-based environments and tools that enable developers to write applications through various APIs.

B. Trigger-Action Platforms

Trigger-action platforms such as IFTTT [26], Zapier [53] and Apiant [4] allow users to connect services together. A service includes a set of APIs on a trigger-action platform. Users authorize services to their trigger-action platform accounts. For example, a user with a SmartThings IoT platform account can authorize the SmartThings service through the OAuth protocol to communicate with her SmartThings account. Services communicate with each other using REST APIs over HTTP(S) [6], [18]. Trigger-action platforms allow users to create custom automation on services through DO and IF rules. These rules let users connect a trigger in a service to take the desired action in another service—when an event happens in a service, the platform automatically triggers a separate action in another service. DO rules acts as a virtual button trigger to take a set of actions; for example, a DO rule may turn on a smart switch of a user when a button is tapped. IF rules combine two services using a trigger and an action; for example, an IF Rule may make a phone call to the security guard when a motion sensor of a smart home service detects motion after midnight. Users are required to install a companion app provided by the trigger-action platform to trigger DO rules. IF rules run automatically after users configure them via a trigger-action platform web API. As of May of 2018, IFTTT has the largest market share in trigger-action platforms [52]; it provides users with 500 services, 158 of which are IoT services. IFTTT IoT services fall into different categories such as wearables, fitness and health devices, home devices, and monitoring systems.

C. Definitions

We adopt a general terminology that describes actions, events, services, and states in IoT apps and trigger-action rules. A device has a set of *attributes*, which are the states of the device. *Actions* of a device can change attributes. For example, the door may have opening, opened, closing and closed attributes and only open and close actions. *Events* are triggered when there is a change to device states. An app

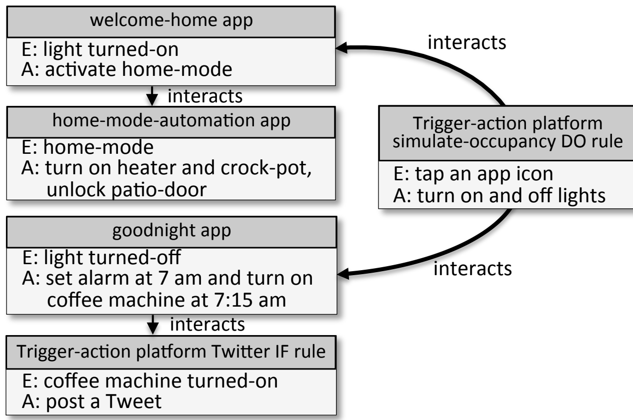

Fig. 1: Events (E) and Actions (A) of IoT apps and trigger-action platform rules, and their interactions with each other.

subscribes to some event and takes actions when that event happens. For instance, an app subscribes to the boolean attribute of a motion detector's "motion-active" event and changes the state of a switch to "switch-on". Trigger-action platforms connect events and actions of different online services. Events, in a trigger-action platform, are the state changes in a service, and actions are the functions that are initiated as a result of the event. For instance, a trigger-action rule may invoke the "post a Tweet" action of Twitter when the "coffee machine-turned-on" event is triggered in an IoT platform.

III. Motivation and Assumptions

Problem Statement. The interaction among IoT devices is an increasing cause of unsafe and insecure states [11]. To illustrate, we consider a scenario where there are three IoT apps and two trigger-action rules in a shared environment, as shown in Figure 1. A welcome-home app sets the mode to home when the light in the living room is turned on. A home-mode-automation app turns on the heater and crock-pot and unlocks the patio door when home-mode is activated, and a good-night app sets the alarm and brews coffee at a time defined by the user when the light is turned off. A Twitter IF rule posts a tweet of "Good morning, what a beautiful day in Palo Alto!" when the coffee machine is turned on, and a simulate-occupancy DO rule simulates the occupancy in a home at night by turning on and off lights when the user clicks on a button in an app or at specific times defined by the user.

Joint behavior of otherwise-safe apps may leave the user in unsafe and insecure states. To illustrate, turning on the switch in the simulate-occupancy rule interacts with the welcome-home app, and the welcome-home app interacts with the home-mode-automation app through the home-mode event. Turning off the switch in the simulate-occupancy app interacts with the good-night app. Turning on the coffee machine in the good-night app interacts with the Twitter IF rule. The interaction among these apps can turn on the heater, crock-pot, and coffee machine, unlock the patio-door, set the alarm, and post a tweet on Twitter. The resulting states may put the user at risk or cause embarrassment or other harms; e.g., the heater is turned on, and the door is unlocked when the user is not home, or post a public tweet when the user is on vacation.

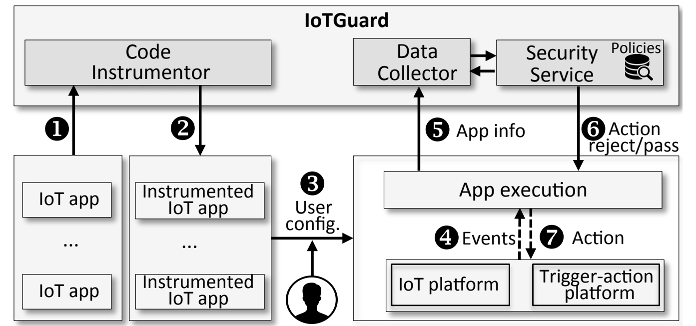

Fig. 2: Architecture of the IoTGuard system.

Definition of Interactions. Apps interact through a common device or abstract events. For our purposes, we use the term apps to refer to both IoT apps and trigger-action rules. Two apps interact with each other, (1) when an event handler of an app changes a device attribute, which triggers another event that is subscribed to by another app; for example, an app turns on the light when there is smoke, and another app unlocks the door when the light is turned on, (2) when multiple apps change the same device attribute of some device; for example, a water-leak-detector app shuts off the water valve when there is a leak, while a smoke-alarm app opens the water valve to activate the sprinkler, and (3) when apps that subscribe to the same event change a device attribute in conflicting ways; for example, when the motion is active, an app turns on a switch while another app turns off the switch. These interactions among devices may cause security, safety, and privacy risks even though individual apps are safe in operation (See Section V-C). We found that apps also interact through *modes*, which are behavior filters that automate device actions. For instance, an app that changes the "home" mode to "away" mode when a user leaves home interacts with an app that uses the "mode change" event to unlock the door. Lastly, we define the interaction size of an initial event as the number of apps whose event handlers get executed, either directly triggered by the initial event or indirectly triggered (since event handlers may cause attribute changes, generating more events along the process).

Threat Model. We consider integrity and confidentiality violations caused by flaws in apps or malicious apps in an IoT environment. For malicious cases, integrity violations occur when the adversary inserts malicious code to an app or provides a user with an app that can cause an unsafe or insecure state; confidentiality violations happen when private information in an IoT system becomes publicly available in an online service. For instance, a user's presence state is saved to a public file when the user leaves home. We do not consider adversaries' ability to thwart security measures (e.g., crypto, forged inputs) of IoT and trigger-action platforms. We assume IoTGuard is tamperproof, and device owners are trusted.

IV. Approach Overview

IoTGuard is a dynamic, policy-based behavioral enforcement system for IoT, which protects users from unsafe and insecure device states by monitoring the behavior of IoT apps (See Figure 2). IoTGuard acts as a conduit between IoT apps and devices and could be implemented in several ways, such as in hub software, as a software service in the cloud, or in a

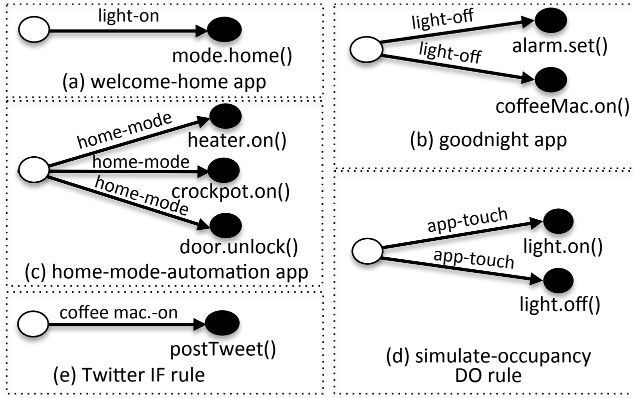

Fig. 3: Dynamic models of apps depicted in Figure 1.

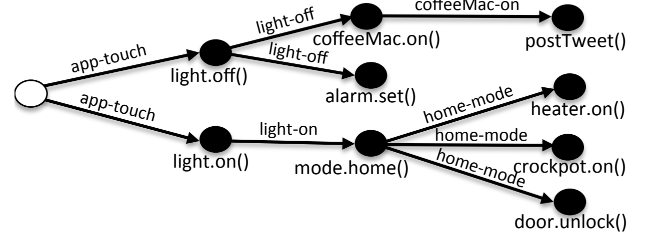

Fig. 4: The unified dynamic model of the apps in Figure 3.

local server. We implemented our prototype on a local server. Compared to a hub-based implementation, our prototype does not require modifying the hub, which is often closed source. Compared to a cloud-based implementation, a local-server implementation eliminates the need to trust cloud providers, while still providing complete mediation of app behavior.

IoTGuard checks an app's events and actions against a set of policies when the app receives an event and attempts to invoke actions. The policies are templates of safety and security properties. For example, a policy, user-not-present–appliances-off and doors-locked, requires the door is locked, and appliances are off when the user is not at home. An app is authorized to execute device actions if all policies are passed. The IoTGuard system includes three components: (a) a code instrumentor, (b) a data collector, and (c) a security service.

The code instrumentor instruments an app's source code to work with IoTGuard. It patches an app with code that collects an app's events, actions, and predicates that guard the actions at runtime. To do so, it first models an app's lifecycle before an app is submitted for execution (❶). It then adds instructions necessary for obtaining the app's information at runtime (❷). A user installs an instrumented app and configures the app's settings (e.g., the threshold value required for energy consumption) through the app's configuration interface (❸).

An attribute change on a device generates an event, which triggers an event handler method of an app if the app subscribes to that event (❹). When the app receives the event, the event and corresponding actions and the predicates that guard the actions are transmitted to the data collector through instructions added to the instrumented app (❺). The data collector stores this information in the form of a dynamic model. The dynamic model represents the runtime execution behavior of apps observed so far; it consists of states and state transitions. Turning to the apps in Figure 1, the dynamic model of apps after they get executed is presented in Figure 3. The data collector merges the dynamic models of apps if apps interact through a common device or an abstract event. Figure 4 shows the unified dynamic model of three IoT apps and two trigger-action rules.

When the data collector receives an event and its corresponding actions at runtime, the security service evaluates them against a collection of IoT safety and security policies. These policies are adapted from use/misuse case requirements engineering that addresses the real-world needs of users and environments, and many of them were thoroughly exercised on the source code of IoT apps through a model checker [11]. The policies are checked on the dynamic model of an app (if an app is independent of other apps) or on the unified dynamic model (if an app interacts with other apps) by means of reachability analysis. Based on user needs, the security service adopts two solutions to enforce the policies. First, the instrumented app guards each action with a predicate conditioned on the security service's response. If an action fails to pass a policy, the security service rejects the action; otherwise, the action is executed (❻). Therefore, an app's actions that violate a policy are blocked or allowed based on the response from the security service (❼). Turning to the example apps, IoTGuard finds a violation of the user-not-present–appliances-off and doors-locked policy. The interactions lead to the state of door-unlock and appliances-on when the simulate-occupancy app triggers actions through the app-touch event; thus these actions are blocked, and the user is notified. The second solution is to present users an interface for approval of each policy violation through runtime prompts. For instance, when the light is turned on by simulate-occupancy app, the door-unlock() action requires user approval to be executed. This allows the user to be aware of policy violations, and reject or accept them. Clearly, this option is less secure for users who install apps without understanding warnings. This paper focuses specifically on identifying potentially harmful device states, blocking the action that violates a policy, and building a user interface for presenting policy violations.

V. IoTGuard

Implementing IoTGuard requires addressing several system challenges that include: implementing a code instrumentation tool to characterize the app states and transitions (Section V-A), storing each app's runtime information in an efficient dynamic model (Section V-B), identifying a set of security and safety policies, and enforcing these policies on the (unified) dynamic model of apps at runtime (Section V-C).

A. Code Instrumentor

The code instrumentor adds extra logic to an app's source code to collect its four type of information at runtime: (1) devices and events, (2) actions invoked for each event in the event handlers, (3) predicates that guards device actions (IoT apps may change device states conditionally, for example, an app may turn off a switch when the energy consumption is above some threshold and turn on the switch when the energy consumption is below another threshold. As a result, those device changes only occur when the predicates in the conditional branches hold), and (4) numerical-valued attributes

```
1: // Devices
2: presence_sensor ps
3: light_switch s
4: door d
5: thermostat t
6: power_meter p
7: // User inputs
8: t_away
9: thold
```

```
10: when ps.not-present
11:   s.off(); d.lock();
12:   t.set(t_away);
13: when ps.present
14:   t_home=71; d_thold=5;
15:   s.on(); d.unlock();
16:   if (p.power<thold+d_thold){
17:     t.set(t_home);
18:   }
```

Fig. 5: An example code block for illustrating the code instrumentation logic of IoTGuard.

of the device actions (some devices require a numerical value for invoking the actions, for example, a thermostat requires a discrete numerical-valued attribute for setting the temperature heating point). The instrumented app transmits the information to IoTGuard's data collector when the app receives an event and before the app executes an action. Furthermore, the instrumentor inserts a guard before each device action that either allows or blocks the action based on the security service's response.

Collecting Runtime Information. The code instrumentor models an app's lifecycle including its entry points, event handler methods, and call graphs. It then inserts instrumentation code that is necessary to collect the app's runtime information for policy enforcement. From the inter-procedural control flow graph (ICFG) of an app, the instrumentor proceeds in three steps: (1) it first identifies the app's actions, (2) for each action, it then performs a path-based static analysis to collect the event that triggers the action, the path condition for the action, and the numerical-valued attributes in the action call, and (3) it inserts instrumentation code before an action to transmit the action's information to the data collector. If multiple actions have the same information (event, path condition, etc.), their instrumentation code is shared. Furthermore, the instrumentation code also sends to the data collection the device ID associated with an action or an event; the device IDs are important for determining the causal interactions between the devices. For example, a user may have multiple smart switches that control a set of devices; thus a turn-on event must be associated with a specific switch.

To illustrate, we use pseudocode of the "home-away" IoT app as shown in Figure 5. When the user arrives at home, the app unlocks the front door, turns on a set of lights and sets thermostat temperature to a specific value if power consumption is less than a threshold. When she leaves, it locks the front door, turns off the lights, and sets the thermostat to another specific value. The code instrumentor searches for entry points of the app and finds two entry points: the not-present event handler that turns off the switch, locks the door, and sets the temperature (lines 10-12), and the present event handler that turns on the switch, unlocks the door and sets the temperature (lines 13-18). For each action, the code instrumentor finds the predicate that guards the action and the numerical-valued attributes used in the action call. As one example, s.on() and d.unlock() actions are triggered when the presence event happens. Since both actions share the same information (event and path condition), a single instrumentation code block is inserted before them; in particular, a code block is inserted before line 15 for transmitting the following information to the data collector:

Event: ["presence_sensor$_{id}$": present]
Actions: [["light_switch$_{id}$": on], ["door$_{id}$": unlock]]

As another example, the set-thermostat action t.set(t_home) at line 17 is conditioned on p.power>thold+5, and uses the t_home numerical-valued attribute for setting the thermostat. The instrumentor inserts a code block before line 17 to transmit the following information to the data collector:

Event: ["presence_sensor$_{id}$": present]
Action: ["thermostat$_{id}$": set(t_home)]
Action_var: ["t_home": t_home]
Predicate: ["power_meter$_{id}$.power>thold+5"]
Predicate_var: [["power_meter$_{id}$.power": power_meter$_{id}$.power, ["thold": thold]]

We note that the code instrumentation logic of an IoT app depends on the APIs that an IoT programming platform provides. For instance, some platforms explicitly allow access to the event value (e.g., presence) and device ID when an event happens, while other platforms provide this information through an event object such as event.value and event.deviceID. We present IoTGuard's code-instrumentation logic on our target IoT platform SmartThings in Section VI.

Guarding Actions. The main functionality of IoTGuard is to protect users from undesired device states. Therefore, before each action, the instrumentor also inserts a guard, which is predicated on the decision by the security service. This allows an app to execute an action based on the response returned from the security service. If the action associated with an event passes all policies, the security service returns true for the predicate that guards the action. This means the app is allowed to execute the action. If an app violates a policy, false is returned; thus the device action is not executed to preserve the system safety. For instance, the d.lock() action when user is not present (line 11) is guarded by a predicate response["door.lock"]. We will discuss more about security service in Section V-C).

B. Data Collector

An instrumented app forwards its information to the data collector when its event handler is executed. The data collector stores app's information in a dynamic model. A dynamic model is made up of a set of states and transitions. States represent the attributes of a device when an action is taken, and the transitions are the events along with the predicates that conditioned on the device actions. For instance, when the motion-active event turns on the lights at a patio after sunset, the transition is the motion-active and sunset, and the state is the light-on attribute. The actions and events include an app's device IDs for inferring the causal relationships between apps.

The data collector maintains a mutable directed graph for storing the dynamic model with additional properties to reduce the memory overhead and execution time of the policies enforced by security service. For illustration, we use two example IoT apps. The welcome-home app changes the mode to home when the light switch is turned on, and home-mode-automation app turns on the heather and crock-pot and unlocks the door when the mode is changed to home. Figure 6 depicts the structure of the states and transitions of two example apps in the data collector. The data collector represents events and device actions as nodes in the graph. A transition is added

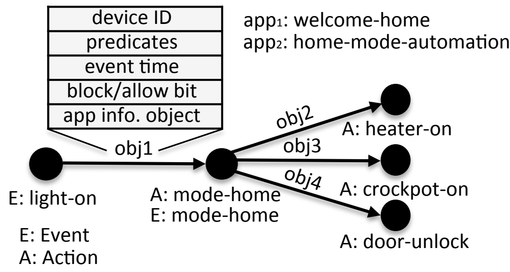

Fig. 6: Illustration of the unified dynamic model of two IoT apps recorded in the data collector.

from an app's event to each device action defined in the event handler of that event. For instance, a transition is added from "light-on" event to "mode-home" action of the welcome-home app when data collector receives the app's information. Each transition is an object, which stores an app's information (e.g., a unique ID, and app's definition), a binary bit, predicates and timestamp of the event. The binary bit guards the actions an app may execute when a particular event happens. It is initially set to NULL; however, security service updates it to false or true after evaluating the policies. The app definition is extracted from an app's definition block (if available) specified by the developer and is used to give better explanations to the users when a policy violation is enforced. Predicates are the path conditions of the paths that guard conditional device actions.

When the data collector receives an app's information, the insertion of the app's information to the dynamic model takes one of the following forms: (1) if an app's event does not exist in the dynamic model, a new event state is created, and a transition is added from the event state to each action of the app, (2) if an app's event exists in the dynamic model, a state is created for each action of the app, and a transition is added from the existing event to each action. The resulting dynamic model represents the individual behavior of an app if the app does not interact with other apps and unified behavior when apps interact with each other. To illustrate, when the welcome-home app changes the mode to home, the event handler of the home-mode-automation app is executed because its event handler subscribes to the "mode-change" event. The data collector matches the "mode change" action of the welcome-home app and the "mode-changed" event of the home-mode-automation app, and adds transitions from "mode-change" state to the home-mode-automation app's actions, which are the heater-on, crockpot-on and door-unlock states.

The dynamic model supports parallel edges, self-loops, and loops. As we detail in Section V-C, these properties allow IoTGuard to identify policy violations. For instance, if two apps implement the same functionality by turning on the switch when motion is active, data collector adds parallel edges from motion-active state to light-on state and labels the edges with the app's objects. In this case, a policy is defined by security service to prevent the repeated light-on action.

C. Security Service

The security service evaluates an app against IoT safety and security policies when the data collector receives an app's information. The policies are checked on the dynamic model of an app (if an app is independent of other apps) or on the unified dynamic model (if an app interacts with other apps). If an app's action fails to pass a policy, the security service rejects the action; otherwise, the action is executed. Implementing security service requires addressing several challenges including identifying safety and security policies for IoT (Section V-C1), and building algorithms to enforce the policies (Section V-C2).

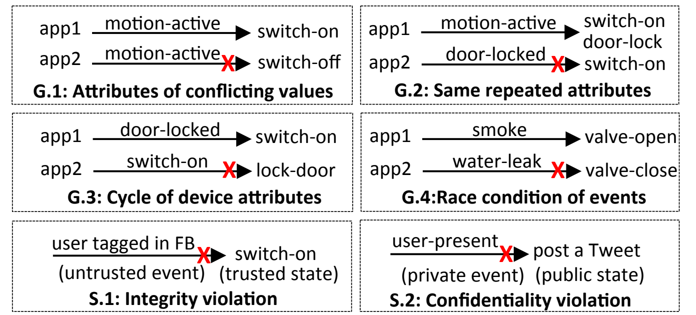

Fig. 7: Illustration of general and trigger-action platform-specific policies. Rejected states are marked with **X**.

1) Policy Identification: Policies are properties that an app must satisfy an IoT environment to be safe and secure. To define this concept for IoT, we extend recently developed IoT properties [11] to identify IoTGuard's policies. These properties were exercised on the source code of IoT apps through a model checker [11]. This approach derives requirements (properties) by evaluating the connections between assets, functional requirements, and functional constraints, where (a) *assets* are artifacts that someone places value upon, e.g., a door lock, (b) *functional requirements* define how a system needs to operate in a normal environment, e.g., when a user arrives home, the door unlocks, and (c) *functional constraints* restrict the use or operation of assets. For example, a door must open only when an authorized user requests it. We used use/misuse case requirements engineering as a policy discovery process on the IoT apps and trigger-action platform rules used in our evaluation (See Section VII), and identified 30 app-specific policies (R.1-R.30, see Table I for example policies), two trigger-action platform-specific policies, and four general policies (S.1-S.2 and G.1-G.4, see Figure 7). The complete list of policies is presented in the Appendix.

Application-specific Policies. In developing app-specific policies, we take a device-centric approach. These policies are developed based on the use cases of one or multiple devices [11]. For example, R.10 says that a smoke alarm must go off when there is smoke—thus ensuring the safe use of the smoke alarm. Another misuse case is to open the valve when there is a water leak. This leads to R.30, which says that the water valve must be closed when there is a water leak (thus involving the leak sensor and water valve). We evaluate an app against a policy if all of the devices in the policy are used in the app.

Trigger-action Platform-specific Policies. We define two trigger-action platform-specific policies to address the integrity and confidentiality violations between trigger-action platform services and IoT platforms. We first label each event and action of trigger-action apps with trusted and untrusted labels for integrity policies, and with public and private labels for confidentiality policies [27], [47]. The trusted label refers to events and actions that a user controls, and anyone can

TABLE I: Examples of application-specific policies [11]. The complete list of policies is presented in Appendix A.

ID	Policy Description
R.1	The door must always be locked when the user is not home.
R.10	The alarm must always go off when there is smoke.
R.12	The light must be off when the user is not home.
R.13	The devices (e.g., coffee machine, crock-pot) must always be on at the time set by the user.
R.14	The refrigerator and security system must always be on.
R.17	The AC and heater must not be on at the same time.
R.22	The battery of devices must not be below a specified threshold.
R.28	The sound system must not play music during the sleeping mode.
R.29	The flood sensor must always notify the user when there is water.
R.30	The water valve must be closed if a leak is detected.

```
⟨policy-set⟩        ::= [⟨statements⟩]
⟨statements⟩        ::= ⟨statement⟩ ';' [⟨statements⟩]
⟨statement⟩         ::= ⟨restrict_clause⟩ | ⟨allow_clause⟩
⟨restrict_clause⟩   ::= 'restrict' ':' [⟨transitions⟩] ':' [⟨states⟩]
⟨allow_clause⟩      ::= 'allow' ':' [⟨transitions⟩] ':' [⟨states⟩]
⟨transitions⟩       ::= ⟨transition⟩ [',' ⟨transitions⟩]
⟨transition⟩        ::= ⟨identifier⟩ | ''
⟨states⟩            ::= ⟨state⟩ [',' ⟨states⟩]
⟨state⟩             ::= ⟨identifier⟩ | ''
⟨identifier⟩        ::= ⟨word⟩
⟨word⟩              ::= ⟨char⟩ [⟨word⟩]
⟨char⟩              ::= ⟨letter⟩ | ⟨digit⟩
```

Listing 1: IoTGuard Policy Language (GPL) syntax in BNF.

cause untrusted events and actions. The private label refers to information that only a user needs to know, and the public label refers to information with unrestricted access. An integrity policy violation happens when an untrusted event changes a trusted attribute. For example, S.1 says that an app turning on the light switch when the user is tagged in a photo is an integrity violation (untrusted user-tag event turns on the light). A confidentiality policy violation happens when an event changes an attribute that makes private information publicly available. For example, S.2 says that an app that posts user's presence to social media when the door is unlocked is a confidentiality violation (user's presence is shared publicly).

We label the events and actions of an IoT platform trusted and the information obtained from an IoT system confidential. We label the events and actions of the trigger-action platform based on their properties. For instance, if a rule turns on a smart switch when the user sends an email, the send-email event is labeled with a trusted label as the user sends the email. These labels are stored in an app's dynamic model that the data collector maintains. We will detail labeling actions and events of our target trigger-action platform IFTTT in Section VI.

General Policies. General policies are constraints on dynamic models that are independent of an app's semantics—intuitively, these are states and transitions that should never occur regardless of the app domain [11]. We develop general policies according to the constraints on states and state transitions. We discuss a couple of cases. G.4 states that two or more non-complementary handlers must not change an attribute to conflicting values, e.g., a smoke-detected handler opens the water valve while leak detector closes the water valve—leading to a potential race condition when these events happen at the same time. More subtly, G.3 ensures that apps do not change a device attribute that leads to an infinite cycle of event-action pairs, e.g., a door-lock event handler turns on a switch, which is used in an event handler of another app that locks the door.

Policy Description Language. We illustrate the format and semantics of IoTGuard's policy language (GPL). Users can refine existing policies or add new policies using the GPL syntax. Listing 1 defines the policy description language in the BNF notation. A policy-set is a collection of statements that includes clauses. The collection of clauses defines a user's policies. A policy indicates combinations of transitions and state strings that should be restricted or allowed. The clauses allow each user to dictate an independent policy for devices. *Restrict* and *Allow* are two reserved tags. The clauses are compromised of two parts. The first part, *transitions*, defines a list of events and predicates. This can be a single transition or a comma-separated list of transitions. An empty entry means clauses are allowed or restricted for all transitions. The second part, *states*, is a list of device states controlling when this clause will be executed. A state expresses whether these device states are allowed or not. For example, a user may restrict a "security system off" state without specifying an event. Only if all states and transitions listed in clauses are true, a clause is true.

2) Policy Enforcement: The policies are enforced on the dynamic model of an app if the app is independent of other apps or on the unified dynamic model if the app interacts with other apps. The security service implements reachability analysis for the app-specific (R.1-R.30) and general policies (G.1-G.4), and it checks the trigger-action platform policies (S.1-S.2) based on the integrity and confidentiality labels.

For reachability analysis, the security service first obtains the events and actions of a dynamic model. It then validates policies by matching them with the events and actions of a policy clause. For example, if a set of interacting apps' unified dynamic model includes a path from a not-present event to a door-unlocked action, the security service matches this path with R.1, which says that the door should not be unlocked when the user is not present, and rejects the door-unlock action. To reduce the overhead of policy checks, the security service uses self-loop, cycle, and parallel edge detection algorithms on the dynamic model (See Section VI). For instance, G.3 says that an event of an app must not change a device attribute to a value that is used as an event that triggers a handler of another app and that leads to an infinite cycle of event and actions. To illustrate, an app turns on the switch when the door is locked, while another app locks the door when the switch is turned on. Here, the security service enforces G.3 through a cycle detection algorithm, and rejects the lock-door state of the second app to prevent the infinite cycle. We note that to enforce G.1 and G.4 (See Figure 7), the security service requires users to explicitly specify which action to be blocked. This is because the security service cannot determine which action causes violation without users specifying their needs, especially when there are conflicting policies. For example, consider when a fire alarm triggered by smoke opens the water valve to activate a sprinkler, and a moisture detector closes the water valve to cut off water source. Here the policy that guarantees the water is not running when moisture is detected conflicts with the policy that mandates a sprinkler remains on when smoke is detected. In these cases, IoTGuard requires users explicitly specify what action needs to be taken (either

to block the valve-open or the valve-close action). If the user does not specify the policy explicitly, IoTGuard implements two solutions: It may either enforce the first matching policy (allows the valve-open action when smoke is detected and blocks the valve-close action when the leak is detected) or may ask users through run-time prompts.

Lastly, the security service implements an information flow analysis algorithm to enforce trigger-action platform-specific policies (S.1 and S.2). It first obtains the integrity and confidentiality labels of the states. It then checks whether a path exists to a public state that makes private information public, and from an untrusted state to a trusted state. For integrity violations, it blocks the trusted state, and for confidentiality violations, it blocks the public state.

VI. Implementation

We implemented IoTGuard for SmartThings apps and IFTTT trigger-action applets. SmartThings supports more devices than competing IoT platforms and has a growing number of IoT apps [43]. IFTTT is a widely used trigger-action platform with over 11 million users and 54 million rules [52]. We first extract the events and actions of IFTTT rules to map each IFTTT rule to an IoT app. This allows us to execute the rules in an IoT simulator (detailed below). IoTGuard's code instrumentor then adds extra code logic to an app's source code to collect app's information at runtime without any change to the platforms. The instrumented apps are executed in the SmartThings simulator [46], which simulates the behavior of physical devices with virtual devices. Apps communicate with IoTGuard that operates on a local server through synchronous HTTP requests. We next detail each step of our implementation.

Identifying IFTTT Applet Events and Actions. For trigger-action platform rules, we use IFTTT applets designed for SmartThings [25]. In April of 2018, we obtained over 100 IFTTT SmartThings applets. The IFTTT applets are strings, for example, "log door openings to Google Spreadsheet when the door is unlocked by SmartThings." Here, our goal is to obtain events and actions of an applet and map them to an app that executes within the SmartThings simulator. Turning to the example applet, the app executed in the simulator transmits "log the door-unlock state to the Google spreadsheet" action to IoTGuard when the "door-unlock" event happens. We build a SmartThings app that subscribes to the door-unlock event, and create the "log the door-unlock state to the Google spreadsheet" process when the door-unlock event handler is invoked. The rule executes in a special security context, where it only has access to SmartThings devices and the services connected to the SmartThings devices authorized by the user at install time.

To do so, we first crawl IFTTT applets and obtain the SmartThings applets. We then tokenize the applets, where each token is an alphanumeric word, filter tokens that are stop words, and then stem them with the Porter stemmer [7]. We then create an inverted index of the tokens. The inverted index is used to search the IFTTT-provided actions and events. For example, if the search hits the "door lock" action of SmartThings before the "when" keyword, it is an action, and if the search hits "user present" after the "when" keyword, it is an event. When an applet does not contain "when", we consider it an IFTTT DO applet. DO applets only include SmartThings actions, and

```
 1: // Devices and user inputs
 2: preferences {...}
 3: // Events
 4: subscribe(presenceSensor, "presence", presenceHandler)
 5: // App information
 6: state.appID = "app1"
 7: state.appDescription = "welcome home app…"
 8: state.appName = "welcome-home”
 9: // Entry point
10: def presenceHandler(evt){
11:   if(evt.value == "present"){
12:     def t_home = 65
13:     def d_thold = 5
14:     def power = meter.currentValue("power")
15:     actions = [action: ["s.on()","d.unlock()"],
16:     response = sendRequest(evt,actions)
17:     if(response["s.on"]){s.on()}
18:     if(response["d.unlock"]){d.unlock()}
19:     if(power < thold + d_hold){
20:       actions = [action: ["t.set…(t_home)"],
21:                  action_var: [t_home:t_home],
22:                  pred: "power<thold+d_hold",
23:                  pred_var: [power:power,thold:thold, d_thold:d_thold]
24:       response = sendRequest(evt,actions)
25:       if(response["t.setHeatingSetpoint"]){
26:       t.setHeatingSetpoint(t_home)
27:     }
28: }
29: // Code block of transmitting app information to IoTGuard
30: def sendRequest(evt, actions){
31:   def params // Set IoTGuard server
32:   def jsonRequest  // Create JSON request object
33:   // Append app info from state object
34:   // Append event info (e.g., event value (evt.value)) from evt object instance
35:   // Append device info (e.g., device type (s.typeName)) from device object instance
36:   // Send request to IoTGuard's data collector
37:   httpPostJson(params){ resp-> ...
38:   }
39:  return response
40: }
```

Fig. 8: IoTGuard's code instrumentation logic for the app's presence event handler depicted in Figure 5 (App's other event handlers are similarly instrumented). The instrumented code is highlighted in grey color. The actions guarded with the IoTGuard's response are highlighted in dashed-red boxes.

the actions are invoked through the IFTTT website or the DO mobile app. We map triggers of the DO rules to the "app touch" event of the SmartThings platform and the "app touch" event performs actions when a user clicks on a button in the simulated app. We found that identifying an IFTTT applet's actions and events in some cases requires manual effort because IFTTT applets are not well structured, and their definitions are often unclear (See our discussion in Section VIII). Therefore, we manually check each applet and verify the events and actions, we then associate each action and event with integrity and confidentiality labels required for checking the trigger-action platform-specific policies.

Code Instrumentor. Apps are instrumented by the code instrumentor before they are executed in the SmartThings simulator. Figure 8 shows the instrumented version of the example app in Figure 5. The instrumentor works on the Abstract Syntax Tree (AST) representation of a SmartThings app's Groovy code. The Groovy compiler supports customizing the compilation process with compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [31]). The code instrumentor visits AST nodes during the Groovy compiler's semantic analysis phase when it performs consistency and validity checks on the AST. Our implementation uses an ASTTransformation to hook into the compiler, ASTBrowser to extract entry points, method calls, and expressions inside AST nodes. This allows our implementation to insert an app's information such as the app ID and app name (lines 6-8), and obtain an app's events (line 10), actions, numerical-valued

attributes, and predicates that guard actions (line 15 and 20). The information is transmitted to IoTGuard's data collector (line 16 and 24) through a JSON object with additional information obtained from the app's state, event, and device object instances (lines 29-40). The event object in SmartThings allows accessing the event properties [15]; for example, the event type is obtained through evt.value (line 34). Similarly, a device object allows accessing device features [13]; for example, the device type is obtained from developer-defined device input s through s.typeName. The response returned from IoTGuard either allows or denies the app's actions (lines 17-18 and 25).

The SmartThings programming platform has a number of idiosyncrasies that the code instrumentor needs to address for precise code instrumentation: (1) abstract transitions and states, (2) state variables, and (3) calls by reflection. First, abstract events are triggered when a user clicks on an app icon or by a pre-defined event such as location mode change from away to home. Additionally, events may lead to abstract states. For instance, setLocationMode() sets the location mode to a pre-defined mode. The code instrumentor models app lifecycle based on the complete set of abstract events and states defined in the SmartThings documentation [44]. Second, apps may use state variables that are stored in either the global state or atomicState object to persist data across executions. State variables are often used in conditional branches to guard state transitions. The code instrumentor applies field-sensitive analysis to track the data dependencies of all fields defined in the state and atomicState objects. Lastly, SmartThings supports call by reflection (using GString) [44], which allows a method to be invoked by providing its name as a string. To handle calls by reflection, the code instrumentor's call graph construction adds all methods in an app as possible call targets.

Data Collector and Security Service. The data collector and security service run on a Jetty [3] local server. App requests are tunneled from the SmartThings cloud to the local server running IoTGuard with ngrok [38]. The data collector extends Guava's Graph library [19] to store dynamic models because of its computation efficacy and openness. The graph library implements a network data structure that provides important prerequisites for our purposes, in particular, parallel edges, self-loops and unique transition objects. The network data structure uses hash-based (and enum-based) collections, which implement single-entry operations in constant time and all tree-based/sorted collections have logarithmic time for single-entry operations. The security service implements graph algorithms on top of Guava's network data structure to enforce policies on the dynamic models, i.e., reachability analysis, self-loop and cycle detection, and information-flow analysis.

IoTGuard User Console. Figure 9 shows the user console of IoTGuard. The console displays a visual representation of a policy violation. For each policy violation, it shows the description of the violated policy and events and actions of the interacting apps that lead to the violation. The users can either select IoTGuard to automate the blocking of an action that violates a policy (❶) or may allow or deny the action through a runtime prompt (❷). The second option is less secure for users who install apps without understanding warnings. Furthermore, runtime prompts in some cases may prevent real-time automation; for instance, users need to be awake to approve an action. We note that the IoT console can be improved

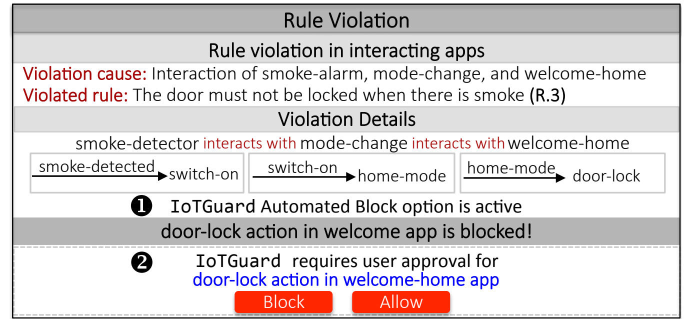

Fig. 9: IoTGuard user console provides two solutions for policy violations: blocking the undesired state and informing users about the policy violation (❶) and allowing users to reject or accept the actions through runtime prompts (❷).

with various information such as app descriptions and device locations through IoTGuard's data collector to meet the usability and accessibility requirements for users.

VII. Evaluation

We present two studies evaluating the IoTGuard system—one synthetic and one market-based. The first is a study of 15 hand-crafted SmartThings apps and five IFTTT applets, which contain a number of representative policy violations (Section VII-A). In a second study, we execute market vetted of 35 SmartThings apps and 30 IFTTT applets with various configurations in a simulated smart home (Section VII-B). Lastly, we study the performance overhead of the IoTGuard system (Section VII-C). In these studies, we sought to validate the correctness, completeness, and performance of IoTGuard on the target apps. We performed our experiments on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java runtime version 1.8 (64 bit) in its default settings. We use the SmartThings simulator [46] the execute the apps. The apps send their information to the IoTGuard system that runs on a Jetty 8 HTTP server and Java Servlet container [3]. The requests of the apps are tunneled from SmartThings cloud to the local server with ngrok 2.0 [38].

A. Effectiveness

This section reports on an application study that uses IoTGuard to analyze how 15 hand-crafted SmartThings apps (ST1-ST15) and five IFTTT (IFTTT1-IFTTT5) applets violate the policies. Each app represents a unique malicious behavior or flaw that causes a policy violation in an individual app and multi-app environments. The apps include various devices and services covering diverse real-life use cases. We constructed these apps based on a survey of recent literature on IoT safety and security [11], [14], [28], [47], [51].

Our analysis of IoTGuard showed that it correctly enforced 12 of the 12 policy violations, including a policy violation in an individual app and 11 policy violations in five group of apps that interact with each other. We manually exercised the functionality offered by the apps and confirmed the policy violations. Table II shows the groups of apps, transitions, and states of the apps, violated policies and blocked states to prevent the violations. Each group includes a set of apps that are co-located in an environment and authorized to use the same

TABLE II: Effectiveness of IoTGuard in enforcing the policies in malicious and flawed apps.

Gr.ID	App†	Transitions/States	Enforced Pol.	Blocked States
1	ST1	$\xrightarrow{\text{battery low}}$ unlock front door	R.1	unlock front door (ST1)
2	IFTTT1	$\xrightarrow{\text{11pm}}$ turn off lights	R.14(x2)	turn off alarm (ST3) turn off security system (ST3)
	ST2	$\xrightarrow{\text{lights turned off}}$ to sleeping mode		
	ST3	$\xrightarrow{\text{mode changed}}$ turn off appliances		
3	ST4	$\xrightarrow{\text{smoke detected}}$ turn on lights and alarm	R.3 S.2	lock door (ST6) log public spreadsheet (IFTTT2)
	ST5	$\xrightarrow{\text{lights on}}$ to home mode		
	ST6	$\xrightarrow{\text{home-mode}}$ lock door		
	IFTTT2	$\xrightarrow{\text{door-locked}}$ log to a public spreadsheet		
4	ST7	$\xrightarrow{\text{contact sensor open}}$ turn on lights	G.1	turn off lights (ST8)
	ST8	$\xrightarrow{\text{contact sensor open}}$ turn off lights		
5	IFTTT3	$\xrightarrow{\text{Google Assistant (by voice)}}$ turn off light	G.3	turn off light (ST10)
	ST9	$\xrightarrow{\text{light turned off}}$ change mode		
	ST10	$\xrightarrow{\text{mode-change}}$ turn off light		
6	IFTTT4	$\xrightarrow{\text{Anyone checks in \#hashtag}}$ unlock door	S.1 R.13(x3) R.12	unlock door (IFTTT4) brew coffee (ST11) sound music (ST12) set thermostat cooling (ST14) set thermostat heating (ST15)
	IFTTT5	$\xrightarrow{\text{email sent}}$ turn on light		
	ST11	$\xrightarrow{\text{light turned on}}$ brew coffee		
	ST12	$\xrightarrow{\text{light turned on}}$ sound music		
	ST13	$\xrightarrow{\text{light turned on}}$ change mode		
	ST14	$\xrightarrow{\text{mode-change}}$ set thermostat cooling		
	ST15	$\xrightarrow{\text{mode-change}}$ set thermostat heating		

† ST is for SmartThings apps, and IFTTT is for IFTTT applets.

TABLE III: Properties of analyzed IoT apps and trigger-action platform applets in market-based studies.

	Nr.	Uniq. Devices	Uniq. Services	#Events	#Actions	Func.
IoT	35	20	–	86	78	†
Trigger-action	30	7	12	30	30	‡

† The SmartThings apps cover functionality including security and safety, green living, convenience, home automation, and personal care. We determined an app's functionality by checking definition block in its source code.

‡ The IFTTT applets connect SmartThings with services of the phone call, Foursquare, Google Spreadsheet, Google Voice, time, email, Philips, Slack, Douglas, Twitter, GraspIO, and Wemo.

devices. In the following discussion, we will use app group IDs (Gr.1-Gr.6) in Table II. For instance, in Gr.1, IoTGuard enforces R.1 and blocks the "unlock front door" action of ST1 that unlocks the front door without checking whether the user is at home. In Gr.3, three IoT apps (ST4-ST6) and one IFTTT app (IFTTT2) interacts with each other. The interaction between ST4, ST5 and ST6 violates R.3 by locking the door when there is smoke at home. ST6 and IFTTT2 violates S.2 by logging private door-locked state to a public file. IoTGuard blocks the "lock door" action of ST6 to prevent violation of R.3, and "log door-state to a public spreadsheet" action of IFTTT2 to prevent violation of S.2.

B. Market App Study

We performed two market-based studies to evaluate the effectiveness of the IoTGuard in supporting users in avoiding undesired states. In a first study, we configure apps with a single separate device, and in a second study, we configure the apps with multiple devices based on the description of apps. Through these studies, we evaluate IoTGuard in violations that can happen in practice when an app works in isolation and when multi-apps are co-located in an environment.

Experimental Setup. We simulate a smart home as shown in Figure 10. The smart home includes 20 different IoT devices, a total of 29 devices. Some IoT devices are deployed multiple times; for example, water leak detector (⑪) is deployed both in the kitchen and bathroom. These devices are the most selling IoT consumer products for smart home [1]. To build automated

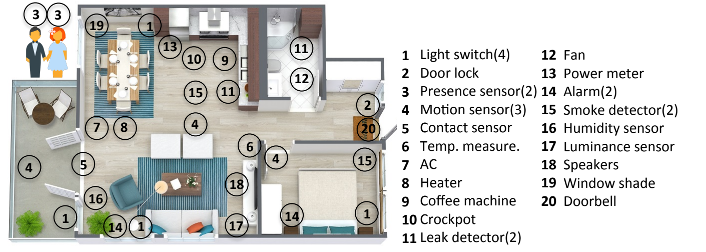

Fig. 10: The simulated smart home used in market app study.

tasks for the smart home devices, we obtained 35 official (vetted) IoT apps (M.ST1-M.ST35) from the SmartThings GitHub repository [40] and 30 official IFTTT applets (M.IFTTT1-M.IFTTT30) from IFTTT market [25], [47] (See Table III). The IFTTT applets connect seven IoT devices with twelve unique services such as Google voice and phone call. These apps and applets include various devices, services, and functionality that encompass diverse real-life use cases. Some apps require pre-defined mode inputs. We defined four modes, home, away, sleeping and vacation based on the use cases of modes in SmartThings documentation [36]. We generate an app's all events to trigger its all event handler methods. If an app requires a input for a numerical-valued device attribute, we generate inputs in a range the device supports based on the app logic (similar to fuzz testing that guides a fuzzer to cover the app code intelligently [48]). For instance, a thermostat input to set the temperature value can be generated between 50 and 95.

Apps Used in Isolation. In our first study, we run each app by configuring with a single separate device. For instance, an app that turns on lights in the kitchen when motion is active is configured with a smart switch and a motion detector in the kitchen. The goal of the study is to enforce policy violations when apps are used in isolation; however, we found that apps require a greater number of devices than those found in the smart home. For instance, eight apps in our corpus use a motion detector, yet three motion detectors are deployed in the smart home. To be consistent in our experiments, we assume the apps sharing a common device are not installed by a user at once.

We found that apps used in isolation lead to unsafe and undesired states. Table IV rows labeled with ❶ shows the violations and blocked states. IoTGuard enforced a policy that an IFTTT app violates, and two policies in two groups that have four apps. We found that there are three reasons for policy violations enforced by IoTGuard. First, though apps are configured with a single separate device, the interaction of apps through abstract attributes cause policy violations; for example, when M.ST4 changes the mode at a specific time, M.ST7 turns on a configured appliances (heater based on our configuration) when the user is not at home (R.13). Second, misconfiguration of numerical-valued device attributes such as thermostat heating point cause policy violations; for example, AC and heater run at the same time when a common heating and cooling value is set (R.17). The reason behind the configuration errors is running the apps with the complete test inputs that a device supports; thus these errors depend on the user's configuration of apps' numerical-valued attributes at install time. Third, M.IFTTT24 violates S.1 by turning on a light switch when someone Tweets #ChristmasSpirit. IoTGuard enforces S.1 and blocks "light turn on action". We note that none of the official SmartThings apps

TABLE IV: Potential policy violations by 65 (35 IoT apps and 30 IFTTT applets) of the studied apps.

Study	Gr.ID	App‡	Transitions/States	Enforced Pol.	Blocked State
❶†	1	M.ST11	$\xrightarrow{\text{temp}\geq\text{user_input}}$ heater on	R.17 R.13 S.1	AC on (M.ST12) heater switch on (M.ST7) light on (M.IFTTT24)
		M.ST12	$\xrightarrow{\text{temp}\geq\text{user input}}$ AC on		
	2	M.ST4	$\xrightarrow{\text{time}}$ mode change		
		M.ST7	$\xrightarrow{\text{mode-change}}$ heater switch on		
	3	M.IFTTT24	$\xrightarrow{\text{anyone Tweets \#ChristmasSpirit}}$ light on		
❷†	1	M.ST21	$\xrightarrow{\text{motion active}}$ lights on	G.2(x5)	lights on (M.ST9) lights on (M.ST15) lights off (M.ST9) lights on (M.ST9) capture photo (M.IFTTT21)
			$\xrightarrow{\text{motion inactive}}$ lights off		
		M.ST15	$\xrightarrow{\text{motion active}}$ lights on		
		M.ST9	$\xrightarrow{\text{motion active}}$ lights on		
			$\xrightarrow{\text{motion inactive}}$ lights off		
		M.IFTTT21	$\xrightarrow{\text{light on}}$ capture photo		
	2	M.ST1	$\xrightarrow{\text{motion inactive}}$ switch off	G.3 R.13(x3) R.12 R.14 S.1(x2) S.2	switch on (M.ST7) heater, coffe mac., crock. on (M.ST7) light on (M.ST7) alarm off (M.ST6) switch on (M.IFTTT20) send Slack notification (M.IFTTT16) open window shade (M.IFTTT17)
		M.ST2	$\xrightarrow{\text{power}>\text{threshold}}$ switch off		
		M.ST33	$\xrightarrow{\text{time}}$ switch off		
		M.IFTTT1	$\xrightarrow{\text{sunrise}}$ switch off		
		M.IFTTT28	$\xrightarrow{\text{Google voice}}$ switch off		
		M.ST23	$\xrightarrow{\text{contact sensor open}}$ switch off		
		M.ST10	$\xrightarrow{\text{switch off}}$ mode change		
		M.ST6	$\xrightarrow{\text{mode change}}$ switch off		
		M.ST7	$\xrightarrow{\text{mode change}}$ switch on		
		M.IFTTT13	$\xrightarrow{\text{leak detected}}$ switch on		
		M.IFTTT20	$\xrightarrow{\text{door ring pressed}}$ switch on		
		M.IFTTT9	$\xrightarrow{\text{missed call}}$ switch on		
		M.IFTTT30	$\xrightarrow{\text{send email}}$ switch on		
		M.IFTTT16	$\xrightarrow{\text{switch on}}$ send Slack notification		
		M.IFTTT17	$\xrightarrow{\text{switch on}}$ open window shade		
	3	M.ST21	$\xrightarrow{\text{motion inactive}}$ switch off	G.1(x2) G.2 G.4	switch off (M.ST15) switch on (M.ST7) switch off (M.ST6) switch off (M.ST23)
		M.ST15	$\xrightarrow{\text{motion inactive}}$ switch off		
		M.ST7	$\xrightarrow{\text{contact sensor open}}$ switch on		
		M.ST6	$\xrightarrow{\text{contact sensor open}}$ switch off		
		M.ST18	$\xrightarrow{\text{app touch}}$ switch on		
		M.ST23	$\xrightarrow{\text{app touch}}$ switch off		

† ❶ is the study results of apps used in isolation, and ❷ is the results of multi-apps co-located in an environment.
‡ M.ST is for SmartThings market apps, and M.IFTTT is for IFTTT market applets.

were flagged as violating policies when apps run in isolation; we believe this is because of the strict manual vetting enforced on official SmartThings apps, which takes a couple of months [45].

Apps Co-located in an Environment. In a second study, we configure the apps and applets with a number of devices based on app descriptions. For instance, if an app's description states that "Turn things off if you are using too much energy", and if an applets description states that "At sunrise automatically turn off a smart device you choose", we configure the apps with all switches in the smart home. Our goal in this set of experiments is to evaluate the effectiveness of IoTGuard on policy violations when apps share at least a common device. Naturally, this can happen in practice when the apps are co-located in an environment by a user.

We found that multiple apps work in concert violates nine unique properties. IoTGuard blocked 18 unsafe and undesired states in three group of apps violating multiple policies. We examined 16 apps and nine applets that interact with each other through 27 events and actions. Table IV rows labeled with ❷ shows the app groups, transitions, and states that constitute violations, violated policies and blocked states. In the following discussion, we will use app group IDs (Gr.1-Gr.3) in Table IV. Each group includes a set of apps and applets that a user may install together and authorize them to use the same devices.

In Gr.1, M.ST21, M.ST9, and M.ST15 turn on the lights when motion is active and turns off the light when motion is inactive. This leads to turning on and off the lights multiple times because of the same functionality provided in some branches of the apps. Similarly, M.IFTTT20 takes a photo multiple times every time the lights are turned on. IoTGuard enforces G.2 and blocks all repeated states. In Gr.2, a set of apps and applets turns off the switch with different events such as time, and voice. When "switch off" event happens, M.ST10 changes the mode. When the "mode" is changed, M.ST6 and M.ST7 turns off and turns on a set of devices. The interaction between apps and applets result in an unauthorized control of a set of devices. IoTGuard enforces R.12, R.13, R.14, and G.3 policies and blocks the states that cause security and safety risks for users. For instance, IoTGuard blocks "heater on" and "crockpot on" states (M.ST7), and "alarm off" state (M.ST6) when the mode is changed to sleeping, away and vacation. Similarly, a set of applets turns on the switch when "door ring pressed", "missed call", "leak" and "send email" events happen. In this, IoTGuard enforces integrity and confidentiality policies of S.1 and S.2. For instance, "open window shade" (M.IFTTT17) state is blocked when the door ring pressed, and "send Slack notification" is blocked when the switch is turned on. Lastly, in Gr.3, IoTGuard enforces G.1 when "contact sensor open" event of M.ST18 and M.ST23 change the switch state to conflicting values of "on" and "off". Furthermore, IoTGuard enforces G.4 when "app touch", "motion inactive", and "contact sensor open" events change a device state to conflicting "on" and "off" states when these events happen at the same time.

C. Performance Evaluation

We study IoTGuard's code instrumentation and runtime overhead. We performed the tests during the market app study.

Code Instrumentation Performance. We evaluated IoTGuard's code instrumentor in terms of the process time required for adding the instrumentation code to an app, and the number of Lines of Code (LoC) required for instrumenting an app. The average time to insert instrumentation code for an app is 4.1±2 secs. The SmartThings apps are on average 220 LoC, and the number of LoC added to an app is on average 20±8 LoC (9.1%). IFTTT rules are on average 60 LoC after they are converted to an app that runs on the SmartThings simulator, and the number of LoC added to an IFTTT rule is 8±2 (13.3%). We note that IoTGuard also appends on average 20 LoC for transmitting the app's information. An app's instrumentation time and the number of LoC depend on the algorithms developed for extracting the events, actions, and predicates of the apps. For instance, an app that has many actions in conditional branches takes more time than an app that does not have any branches. We note that the code instrumentor adds the instrumentation code to an app at install time; thus it does not introduce runtime overhead.

Runtime Overhead. To study the overhead introduced into a system by IoTGuard, we record, end-to-end overhead, the time between when an app receives an event and when an app executes an action. For instance, the end-to-end overhead of an app that turns on the switch when the user is present is the time between triggering the "user-present" event handler and executing the "switch on" action. We generate the consecutive events of the apps with instrumentation and without instrumentation and measure each test 20 times. The end-to-end overhead of apps without instrumentation is on average 0.52±0.2 secs. The end-to-end overhead of instrumented apps includes the time for transmitting the app's information to the data collector, checking the policies and sending a response to

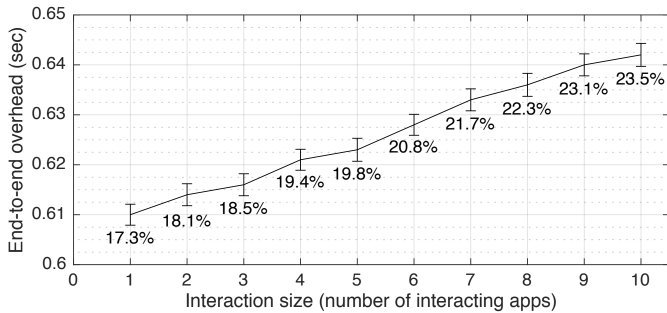

Fig. 11: IoTGuard's end-to-end overhead on policy enforcement. Error bars indicate standard errors, and percentages shows the overhead with respect to the unmodified system.

the app. Figure 11 shows the end-to-end overhead, in seconds, of the different number of interacting apps. The interaction size represents the number of states which impacts the number of policies that IoTGuard checks on the unified dynamic model of interacting apps. For instance, if ten apps are interacting with each other, IoTGuard checks more policies because the number of devices that a unified dynamic model includes is more than the devices of an app's dynamic model. As can be seen, most policy checks on an instrumented app require on average 90 ms (17.3%) with respect to the unmodified system. The overhead increases with the number of interacting apps. For instance, the overhead for ten interacting apps is on average 122 ms, which constitutes less than a 23.5% runtime overhead. The end-to-end overhead is dominated by buffering of app's information and checking the policies. While these overheads are acceptably low for many applications, they may be partially reduced by a tighter coupling of IoTGuard and the edge system (i.e., hub or cloud). We note that the actual overhead in an IoT system often happens due to the communication between the edge system and physical device; for example, execution of a device action often has a latency over a second [28], [51]. Thus, IoTGuard's overhead in real-world scenarios would be negligible because it does not add latency for device action execution.

IoTGuard Console-prompt and Data Storage Overhead. When the user deactivates the automated blocking, IoTGuard provides the user with a console to review the policy violation, and the user may either deny or allow an app's action. We measure the overhead of displaying the console to the users through a Web interface in 21 policy violations recorded in our market-based study. The console adds negligible perceived latency, on the order of milliseconds, to the end-to-end overhead. We next determine the storage cost of IoTGuard by measuring the app's information recorded in the data collector. We randomly triggered 500 app events by considering a highly active IoT user. The data collector imposes 80KB of storage cost. We note that storage cost can be reduced either by deleting the logs based on the user's needs or integrating the IoTGuard into the edge system or cloud based on the IoT platform architecture.

VIII. Limitations and Discussion

A limitation of IoTGuard is in taking the right course of action if a state is blocked. In some cases, merely blocking a state caused by users or policy errors could have physical consequences. For example, suppose that a door should be unlocked only for a security service based on a time window specified by the user when she is on vacation. However, a policy that blocks the unlock-door state prevents the security service from entering the house, which may or may not be preferable depending on the circumstances. To help keep the IoT environment stable when an action is rejected, future work will need to study more complex policies through multiple users and better handle blocked states.

IoTGuard allows a user to specify policies through IoTGuard's GPL. This can pose problems especially when users create policies in highly complex IoT environments, where an incorrect policy specification may prevent legitimate states, fail to block unsafe and insecure states, or conflict with another policy. For instance, one policy may allow action "a" when a specific event occurs, while a second policy may deny a set of actions, of which "a" is a member. To address these issues, we plan to adapt machine learning and other modeling techniques to automate the property-discovery process and policy conflict resolution in IoT devices and domains.

IoTGuard implements an algorithm to find the events and actions of IFTTT trigger-action applets. Thereafter, we manually label the events and actions with integrity and confidentiality labels. We found that extracting IFTTT events and actions and labeling them is not a trivial process because an applet's event and actions often do not match the device capabilities of an IoT platform. Additionally, this process does not scale to a large number of IFTTT applets. We plan to use more semantically rich natural language processing techniques for automated and scalable applet processing.

We have shown that IoTGuard can express meaningful policies to preserve system safety and security. We plan to conduct a user study to evaluate the usability of IoTGuard based on user configuration of the apps. We will ask independent users to configure the IoT apps and trigger-action applets with the assumption that they deploy them in a smart home. We will then execute the apps and study the effectiveness of IoTGuard, focusing on policies, blocked states, and user-perceived risks based on specific user configurations.

Lastly, IoTGuard's implementation and evaluation are based purely on the SmartThings home automation platform and IFTTT trigger-action platform apps. There are other IoT domains suitable to evaluate safety and security violations, such as FarmBeats for agriculture [50], HealthSaaS for healthcare [23], and KaaIoT for the automobile industry [29], and Zapier [53] and Microsoft Flow [35] for trigger-action platforms. We plan to extend IoTGuard's algorithms to these platforms and engage in large-scale analyses of IoT markets and industries.

IX. Related Work

There has been an increasing amount of recent research exploring IoT security and more broadly safety. We compare IoTGuard with several previous approaches that differ in scope, focus, precision, and runtime. The approaches studied here are the most applicable that run directly on IoT app source code. As presented in Table V, IoTGuard supports more features than any previous approach to IoT security. ContexIoT is a permission-based system that provides contextual integrity for IoT apps at run time [28]. SmartAuth generates an authorization interface for users and enforces the apps permissions after a

TABLE V: A comparison of IoTGuard with other IoT systems.

System	Constraints			
	Multi-app analysis	Trigger-action applet analysis	Policy identification	Runtime policy enforcement
ContexIoT [28]	✗	✗	✗	✗
SmartAuth [49]	✗	✗	✗	✗
ProvThings [51]	✓	✗	✗†	✗
Soteria [11]	✓	✗	✓‡	✗
IoTGuard	✓	✓	✓	✓

† ProvThings implements a policy engine that allows users to create policies through provenance database.
‡ Soteria identifies safety and security property violations through source code analysis.

user authorized them [49]. ContexIoT and SmartAuth are only applicable to an IoT app running in isolation—collecting context of an individual app. ProvThings logs system-level provenance through security-sensitive APIs and leverages it for forensic reconstruction [51]. Lastly, Soteria is a static analysis system for model checking of IoT apps to validate whether an IoT app or IoT environment adhere to safety and security properties [11]. ProvThings and Soteria support analysis of interactions among IoT apps. ProvThings supports this capability through the analysis of provenance logs of multiple apps, and Soteria constructs a union state model that represents the unified behavior of apps when they installed together. However, ProvThings and Soteria do not handle the interactions between IoT apps and trigger-action platform services. Furthermore, none of the systems evaluate and ultimately enforce identified security and safety policies on market-apps to protect users from undesired states at runtime.

Traditional security measures have been used to mediate access to system resources such as files, ports, etc. [27]. Instead, IoTGuard directly mediates actions sent by apps to the physical devices. Previous representative efforts at securing control systems have constructed models using state-space and control-theoretic approaches to model the normal operation of the devices for detecting anomalies and faulty systems. The examples include models built on water control systems [21], chemical reactor processes [8], medical devices [24] and power grid systems [33]. These tools model applications using the domain-specific information and exploit the structure of the control system implementations, e.g., plant behavior [37] and process controller code [34]. While we build on these results, IoTGuard addresses the diversity of IoT devices in sensors, resources, and interactions among devices which provides unique challenges that require a different approach to preserving the safety and security of the IoT environment.

X. Conclusions

As users become more comfortable installing IoT apps and trigger-action platform rules in an IoT environment, the interaction between devices will increase. IoTGuard detects when an individual app and interactions among apps lead to unsafe and insecure states and ameliorate these undesired states by blocking them. We evaluated IoTGuard in two studies: a study on a flawed app corpus, and a market study of SmartThings apps and IFTTT applets. These studies demonstrated that IoTGuard accurately identifies policy violations and blocks the undesired states, both when apps are used in isolation and when they are used together in multi-app environments. IoTGuard incurs less than 17.3% runtime overhead for an individual app and 19.8% for five interacting apps with respect to the unmodified system.

Future work will expand our analysis to support more platforms and to continue to study security requirements engineering process to discover more complex and subtle policies. This work is the first step toward realizing practical IoT security and safety. We plan to extend our study to the complex interactions between users and the smart devices that they use to enhance their lives.

Acknowledgment

We thank our shepherd Luyi Xing, and Eric Pauley for their comments and suggestions. Research was supported in part by the Army Research Laboratory, under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and the NSF Grant No. CNS-1564105. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] "Best seller home improvement automation devices," https://goo.gl/XLLzUP, [Online; accessed 21-July-2018].

[2] "Android sensor API documentation," https://goo.gl/vEDwKu, [Online; accessed 30-July-2018].

[3] Apache, "Jetty servlet engine and Http server," https://www.eclipse.org/jetty, 2018, [Online; accessed 30-August-2018].

[4] "Apiant: Connect your apps, automate your business," https://apiant.com/, [Online; accessed 11-April-2018].

[5] "Apple's HomeKit," https://www.apple.com/ios/home/, [Online; accessed 9-January-2018].

[6] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in IoT apps," in *ACM CCS*, 2018.

[7] S. Bird and E. Loper, "Nltk: Natural language toolkit," in *ACL Interactive poster and demonstration sessions*. Association for Computational Linguistics, 2004.

[8] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: risk assessment, detection, and response," in *ACM symposium on information, computer and communications security*, 2011.

[9] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity IoT," in *USENIX Security*, 2018.

[10] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel, "Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities," *arXiv preprint: 1809.06962*, 2018.

[11] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[12] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app threats in smart homes: Categorization, detection and handling," *arXiv preprint:1808.02125*, 2018.

[13] "SmartThings device API documentation," https://goo.gl/HCtuka, [Online; accessed 29-July-2018].

[14] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *ACM CCS*, 2018.

[15] "SmartThings event API documentation," https://goo.gl/GPPXV3, [Online; accessed 29-July-2018].

[16] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *IEEE Security and Privacy (S&P)*, 2016.

[17] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical data protection for emerging IoT application frameworks," in *USENIX Security*, 2016.

[18] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action IoT platforms," in *NDSS*, 2018.

[19] Google, "Guava: Google core libraries for Java 1.7+," https://github.com/google/guava, 2018.

[20] Google Fit Developer Documentation., https://developers.google.com/fit/, [Online; accessed 20-August-2018].

[21] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, "Through the eye of the PLC: semantic security monitoring for industrial processes," in *ACSAC*, 2014.

[22] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur, "Rethinking access control and authentication for the home Internet of Things (IoT)," in *USENIX Security*, 2018.

[23] "HealthSaaS: The Internet of Things (IoT) Platform for Healthcare," https://www.healthsaas.net/, [Online; accessed 20-August-2018].

[24] X. Hei, X. Du, S. Lin, and I. Lee, "Pipac: Patient infusion pattern based access control scheme for wireless insulin pump system." in *INFOCOM*, 2013.

[25] "IFTTT SmartThings platform rules," https://ifttt.com/smartthings, [Online; accessed 11-July-2017].

[26] "IFTTT (if this, then that): Helps your apps and devices work together," https://ifttt.com/, [Online; accessed 30-August-2018].

[27] T. Jaeger, "Operating system security," *Synthesis Lectures on Information Security, Privacy and Trust*, 2008.

[28] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity, "ContexIoT: Towards providing contextual integrity to appified IoT platforms," in *NDSS*, 2017.

[29] "KaaIoT: Connected car and IoT automotive," https://www.kaaproject.org/automotive, [Online; accessed 20-August-2018].

[30] S. Kubler, K. Främling, and A. Buda, "A standardized approach to deal with firewall and mobility policies in the IoT," *Pervasive and Mobile Computing*, 2015.

[31] C. Lattner, *LLVM compiler infrastructure project*. The architecture of open source applications, 2012.

[32] O. Leiba, Y. Yitzchak, R. Bitton, A. Nadler, and A. Shabtai, "Incentivized delivery network of IoT software updates based on trustless proof-of-distribution," *arXiv preprint: 1805.04282*, 2018.

[33] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," *ACM TISSEC*, 2011.

[34] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A trusted safety verifier for process controller code." in *NDSS*, 2014.

[35] "Microsoft Flow: Automate processes + tasks," https://flow.microsoft.com, [Online; accessed 11-April-2018].

[36] "Modes in SmartThings," https://goo.gl/DRCHPo, [Online; accessed 21-August-2018].

[37] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *ACM International Conference on High Confidence Networked Systems*, 2013.

[38] "ngrok: Public URLs for exposing your local web server," https://ngrok.com/, [Online; accessed 9-July-2018].

[39] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "IoTSan: Fortifying the safety of IoT systems," in *CoNEXT*, 2018.

[40] "SmartThings Official App Repository," https://github.com/SmartThingsCommunity, [Online; accessed 10-January-2018].

[41] OpenHAB: Open Source Automation Software for Home, https://www.openhab.org/, [Online; accessed 9-January-2018].

[42] "PTC: Innovation with industrial IoT," https://www.ptc.com/en/about, [Online; accessed 20-July-2018].

[43] "Samsung SmartThings," https://www.smartthings.com/, [Online; accessed 29-July-2018].

[44] "SmartThings documentation," http://docs.smartthings.com, [Online; accessed 29-July-2018].

[45] "SmartThings Code Review Guidelines and Best Practices," http://docs.smartthings.com/en/latest/code-review-guidelines.html, [Online; accessed 29-July-2018].

[46] "SmartThings iot platform simulator," https://goo.gl/rfTB7e, [Online; accessed 9-July-2018].

[47] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes," in *World Wide Web (WWW)*, 2017.

[48] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *NDSS*, 2015.

[49] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the Internet of Things," in *USENIX Security*, 2017.

[50] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. N. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, "FarmBeats: An IoT platform for data-driven agriculture," in *NSDI*, 2017.

[51] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the Internet of Things," in *NDSS*, 2018.

[52] "IFTTT platform size metrics," https://platform.ifttt.com/pricing, [Online; accessed 11-August-2018].

[53] "Zapier: Connect your apps and automate workflows," https://zapier.com/, [Online; accessed 11-August-2018].

[54] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga, "A survey of intrusion detection in Internet of Things," *Journal of Network and Computer Applications*, 2017.

APPENDIX

A. IoT Policies

We present the description of the general IoT policies in Table I, trigger-action platform-specific policies in Table II, and application-specific policies in Table III.

ID	Policy Description
G.1	An event handler must not change a device attribute to conflicting values on some control-flow path, e.g., the motion-active event handler must not turn on and turn off a switch in some branch.
G.2	An event handler must not change a device attribute to the same value multiple times on some control-flow path, e.g., the motion-active event handler must not turn on the switch multiple times in some branch.
G.3	An event handler of an app must not change a device attribute to a value which is used as an event in the event handler of another app, e.g., door-lock event handler must not turn on a switch which is used in an event handler of an app that locks the door.
G.4	Two or more event handlers must not change a device attribute to conflicting values, e.g., a user-present event handler turns on the switch while a timer event handler turns off the switch at midnight. This is because the events of user presence and midnight may occur at the same time, leading to a race condition.

TABLE I: Description of general policies.

ID	Policy Description
S.1	Integrity Violation: An untrusted action changes a trusted attribute (untrusted email turns on the light)
S.2	Confidentiality Violation: An action changes an attribute that makes the private information publicly available (when an unlock function posts the user's location to a public log)

TABLE II: Description of trigger action-specific policies.

ID†	Policy Description
R.1	The door must be locked when a user is not present at home or sleeping.
R.2	The lights (in a bedroom, hallway, etc.) must be turned on if the motion sensor is active.
R.3	When there is smoke, the lights must be on if it is night, and the door must be unlocked.
R.4	The light must be on when the user arrives home.
R.5	The camera controlled doors must be closed when the door is clear of any objects.
R.6	The garage door must be open when people arrive home, and it must be closed when people leave home.
R.7	The location beacon must be inside a geo-fence around the home (defined by a user) to turn on the lights and open the garage door.
R.8	The lights must be turned off when the sleep sensor detects a user is sleeping.
R.9	The security system must not be disarmed when the user is not at home.
R.10	The alarm must sound when there is smoke or CO; and when an unexpected motion, tampering, and entering occurs.
R.11	The valve must be closed when water sensor is wet and when the water level threshold specified by a user is reached.
R.12	The devices (e.g., light switches, music player, cleaning supply cabinets, medicine drawers, or gun cases) must not be open or turned on when the user is not at home or sleeping.
R.13	Some device functionality (e.g., coffee machine starting brewing, heating up dinner in a crock-pot, turning on AC and heater) must not be used when the user is not at home or must be turned on before a time specified by a user.
R.14	The refrigerator, alarm, and security system must not be disabled, and their use must not be restricted to save energy.
R.15	The temperature value including idle energy savings must be set to the operating mode values as specified by the user (heating and cooling values are separate) based on the specific event.
R.16	The thermostat temperature (heating and cooling) entered by the user must be changed when the mode selected by a user is changed (e.g., from sleeping mode to away mode).
R.17	The AC and heater must not be on at the same time.
R.18	The HVACs, fans, switches, heaters, dehumidifiers must be off when the humidity and temperature values are out of the threshold specified by the user (e.g., a particular degree above/below the threshold of temperature and humidity).
R.19	The AC must be on when a user is within a specified distance of the house or at a time specified by the user.
R.20	The security camera must take pictures when there is a motion, and contact/door sensors are active.
R.21	The security camera must take a photo and sound alarm when the doors/windows are opening, and when the doors are unlocking at user-specified times. It must turn off all alarm when one alarm is turned off.
R.22	The battery level of the devices (switch, humidity sensor, etc.) must not be below a specified threshold.
R.23	The door must not be unlocked when a camera does not recognize an unauthorized face.
R.24	The windows must not be open when the heater is on.
R.25	The bell must not chime when the door is open.
R.26	The alarm must go off when the main door is left open for too long (specified by the user).
R.27	The mode must be set to "home" when the user is present at home, and "away" when the user is not present at home.
R.28	The sound system must read (e.g., the day's weather forecast and the status of the devices) with the user interaction and must not read at the time not specified by the user (guards against violations when the sleeping mode is on and when the user is not home.)
R.29	The sprinkler system must not be on when it rains, and when the soil moisture is below a threshold defined by a user. Flood sensor must activate the alarm when there is water.
R.30	The water valve must shut off when water/moisture sensor detects leak around a location such as basement and laundry room.

† We define app-specific policies based on the access granted to the devices in an app. For instance, R.22 is separately defined for an app that grants access to a switch and a humidity sensor.

TABLE III: Description of application-specific policies.