Discovering Physical Interaction Vulnerabilities in IoT Deployments

Muslum Ozgur Ozmen
Purdue University
mozmen@purdue.edu

Xuansong Li*
Nanjing University of Science and Technology
lixs@njjust.edu.cn

Z. Berkay Celik
Purdue University
zcelik@purdue.edu

Bardh Hoxha
Toyota Research Institute of North America
bardh.hoxha@toyota.com

Andrew Chun-An Chu
Purdue University
chu108@purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

ABSTRACT

Internet of Things (IoT) applications drive the behavior of IoT deployments according to installed sensors and actuators. It has recently been shown that IoT deployments are vulnerable to physical interactions, caused by design flaws or malicious intent, that can have severe physical consequences. Yet, extant approaches to securing IoT do not translate the app source code into its physical behavior to evaluate physical interactions. Thus, IoT consumers and markets do not possess the capability to assess the safety and security risks these interactions present. In this paper, we introduce the IoTSeer security service for IoT deployments, which uncovers undesired states caused by physical interactions. IoTSeer operates in four phases: (1) translation of each actuation command and sensor event in an app source code into a hybrid I/O automaton that defines an app’s physical behavior, (2) combining apps in a novel composite automaton that represents the joint physical behavior of interacting apps, (3) applying grid-based testing and falsification to validate whether an IoT deployment conforms to desired physical interaction policies, and (4) identification of the root cause of policy violations and proposing patches that guide users to prevent them. We use IoTSeer in an actual house with 13 actuators and six sensors with 37 apps and demonstrate its effectiveness and performance.

CCS CONCEPTS

• Security and privacy → Formal methods and theory of security; Vulnerability scanners;
• Computer systems organization → Sensors and actuators;

KEYWORDS

IoT; Security Analysis; Physical Interaction Vulnerabilities

1 INTRODUCTION

An IoT deployment includes a set of IoT devices and applications (apps). IoT devices are comprised of two components: actuators and sensors. IoT actuators influence physical channels by executing an actuation command, e.g., heater-on command changes the temperature. IoT sensors measure physical channels and generate sensor events, e.g., a sound sensor measures ambient sound and generates a sound-detected event. IoT’s most attractive feature is support for custom automation in the form of apps. Apps are event-driven. They subscribe to sensor events, and when the event happens, they invoke an event handler method to activate actuators.

With the growing number of apps and devices co-installed in an IoT deployment, the interactions among apps cause increasing safety and security violations [14, 60, 63]. There are two fundamental sources of app interactions: software and physical. Software interactions occur when IoT apps interact through a common device in apps’ source code. Consider the scenario where an app that turns on the lights when smoke is detected and an app that locks the door when lights are turned on are co-deployed. These apps interact through a common light device defined in their source code (light-on, light-on, door-locked), which puts users at risk during a fire. Physical interactions are another notable threat: an app invokes an actuation command, a sensor detects the physical channel influenced by this command, which triggers other apps subscribed to the sensor event. For example, an app that turns on a heater interacts with an app that opens the window when temperature exceeds a threshold. These apps interact through the temperature physical channel (heater-on, temperature, window-open), which may allow a burglar to break into the house. These interactions can be caused by design flaws from users and can be leveraged by an adversary to make an IoT deployment reach an undesired state.

Recent efforts mainly focus on identifying software interactions via app source code analysis [7, 8, 12–14, 45, 56]. These approaches find interacting apps by matching the common device attributes in multiple apps, such as the light-on attribute in the first example. Thus, they cannot detect the physical interactions among apps, such as the interaction through the temperature channel above.

There have been limited efforts to discover physical interactions, which is the focus of this paper. These works (1) use manual interaction rule templates to define relations between actuation commands and sensor events [3], and (2) leverage NLP and device behavioral models to map apps’ events and commands [10, 21, 60]. For example, a manually crafted rule maps the heater-on command in the app source code to the temperature channel. A model for the heater-on command defines the temperature increases by 1°C in 8 hours. Yet, these rules and models have limited expressiveness of physical channels, and ultimately fail to identify such interactions. Specifically, these approaches lead to over-approximation of physical channels, which are false alarms (e.g., the system flags that the temperature from oven-on opens the window, yet the oven...
We repeated the experiments in the actual house and verified that platforms such as IFTTT [35], Zapier [62], and Apiant [6]. These faces. Another trend to create custom automation is trigger-action install official apps from IoT markets such as HomeKit [33] and light to events (e.g., motion that monitor and control sensors and actuators. Apps subscribe IoT Deployments. An IoT deployment is composed of a set of apps through static analysis and translates each into a hybrid I/O automaton to define their physical behavior. From this, IoTSeer unifies the automata of apps in a composite automaton to represent the joint physical behavior of interacting apps. IoTSeer then systematically develops a set of physical IoT policies through metric temporal logic to express the physical interactions that cause undesired system states, and uses grid-based testing and falsification to validate that IoT apps conform to those policies. Lastly, IoTSeer determines the root cause of the physical interactions and presents a warning report that guides users on how to fix them.

We used IoTSeer in an actual house with 13 actuators and six sensors, automated by 37 apps to reveal how physical interaction policies are violated. We built the hybrid I/O automata of 26 actuation commands and six sensor events defined in the source code of apps. IoTSeer found 11 unique physical interaction policy violations on the composite automata of different groups of interacting apps. We repeated the experiments in the actual house and verified that all violations reported by IoTSeer are true positives. IoTSeer is adaptive to the newly added, removed, and updated devices and imposes minimal automatization construction and validation time overhead. It takes on average 21 secs to falsify and 69 secs to verify a physical interaction policy on the composite automaton of four interacting apps. In this paper, we make the following contributions:

- **Translating IoT App Source Code into its Physical Behavior:** We translate the actuation commands and sensor events in the app source code into hybrid I/O automata to define their physical behavior.
- **Composition of Interacting Apps:** We introduce a composite model architecture that defines the joint physical behavior of interacting apps.
- **Policies for Physical Interactions:** We develop policies to identify five types of physical interactions due to the design flaws from benign users and malicious intent to cause undesired states.
- **Evaluation in an Actual House:** We use IoTSeer in a real house that contains 13 actuators and six sensors and expose 11 physical interaction policy violations.

### 2 BACKGROUND

**IoT Deployments.** An IoT deployment is composed of a set of apps that monitor and control sensors and actuators. Apps subscribe to events (e.g., motion-detected) that invoke their event handler methods, which trigger actuation commands (e.g., light-on). Users install official apps from IoT markets such as HomeKit [33] and OpenHAB [46] and third-party apps through proprietary web interfaces. Another trend to custom automation is trigger-action platforms such as IFTTT [35], Zapier [62], and Apiant [6]. These platforms allow users to use conditional statements in the form of if/then rules for integrating digital services with IoT devices. For instance, a user can link her IoT platform and email accounts to a trigger-action platform to turn off the lights and send an email when the door is unlocked. In this paper, we use the term app(s) to refer to both IoT apps and trigger-action rules.

**Definitions.** Actuators manipulate a physical channel through an actuation command, e.g., a heater’s heater-on command changes the environment’s temperature. Sensors measure physical changes in an environment (e.g., sound), and generate sensor events based on these changes, e.g., sound-detected. We characterize the physical relations between an app’s actuation commands and an app’s sensor events with four properties: (1) influence type, (2) time, (3) distance, and (4) dependency. Influence defines the physical channels an actuation command changes, and a sensor event observes. An actuation command may influence a single or multiple channels, e.g., a clothes-dryer-on command introduces sound, heat, and vibration. A sensor measures the influence from single or multiple actuation commands, e.g., an illuminance sensor measures the sum of (aggregated) illuminance from TV-on and light-bulb-on. Time defines a command’s instant or continuous influence, e.g., heater-on gradually increases temperature, and bulb-on immediately changes illuminance. Distance quantifies an actuation command’s influence on a physical channel at different locations. A command’s influence on a sensor often decreases when the distance between sensors and actuators increases. Lastly, dependency defines the implicit relation among distinct physical channels, where a change in a physical channel affects another channel. For instance, variations in ambient temperature change the environment’s humidity.

### 3 MOTIVATION AND THREAT MODEL

**Problem Definition.** In an IoT deployment, when an app invokes an actuation command, it influences a set of physical channels measured by sensors, which interacts with other apps subscribed to those sensor events, invoking other commands. An adversary can provide users with apps operating correctly in isolation yet exploit the unintended interactions among apps to cause unsafe states.

Figure 1 depicts an IoT deployment that includes seven actuators and three sensors controlled by four apps App1, App2, App3, and App4. An adversary supplies two apps, MalApp1 and MalApp2, to create unintended physical interactions. In the first scenario (Figure 1-a), the user sets the robot vacuum cleaner to operate at midnight through App1. When
Another user confirms the second scenario saying “...do not put thermostats in a kitchen or laundry room either, the extra warmth from the appliances slow heating or cooling to other rooms” [25].

### 3.1 Threat Model

We consider design flaws (from benign users) and malicious intent (from adversaries) that cause unsafe or undesired states in an IoT deployment due to the physical interactions among IoT apps.

For the design flaws, a user installs a set of apps in an IoT deployment. We assume the apps are vetted by IoT platforms before distribution and operate correctly in isolation. Yet, the physical interactions subvert the intended use of IoT devices, leading to undesired states. The interactions may happen due to errors in users’ creation, installation, and configuration of apps. This is because IoT users are usually uninformed about the implications of the app interactions, as demonstrated by prior work [57, 65].

For malicious cases, an adversary intentionally provides apps to a user that create physical interactions to execute privileged actions, such as unlocking the door. The adversary’s goal is to stealthily exploit the physical interactions among apps. For instance, an adversary that supplies a malicious app that unlocks the patio door at night is not stealthy. This is because the app can be rejected from the IoT app market during the platform’s security vetting, and the users may recognize the malicious behavior by manually checking the app’s events and commands. In contrast, a stealthy adversary may provide single or multiple apps innocuous in isolation such as MalApp1 to gain physical entry to the house when this app interacts with App1, the motion channel. An adversary may cause such hidden interactions through the following scenarios: (1) tricks the user to install malicious apps via phishing and other social engineering methods, (2) distributes malicious apps through IoT forums, and (3) discovers a physical interaction between a user’s apps and exploits the interaction by manipulating the response returned from an external service to trigger apps.

---

**Figure 2: Physical interaction vulnerabilities: (a)-(c) results in undesired states and (d)-(e) delays or cancels the desired states.**

the vacuum cleaner roams around the home, its movements activate the motion sensor. The motion-active event triggers MalApp1, that turns on the lights and unlocks the patio door at midnight. In the second scenario (Figure 1-b), App2 turns on the pressure cooker and activates the cooking-mode at a user-defined time. MalApp2 turns on the oven and exhaust fan when the mode is set to cooking. The oven-on and cooker-on commands both increase the temperature. The increase in temperature triggers App3, that opens the window. Here, oven-on and cooker-on’s individual influence does not trigger App3, yet their aggregated impact on temperature causes an unexpected interaction among App3, MalApp2 and App1. Lastly, the sound from the exhaust fan activated by MalApp2 triggers App3 through the sound-detected event, sending an “intruder alert” message.

The preceding scenarios demonstrate that the final environmental states depend not just on individual apps but are a result of the unified physical behavior of IoT devices controlled by multiple apps. In this, each app is individually safe, yet their unified physical interactions leave users at risk. For instance, the door gets unlocked at midnight, robot-vacuum-on (App1) → motion-active (MalApp1) → door-unlock (MalApp1), which allows the adversary to break into the house, and the erroneous intruder alert, fan-on (MalApp2) → sound-detected (App4) → send-intruder-msg (App4), causes panic and unnecessarily brings resources (e.g., police dispatch) to the home.

**IoT Forum Study.** To investigate whether users run into unintended physical interactions in real IoT deployments, we studied 100 popular posts in IoT community forums including Apple HomeKit [34], Samsung SmartThings [54] official forums, and smart home and home automation subreddits [50]. We found that users encounter similar problems due to the interacting apps. For instance, a Reddit user confirms the first scenario stating, “I put a motion sensor in my living room, the lights will turn on. Well, silly me did not realize that my Roomba is set to clean every night at 3 am would trigger the sensor and keep the lights on until it is finished” [26]. Another user confirms the second scenario saying “...do not put thermostats in a kitchen or laundry room either, the extra warmth from the appliances slow heating or cooling to other rooms” [25].

### 4 PHYSICAL INTERACTION VULNERABILITIES

We introduce five types of physical interaction vulnerabilities among apps that could be leveraged by an adversary to make an IoT deployment reach an unsafe state. To properly designate the circumstances under which the app interactions are a vulnerability or feature, we define two labels, intended (Int) and unintended (unInt).

- **Intended Physical Channel** (Int): An actuation command’s influence on a physical channel is “intended” if the use of the actuation command is to change that physical channel.
- **Unintended Physical Channel** (unInt): An actuation command’s influence on a physical channel is “unintended” if it is unplanned by a system or undesired by a user.

To illustrate, if the heater-on command’s Int labeled influence on temperature triggers apps, it is a feature; but, if clothes-dryer-on’s UnInt labeled influence on the sound, temperature, and motion triggers apps, it is a vulnerability. We note that the physical channel labels might be highly contextual depending on the use cases of devices and the environment in which the devices are deployed. For instance, consider an app that turns on a heater interacts with another app that opens the window when the room temperature exceeds a threshold. The correctness of opening the window through heater-on’s influence on the temperature channel might be deemed
safe or dangerous based on the use cases of devices. As needs dictate, we allow users to label physical channels as Int or UnInt.

Figure 2 depicts the vulnerabilities: (a)-(c) causes undesired states, and (d)-(e) delays or cancels desired states. To systematically illustrate the vulnerabilities, we abstract an app’s source code as a labeled transition system, which is a tuple $TS = (\Delta, L, \rightarrow)$. $\Delta$ is a set of actuation commands (A) and sensor events (S). L is a set of physical channel labels, $\{\text{Int, UnInt}\}$ and $\{\text{Inc, Dec}\}$. Inc and Dec represent if a command increases or decreases a physical channel. $\rightarrow$ defines the labeled physical channels between apps. Lastly, we define a transition $Agg(A_1, \ldots, A_n) \rightarrow S$ to represent the aggregated influence of multiple commands that triggers a sensor event.

**Unintended Implicit Trigger (Figure 2a).** An app’s actuation command introduces an unintended physical channel detected by a sensor. The consequent sensor event triggers other apps that invoke further actuation commands. Formally, two apps interact due to the $A_{\text{UnInt}} \rightarrow S$ transitions. Two examples of this interaction are seen in Figure 1. (1) When App 1 turns on the robot vacuum, it causes the robot-vacuum-on $\rightarrow$ motion-active transition that triggers the MalApp 1, unlocking the patio door. (2) When MalApp 2 turns on the fan, it causes the exhaust-fan-on $\rightarrow$ sound-detected transition, which triggers App 3, sending an intruder alert notification.

**Unintended Aggregation Trigger (Figure 2b).** An actuation command’s influence on a physical channel does not trigger a sensor event; however, the aggregation of unintended influences from multiple actuation commands triggers a sensor event. Formally, multiple apps interact with each other due to the $Agg(A_1,\ldots,A_n) \rightarrow S$ transitions. In Scenario 2 shown in Figure 1-b, App 2 and MalApp 2 cause the $Agg(\{\text{cooker-on, oven-on}\}) \rightarrow \text{temp > threshold transition}$, which triggers App 3, opening the window.

**Aggregation Bypass (Figure 2c).** The aggregation of intended and unintended influences from multiple commands triggers an app while an actuation command’s intended influence itself does not. Formally, this interaction occurs due to $A_{\text{IntUnInt}} \rightarrow S$ transition, yet $A_{\text{UnInt}} \rightarrow S$ s.t. $\exists k \leq n$ transition does not happen. To illustrate, consider a smart bulb, a TV and an illuminance sensor are co-located. Here, an app’s bulb-on command does not create the bulb-on $\rightarrow$ light-detected transition, i.e., the bulb’s luminescence does not trigger the light-detected event itself. Yet, if an app’s TV-on command is also invoked, the aggregated luminescence from bulb-on and TV-on causes the $Agg(\{\text{bulb-on, on TV}\}) \rightarrow \text{light-detected transition}$. Thus, the sensor event only occurs when the UnInt illuminance contributes to the sensor measurements.

**Aggregation Conflict (Figure 2d).** Multiple actuation commands influence an intended physical channel in conflicting ways. That is, an app’s actuation command increases an Int physical channel while another app invokes an actuation command that decreases the same physical channel. This conflicting impact on a physical channel may cause triggering other apps to take longer or to be late, and delay or prevent intended states. Formally, these interactions occur due to the $(A_{\text{Int} \rightarrow S}, A_{\text{Dec} \rightarrow S})$ transitions. For instance, if the AC and heater are turned on at the same time, heater-on increases temperature, whereas AC-on decreases it. This delays or prevents triggering other apps conditioned on the temperature-change event.

**Dependency Conflict (Figure 2e).** Apps also interact due to the dependency between physical channels. While two actuation commands $(A_1, A_2)$ have Int influences on two distinct physical channels, they cause a conflicting impact on a channel and its sensor events. That is, $A_1$ increases $p_1$ and $A_2$ increases $p_2$, yet $A_2$ decreases $p_1$ due to the dependency between $p_1$ and $p_2$. Formally, this interaction occurs due to the $A_{\text{Int} \rightarrow S}, A_{\text{Dec} \rightarrow S}$, and $A_{\text{Int} \rightarrow S}$ transitions. For instance, heater-on’s Int influence on temperature impacts humidity due to their dependency (Detailed in Section 5.1.3). If a heater and humidifier are both on, the heater decreases humidity while increasing temperature and the humidifier increases humidity. Similar to the aggregation conflict, this delays or prevents triggering the apps conditioned on the humidity channel’s events.

**5 IOTSeer**

Figure 3 provides an overview of the six stages of IOTSeer. IOTSeer first extracts an app’s actuation commands and sensor events from its source code through static analysis (i). From this, IOTSeer maps an actuation command to a hybrid I/O automaton for each physical channel it influences, and a sensor event to a hybrid automaton for the channel it measures (j). Each automaton translates the high-level commands and events in the source code into their physical behavior through well-studied differential and algebraic equations. IOTSeer then extends system identification to tune automata parameters using device traces collected from actual IoT devices for precision. For instance, Figure 4 presents the hybrid automaton of App 1’s robot-vacuum-on command’s influence on motion (Left) and MalApp 2’s motion-active sensor event (Right).

IOTSeer next unifies each app’s automaton in a composite automaton to represent the interacting apps’ joint physical behavior (i). IOTSeer’s composition algorithm takes into account complex physical properties, including aggregation and dependency of physical channels. For instance, Figure 5 presents the composite automaton of three apps, App 2, MalApp 2, illustrated in Figure 1-b. Thereafter, IOTSeer integrates conformance algorithms into the composite automaton to account for environmental factors, such as room layouts and the presence of human inhabitants. This process ensures that a composite automaton has the correct functional properties of a real IoT deployment amenable to formal analysis.

We develop a set of policies with metric temporal logic to detect the physical interaction vulnerabilities introduced in Section 4 (i). For instance, a policy for unintended implicit trigger states that “a motion-active event from robot-vacuum-on must not unlock the door”. IOTSeer validates whether the apps’ joint behavior conforms to the identified policies via grid-based testing for exhaustiveness and optimization-guided falsification for scalability (q). If IOTSeer flags a policy violation, it reports the root cause, including the interacting apps, their actuation commands, sensor events, and their physical channel. IOTSeer then proposes patches that guide
users to prevent the physical interactions, such as removal of apps, activating apps in isolation, and new device placement (1).

5.1 From Source Code to Physical Behavior

5.1.1 Extracting Apps’ Commands and Events. IoTSeer extracts an app’s actuation commands and sensor events from its source code. IoTSeer integrates static analysis tools designed for IoT apps [12, 13, 68] to support various IoT programming platform apps. These platforms are diverse, and each offers a different programming language to automate IoT devices. For instance, consumer IoT platforms such as OpenHAB enable users to write apps with a Domain Specific Language based on Xbase [46], and trigger-action platforms such as IFTTT and Zapier implement if-then abstractions through a set of APIs for the ease of users [35, 62]. Common to all, they model an app’s life-cycle, including its entry points and event handlers from an app’s inter-procedural control flow graph (ICFG) and extract the (1) devices and events, and (2) activations invoked for each event (often in the event handlers). To illustrate, the description of the MalApp’s IFTTT app in Figure 1-a says “Turn on the light and open the patio door when motion is detected”. IoTSeer obtains light-on and door-unlock commands, and the motion-active event that triggers these commands via static analysis.

We studied three IoT programming platform markets, IFTTT, Microsoft Flow, and OpenHAB, and obtained the popular IoT apps. We use the actuation commands and sensor events of these apps to define their physical behaviors in the next subsection. Through this process, we extracted 13 different commands such as heater-on, clothes-dryer-on, robot-vacuum-on. The actuation commands influence six physical channels, temperature, humidity, illumination, sound, motion, and smoke, measured by six different sensors.

5.1.2 Describing the Physical Behavior of Apps. IoTSeer maps each actuation command and sensor event extracted from an app’s source code to hybrid I/O automata to describe an app’s physical behavior. We extend physics-based modeling to construct a separate automaton for each physical channel an actuation command influences, and a sensor event observes. We then use system identification (SI) to tune automata parameters [37, 48].

The physics-based modeling integrates a generic differential or algebraic equation from control theory into an automaton to model each command and event’s physical behavior. In the SI stage, we take each automaton constructed at the physics-based modeling and use limited data traces from IoT devices to infer the deviations between automata and actual devices. We then tune each automaton’s parameters to address the discrepancies among them.

This approach enables unique advantages over past efforts that solely use input-output data to determine a system’s empirical description. These efforts provide a set of inputs to a real system, acquire raw data traces, and construct a system’s dynamics from them [15, 17, 20, 49, 64]. Yet, SI solely built on raw data traces has two main limitations in IoT deployments. First, they require an extensive amount of data traces for each device to correctly identify their physical behavior [37]. For instance, they need temperature values when an oven is turned on at different temperatures for different periods of time and distances. Second, they repeat the comprehensive data collection for each separate IoT deployment. This is because environmental factors and device types differ in each deployment [16, 51]. We address the first limitation by constructing generic physics-based automata and tuning their parameters through SI, which requires a minimal amount of data traces. For the second limitation, we build configurable automata that take device parameters as an input. This allows us to define generic automata templates for devices that change the same physical channels.

Actuation Command Hybrid I/O Automaton. IoTSeer builds a hybrid I/O automaton for each physical channel an actuation command changes. To identify the physical channels, we deployed 13 actuators and 6 sensors in a house, invoked each actuation command, and recorded the sensor readings. We repeated the experiments with different distances between sensors and actuators to observe each command’s impact. Through these experiments, we created a physical-channel reference file that includes the complete set of channels each actuation command changes. IoTSeer uses this file to map each actuation command in the source code to single or multiple automata. For example, the clothes-dryer-on actuation command changes sound, humidity, temperature, and motion. Thus, it is associated with a separate automaton for each physical channel.

An automaton models the discrete and continuous dynamics of an actuation command. The discrete behavior is a finite state machine for invoking the command from the app source code. The continuous behavior is an algebraic or differential equation for its physical behavior. Formally, a hybrid I/O automaton [41] is a tuple $H_a = (Q, X, f, →, U, O)$, where $Q$ is a set of discrete states, $X$ is a set of continuous variables, $f$ is a flow function that defines the evolution of continuous variables in each state, $(→)$ defines the discrete transitions, and $U/O$ defines the input/output variables. We define the discrete states as $Q = \{\text{on, off}\}$, and discrete transitions enable switching between them. We initialize each command automaton in the $\text{off}$ state. The automaton switches to the $\text{on}$ state when an actuation command is triggered in the app’s source code.

The flow functions govern the physical behavior of actuation commands. They are in the form of differential equations for continuous physical channels such as temperature and algebraic equations.
for instant channels such as sound. A flow function takes two continuous variables (parameters) as input: device configuration and distance from the actuator. The device configuration parameter describes the characteristics of a device, such as its operating power. This enables us to use the same flow function for different actuators influencing the same channel (e.g., oven-on and cooker-on), and the actuators with multiple working patterns (e.g., 

This makes IoTSeer practical against dynamic device placement changes in the deployment, and enables effortless porting of IoTSeer to other IoT deployments with different device placements.

**Example.** We constructed 26 hybrid I/O automata of 13 actuation commands (See Section 6). Figure 6 illustrates the automata of clothes-dryer-on command that changes sound and temperature.

First, we associate an app’s clothes-dryer-on command with apps subscribed to currentTemp and sound-detected events. We construct an automaton for each physical channel. Figure 6 (Top) depicts a sample output of the temperature automaton, which uses a flow function of the partial differential heat diffusion equation [31]. The automaton’s output is the dryer’s influence on environment temperature over time per distance.

Second, the dryer-on’s sound automaton uses a flow function with an algebraic equation based on the inverse square law [59]. Given the dryer’s sound pressure parameter at a fixed distance, the automaton outputs the sound pressure (dB) at arbitrary distances.

**Sensor Event Hybrid I/O Automaton.** A sensor event’s hybrid I/O automaton is similar to actuation commands, yet it only measures physical channels. We define a sensor event’s automaton (H_s) with a single state, Q = {on}, and a timed self-loop transition on \( \cdot \) on, where time t is the frequency a sensor samples its readings.

The sensor event automaton takes a sensitivity-level parameter, which defines the minimum amount of change in the physical channel (threshold) required for a sensor to change its numerical or binary measurement. A threshold function outputs a sensor reading indicating whether the physical channel level is equal or greater than the sensitivity level. If the sensor measures Boolean-typed values such as motion, the automaton outputs a binary bit that indicates “motion-detected” or “motion-undetected” events. If the sensor makes numerical readings such as temperature, the automaton outputs the numerical value. The sensor event automata’s output is then used to trigger other apps subscribed to those events.

**Example.** We constructed hybrid I/O automata for 6 sensors, temperature, humidity, illuminance, sound, motion, and smoke (See Section 6). Figure 6 depicts the automaton output of the temperature sensor’s currentTemp and the sound sensor’s sound-detected events.

The temperature sensor event automaton monitors the temperature based on its sensitivity level and outputs numerical temperature values. For instance, a temperature sensor with a sensitivity level of \(\pm 1.8^\circ F\) and a sampling rate of 5 secs at 2 m away from the dryer outputs currentTemp \(= 88.6^\circ F\) event when the clothes-dryer-on command increases the ambient temperature \(78.8^\circ F\) by 2°F. The sensor event automaton detects this change as it exceeds the sensor’s sensitivity level and triggers the apps that are conditioned on this change, e.g., currentTemp \(= 80.6^\circ F\) and currentTemp \(> 80^\circ F\).

Similarly, the sound sensor automaton outputs a binary value when the dryer’s sound exceeds its threshold. It then sends a sound-detected event to the apps subscribed to this event.

**System Identification (SI) for Tuning Automata Parameters.** SI is a learning-based method commonly used by control engineers to estimate parameters or models of physical processes using experimental data traces [37, 48]. We use SI to ensure that the device automata constructed through physics-based models have high conformance with actual devices in an IoT deployment. Each automaton requires a set of configuration parameters that affect how the devices influence or measure the physical channels. We initially determine these parameters from the device datasheets. However, a discrepancy in parameters could occur, for example, due to an error in the device sheets or device aging [51].

To address such discrepancies, we extend \((\tau, \varepsilon)\)-closeness [2] that measures the conformance of automata with actual device traces. We then use SI to estimate the automata parameters to maximize conformance. This process requires a fewer amount of traces than SI’s traditional usage as we only tune the configuration parameters instead of estimating the complete equations for each device.

\((\tau, \varepsilon)\)-closeness determines the difference among two traces in their timing \((\tau)\) and states \((\varepsilon)\), where \(\varepsilon\) is referred to as deviation score [2]. Let \(x\) be an automaton’s traces, and \(y\) be the real device traces generated with the same inputs. Given \(T \in \mathbb{R}_+, \text{ and } (\tau, \varepsilon) \geq 0\), we determine \(x\) and \(y\) are \((\tau, \varepsilon)\)-close if for all \(t \leq T\), there exists \(s \in y\) where \(|t - s| \leq \varepsilon\) and \(|x(t) - y(s)| \leq \varepsilon\), and for all \(t \in y\), \(s \in x\) where \(|t - s| \leq \tau\) and \(|y(t) - x(s)| \leq \varepsilon\).

Given an IoT deployment, we first select an actuation command-sensor pair where the sensor measures the actuation command’s influence. We collect sensor traces with different distances between the actuator and sensor (0.5 - 2.5 m with 0.5 m intervals), and compute the \((\tau, \varepsilon)\)-closeness between the actual device and automata traces. For instance, consider the oven-on automaton trace yields \(t = 19.6, v = 78.8^\circ F\), and actual device trace yields \(t = 23.8, v = 78.8^\circ F\), where \(v\) is the sensor output at time \(t\). IoTSeer’s conformance algorithm outputs that the automaton and actual device traces differ in time \((\tau)\) by 4.2 with a deviation score \((\varepsilon)\) of 0. If an actuator-sensor pair yields a high deviation score, we run automata with a grid of configuration parameters and conduct a binary search to obtain the optimal values that minimize the deviation score. In Section 6.1, we present our detailed numerical results in an actual house.

5.1.3 **Combining the Physical Behavior of Apps.** IoTSeer builds a separate composite automaton representing the joint physical
behavior of a set of interacting apps. The separate composite automaton allows IoTSeer to perform scalable analysis of many apps attached to many devices and receiving many events.

Algorithm 1 presents our approach to composite automaton construction. The algorithm starts with identifying the interacting apps by matching the physical channels of the sensor events and actuation commands. First, if a sensor measures a physical channel that an actuation command influences, we add a transition from the actuation command automaton \( (h_i) \) output to the sensor event automaton \( (h_j) \) input (Lines 2-4). Second, if the actuation command’s influence on a channel exceeds the sensor event’s sensitivity, it triggers the event handler of apps subscribed to the sensor event. In this case, we add a transition from the sensor event automaton \( (h_j) \) output to the actuation command automaton \( (h_i) \) input (Lines 5-10). The transitions are expressed with a UNIFY(\( H_i, H_j \)) operator, which defines the interactions as a labeled transition, \( H_i \mapsto H_j \), and \( H_i \mapsto H_j \). Here, \( L = \{ \text{UNInt, Int} \} \), \( \mapsto \) is an actuation command’s labeled influence on a physical channel, and \( \mapsto \) is a sensor event that triggers actuation commands in an app’s event handler method.

Figure 5 illustrates the composite automaton of four apps in Figure 1-b. Here, when MalApp, invokes oven-on and exhaust-fan-on commands, IoTSeer identifies that smart-oven-on’s temperature automation interacts with the temperature sensor event in App2 and exhaust-fan-on’s sound automaton interacts with App4. The following transitions are then added to the composite automaton:

- \( H_2 \text{ (smart-oven-on: MalApp2) \quad UNInt-temp} H_2 \text{ (temp > threshold: App2) \quad UNInt} \)
- \( H_2 \text{ (exhaust-fan-on: MalApp2) \quad UNInt-sound} H_2 \text{ (sound-detected: App4) \quad UNInt} \)

Similarly, new transitions are added when (1) the temperature sensor measures an increased temperature, App2 triggers a sensor event that opens the window, and (2) the sound sensor detects sound, App4 triggers a sensor event that sends intruder alert message:

- \( H_2 \text{ (temp > threshold: App2) \quad UNInt} H_2 \text{ (window-open: App3) \quad UNInt} \)
- \( H_2 \text{ (sound-detected: App4) \quad UNInt-sound} H_2 \text{ (send-intruder-msg: App3) \quad UNInt} \)

Physical Channel Aggregation. A sensor measures the accumulated influence of multiple actuation commands on a physical channel. For this, we define an aggregation operator \( \text{AGG} \). AGG combines the UNIFY(\( H_i, H_j \)) operators where a sensor event automaton takes the aggregated output of the actuation command automata as input (Lines 11-13). Turning to App1, that turns on the pressure cooker and MalApp2 that turns on the oven, a transition is added to the composite automaton, \( \text{AGG} (h_2^1 \text{ (cooker-on: App1), } h_2^2 \text{ (oven-on: MalApp2)}) \).
besides humans (e.g., plants). Similar to addressing uncertainty in security risk assessments, we add noise automata to the composite automaton. Each noise automaton outputs a sample from a given mean and standard deviation in each time step. Its output is then aggregated with the physical channel values through the AGG operator. The noise parameters for mechanical devices can be estimated by activating them and recording the changes in the sensor measurements. For uncertain factors, the parameters are manually set based on the environment’s needs, high-values to over-approximate the physical channel values, and low-values to under-approximate.

5.2 Security Analysis of an IoT Deployment

We develop a security analyzer that evaluates an IoT deployment’s apps against a set of physical interaction policies (Section 5.2.1). The policies are validated on the composite automaton of interacting apps (Section 5.2.2). If a violation is flagged, the security analyzer identifies its root cause and presents a warning report that guides users on preventing the interactions causing the undesired states.

5.2.1 Policy Identification. The policies define the physical interactions that apps must satisfy for an IoT deployment to be considered safe. We introduce five policies to evaluate an IoT deployment against physical interaction vulnerabilities introduced in Section 4.

We formally represent the policies with Metric Temporal Logic (MTL) [1, 38]. In contrast to Linear Temporal Logic (LTL) [47] and Computation Tree Logic (CTL) [18] that enable reasoning over event ordering, MTL expresses both temporal and causal relations. This enables us to reason about the physical behavior of apps and identify their interactions with state and time constraints [24].

Table 1 presents five policy templates for each physical interaction vulnerability depicted in Figure 2: (1) an individual (1), (2) three aggregation (G1, G2, G3), and (3) a dependency (D). For instance, the individual policy (1) template states that an actuation command’s unintended influence on a physical channel must not trigger a sensor event. To illustrate, the sound produced by garbage-disposal-on command must not trigger the sound-detected event in apps within t minutes. Its MTL policy is $\square_{[0,t]}(\text{Imp}(\text{Gar-disp-on, sound})) < \text{th}$; $\square_{[0,t]}$ means always within t time units and Imp() denotes the labeled (Int/UnInt) influence on a physical channel.

5.2.2 Policy Validation. We use two separate approaches, grid-based testing, and optimization-guided falsification, to validate MTL policies on the composite automaton. Testing is exhaustive as it checks an automation’s behavior until a policy is satisfied. In contrast, falsification is scalable as it confirms an automation’s behavior

Table 1: Descriptions and MTL formulas of policy templates for physical interaction vulnerabilities.

| Category  | ID | Policy Description                                                                 | MTL Formula |
|-----------|----|-------------------------------------------------------------------------------------|-------------|
| Individual| 1  | Unintended physical channels from an actuation command must not trigger a sensor event | $\square_{[0,t]}(\text{Imp}(\text{UnInt}, p)) \leq \text{th}$ |
| Aggregation| 2  | Unintended physical channels must not aggregate with intended physical channels to trigger a sensor event unless intended physical channels alone can trigger the sensor event. | $\square_{[0,t]}(\text{Imp}(\text{Int}, p)) < s \land \text{Imp}(\text{Int}, p), (\text{UnInt}_{1}, p),...,(\text{UnInt}_{n}, p) \geq \text{th}$ |
|          | 3  | Multiple intended physical channels must not influence a sensor event in opposite ways. | $\square_{[0,t]}(\text{Imp}(\text{Int}, p), \text{Imp}(\text{Int}, q)) \land \square_{[0,t]}(\text{Imp}(\text{Int}, p), \text{Imp}(\text{Int}, q)))$ |
| Dependency| D  | An intended physical channel must not influence a sensor event on another channel in an opposite way due to the dependency between the physical channels. | $\square_{[0,t]}(\text{Imp}(\text{Int}, p), \text{DepImp}(\text{Int}, p), (p_1))$ |

\[ \text{Imp}(\text{Int}, p) \text{ and } \text{Imp}(\text{UnInt}, p) \text{ denote the intended (Int) and unintended (UnInt) influence on a physical channel } p \text{ (e.g., } p = \text{temp}). \text{ When multiple Int/UnInt channels are defined in } \text{Imp, it denotes the aggregated influence. } \text{th denotes the sensitivity level of a sensor. }\square \text{ is } \text{Imp}(\text{Imp}, \text{Imp}) \text{ denotes a binary function that outputs whether } \text{Imp}_1 \text{ and } \text{Imp}_2 \text{ introduce opposing influences on a physical channel. } (3) \text{ DepImp}(\text{Int}, p), (p_1) \text{ denotes the impact on } p_1 \text{ due to the dependency on } p_2. \]

5.2.2 Algorithm 2 Grid-based Testing

Input: Composite automaton (\(M_{\text{comp}}\)) with actuation command (\(H_a\)) and sensor event automata (\(H_s\)), parameters (\(x \text{ - distances among devices})

Output: \(P = (\text{inputs, apps, dist, stime, y})\)

1: function Grid_Test\(\left(H_a, H_s, x, y, M_{\text{comp}}, t, \psi\right)\)
2: for \(j \in H_a, H_s \subseteq \text{do}\)
3: for \(i \text{ do}\)
4: for Different start times of each actuator in \(H_a\)
5: if \(\Phi(M_{\text{comp}}, x, y) \neq \psi\) then
6: \(P \leftarrow P \cup \{x, y, \Phi(M_{\text{comp}}, x, y)\}\)
7: end if
8: end for
9: return \(P\)
10: end function

until a violation is observed. We use falsification if an IoT deployment includes many apps that lead to a large composite automaton, because it offers scalability at the expense of full verification. Both approaches report the root cause of the violations and present users with patches to prevent them.

Executing Composite Automaton. We execute the composite automaton and collect data traces to validate the MTL policies. The execution requires two parameters: time-step, the time between two consecutive recorded data trace points, and execution-time, the duration of trace collection (See Section 6 for parameter selection). The automata traces (\(v, t\)) are timed (\(t\)) state sequences of commands and sensor events (\(v\)). Each actuation command automaton’s state is its influence on a physical channel, and each sensor event automaton’s state is its measurements. The traces include labels (\(Int/UnInt\)) and app IDs of the commands and sensor events.

Grid-based Testing. Testing determines whether the composite automaton satisfies a policy, \(\Phi(M, x, u) = \psi\), where \(\Phi\) represents the behavior of the composite automaton (\(M\)) under a finite set of parameters (\(x \in X, \text{distances among devices}) and inputs (\(u \in U, \text{actuator start times})–the time that apps invoke actuation commands). Algorithm 2 presents our grid-based testing approach. We set the distance between actuators and sensors to fixed paths (e.g., robot vacuum and motion sensor) or constants (e.g., temperature sensor and heater). We then set actuators’ start times as a grid (\(t_0 : \Delta t : t_{\text{end}}\)) (Line 3). The algorithm executes the composite automaton with a grid-search on the actuators’ start times and validates a policy on each execution’s automata traces with a robustness metric (Lines 4-6). The robustness quantifies how close an MTL formula is to the policy violation. Positive robustness values indicate the policy is satisfied, and negative values indicate the policy is violated.
Table 2: Physical channels of studied actuators and sensors.

| Actuator (Actuation Command) | Temp | Hum | Sound | Light | Motion | Smoke |
|-----------------------------|------|-----|-------|-------|--------|-------|
| Murray Baseboard Heater (on) | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Whirlpool washer (wash)      | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Whirlpool washer (dry)       | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Sunbeam Humidifier (on)      | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Tine M. Vacuum (on)          | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Hoover M54 Robot Vacuum (on)| ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| THREE Smart Light Bulb (on)  | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |
| Kenmore AC (set(val))       | ✓    | ✓   | ✓     | ✓     | ✓      | ✓     |

† Actuation command does not influence sensors (Determined with experiments (Section 5.1.2)). Int/UnInt is marked based on the description of IFTTT apps. See Appendix 6 for sensor brands.

Optimization-Guided Falsification. Falsification searches for counterexamples \( (\exists x \in X, u \in U) \) that violate a policy on the composite automaton \( \Phi (M, x, u) \neq \psi \) \[1, 4\]. Unlike grid-based testing, it takes ranges for the distances between devices and the actuators’ start times. It uses an optimization algorithm (Monte Carlo sampler) to scalably search for policy violations by sampling distances and start times from given ranges. It then executes the composite automaton and records data traces, and computes a robustness value similar to the grid-based testing. The sampler then seeds another input to the composite automaton within the ranges (similar to the input generation in fuzzing \[29\]). The sampler’s objective is to minimize robustness, as its negative values indicate a policy violation. The termination criteria for input generation is either when the policy is falsified or a user-defined max number of iterations is met.

Root Cause Analysis. Testing and falsification output a quintuple, \( 0 = (\text{inputs}, \text{apps}, \text{dist}, \text{stime}, \nu) \), that details a policy violation’s root cause. Here, \( \text{dist} \) is the distance from actuators to sensors, \( \text{stime} \) is the actuators’ start times, and \( \nu \) is a set of \( \{\text{Int, UnInt}\} \) labeled sensor and actuator states. Turning to Figure 1, robot-vacuum-on \((\text{App}_1)\) triggers motion-active \((\text{MalApp}_1)\) that turns on the lights and unlocks the patio door. The security analyzer outputs a quintuple:

\[
\text{input} = [\text{robot-vacuum-on}], \\
\text{apps} = [\text{timer, robot-vacuum, app}], [\text{mot-sen, light, pat-door, malapp}],] \\
\text{stime} = [\text{robot-vacuum}] = 10, \text{dist} = \text{robot-vacuum, mot-sensor} = 2, \\
\nu = [\text{timer-event, robot-vacuum move}, \\
\text{motion-active(UnInt), light.on, patio-door unlocked}].
\]

The security analyzer parses the quintuple to identify the root cause, robot-vacuum-on’s UnInt motion triggers motion-active event invoking actuation commands. It further outputs the violation occurs when the robot-vacuum-on command is invoked at minute 10 of the automata execution, and it is 2 m away from the motion sensor.

Patch the IoT Deployment. We present a report that guides users to patch an IoT deployment and eliminate the identified physical interaction vulnerabilities. The security analyzer first reports to users which apps to remove to eliminate the physical interaction. For instance, removing App or MalApp eliminates the individual policy violation above. However, users may not always desire to remove apps if their functionality is needed. In this case, the security analyzer proposes patches based on the violated policy category. If an aggregation or a dependency policy is violated, it outputs a predicate code block, which guards the app’s actuation commands, to be inserted to app source code. The predicate is false if other actuation commands in the policy violation were invoked within a time limit (determined from the actuators’ stime). This means the app cannot invoke its actuation command causing the physical interaction. Turning to the scenario 2 in Figure 1, we insert a predicate in MalApp that checks if cooker-on occurred. If it did occur, the predicate blocks the oven-on command. This eliminates the interaction with the temperature sensor in App since cooker-on’s individual influence cannot trigger an event. If an individual policy is violated, the security analyzer guides users to increase the distance between the actuator and the sensor. For instance, increasing the distance between the sound sensor and the exhaust fan eliminates the interaction between MalApp’s Fan-on and App’s sound-detected.

6.1 Automata Conformance Experiments

We evaluate IoTSeer in a real home with 6 sensors and 13 actuators, as shown in Figure 7. We use 37 popular official IFTTT apps \[35\] to automate the IoT devices. We constructed 26 actuation command and 6 sensor event automata from the apps. Table 2 presents physical channel labels \((\text{Int/UnInt})\) of each actuation command measured by sensors. We develop IoTSeer in Matlab \[42\] using the Simulink toolbox \[43\]. We present the apps, automata flow functions, and an example automaton developed in Simulink in Appendix 1. We run the automata executions on a laptop with a 2.3 GHz 2-core Intel i5 processor and 8 GB RAM, using Matlab R2019b and Simulink 10.0. We set the execution-time to 60 mins with a 12 secs time-step to collect data traces for automata conformance and policy validation.

We begin by evaluating the conformance of the automata with real devices (Section 6.1). We next validate the policies on the composite automaton and identify policy violations (Section 6.2). We verify all reported violations are true positives through experiments in the actual house (Section 6.2.1). Lastly, we present IoTSeer’s performance overhead (Section 6.3).
devices are turned on to estimate the noise from the environmental factors. We use these data traces to tune automata parameters and integrate environmental factors. This process ensures that a composite automaton has the correct functionality with the IoT deployment amenable to policy validation.

**Tuning Automata Parameters.** We found that 10 out of 26 actuation command automata have high deviation scores \((\tau, \epsilon)\)-closeness). These include AC-on (3), heater-on (2), oven-on (1), humidifier-on (1), dehumidifier-on (1), garbage-disposal-on (1), and TV-on (1)—the values in parentheses represent the number of automata with high deviation scores. To illustrate, the sound sensor automaton detects the garbage-disposal-on’s sound in greater distances than those observed in the actual device traces. Specifically, the deviation score is \(\epsilon = 1\) (highest for a binary output) when the distance between the garbage disposal and sound sensor is greater than 1 meter. To address this, we use SI and find the optimal automata configurations where the deviation score between automata traces and device traces is minimized. For instance, we found that decreasing the sound pressure parameter of the garbage-disposal-on automaton minimizes the deviation. Following the same approach for 10 actuation command automata, we tuned their configuration parameters.

Table 3 details the \((\tau, \epsilon)\)-closeness of actual devices with the automata after parameters are tuned. Each row includes the mean and standard deviation of the timing difference \((\tau)\) and deviation score \((\epsilon)\) in different distances. We observe slight deviations in temperature, illuminance, sound, and humidity automata inputs. This is due to the uncertainties in the environmental factors (e.g., amount of sunlight in the room [22]) that impact actual device traces.

To illustrate, Figure 8 plots the traces from automata and device experiments of light-bulb-on’s influence on the illuminance sensor and oven-on’s influence on the temperature sensor. We found that the light-bulb-on automaton deviates on average \(\approx 20\) lux from device data traces. Additionally, oven-on increases the temperature sensor readings by 1.8°F in both automata and actual device traces at 0.5 meters. The automaton yields an increase at minute 19.6, but the increase in the device traces occurs at 23.8. This leads to an average \(\tau = 0.8 \pm 1.9\) timing difference and \(\epsilon = 0\) deviation score. As we detail in Section 6.2, these observations lead to safe overapproximation in detecting violations at slightly different times.

**Environmental Factors.** We evaluate the following environmental factors in the house’s actual layout (depicted in Figure 7): (1) humans doing an activity in the environment, (2) furniture between devices, and (3) an uncontrolled environmental noise. We found these factors impact automata conformance under rare conditions.

To account for them, we add new automata to composite automata and refine the actuation command automata’s flow functions.

**Human Inhabitants.** To assess the human-presence on physical channels, we obtain traces from sensors when an author and an independent Ph.D. student are cooking, watching TV, and exercising. We select these common activities as they potentially influence the temperature and humidity physical channels. We evaluate the impact of body heat and respiration on physical channels, as actuation command automata of cooking appliances and TV are already constructed. We observe a 2% RH increase in humidity sensor readings when a user is < 20 cm away from the sensor and exercising for 10 mins. For other activities, the users do not introduce detectable changes to sensors. We add a human-exercising automaton with int label output into the composite automaton to account for the user’s influence on humidity. The \{on, off\} states are defined as user-present and not-present that allows executing the composite automaton with and without human presence in the house.

**Physical Obstacles.** We collect sensor data traces with furniture placed between devices based on the room layout. The opaque obstacles between a light source (bulb and TV) and illuminance sensor change the sensor readings, whereas furniture negligibly impacts other physical channels. For instance, when the bulb turns on 1.4 m away from the sensor, we observe \(\approx 40\) lux without an obstacle, and \(\approx 15\) lux with an obstacle. To integrate this into composite automata, we refine the bulb-on automaton by multiplying its output with a coefficient \(c = 0.4\), which is obtained from the deviation difference between automata and actual sensor traces.

**Environmental Noise.** To evaluate uncontrolled environmental factors, we obtain data traces from sensors when only the mechanical devices in the deployment are on. We observe a 40 \(\pm 1.5\) dB ambient noise on the sound sensor due to the refrigerator engine’s routine sound, whereas the noise on other physical channels was negligible. Hence, we add an ambient-noise automaton that influences the sound sensor with Gaussian noise of mean \(\mu = 40\) and standard deviation \(\sigma = 1.5\) to the composite automaton.

### 6.2 Effectiveness

Table 4 presents 11 flagged policy violations caused by physical interactions among different groups of devices in 37 apps installed in the house. In these experiments, we use the composite automaton of apps after automata conformance when a user is present. All violations are identified with both grid-based testing and falsification.

IoTSeer’s prototype implementation (before automata conformance) identified 9 out of 11 policy violations, where two violations involving AC-on’s sound and humidity physical channels were
missed due to the high deviations in its automata. We exercised each policy violation in the house and verified they are true positives.

**Individual Policy (I) Violations.** IoTSeer flagged two unique physical interactions that cause I violations. UnInt sound from garbage-disposal-on triggers 5 apps ($V_1$), and UnInt motion from robot-vacuum-on triggers 5 apps ($V_2$). For instance, in $V_1$, the sound sensor ($A$) detects the sound from garbage disposal ($B$) and triggers $App_{17}$ that turns on the lights, and $App_{20}$ that sends a notification. In $V_2$, the motion sensor ($A$) detects the presence of the robot vacuum ($B$) and triggers $App_{32}$ that calls the user, and $App_{23}$ that turns on the oven and cooker. We note that there are many other apps in the IFTTT market conditioned on sound-detected and motion-active events, such as those that open the garage door, window, or sound an alarm. While these apps do not yield 1 policy violations in the experimental house as the actuators are not installed, such interactions may cause violations in different deployments.

**Aggregation Policy (G) Violations.** IoTSeer flagged eight unique physical interactions among 25 apps causing $G_1 - G_6$ violations. We detail a violation for each aggregation policy, $V_3$ for $G_1$, $V_4$ for $G_2$, and $V_5$ for $G_6$. In $V_3$, the temperature sensor ($A$) measures the aggregated UnInt temperature from $coffee-maker-on$ ($B$), $oven-on$ ($B$), $dryer-on$ ($B$), and $cooker-on$ ($B$). The increase in temperature triggers $App_{21}$ that turns on the AC, and $App_{22}$ that turns on the humidifier. We note that the same violation occurs when a subset of these commands (two or three of them) are invoked. In $V_4$, the illuminance sensor ($A$) measures the aggregated luminance of $bulb-on$ ($B$) and $TV-on$ ($B$). The increase in luminance triggers ($App_{24}$) turning off the bulb. Here, bulb-on’s 1Int labeled luminance does not exceed the illuminance sensor’s sensitivity threshold. Yet, its aggregation with $TV-on$’s UnInt luminance exceeds the threshold. Lastly, in $V_5$, the heater ($A$) and $AC$ ($B$) turn on at the same time that causes a conflicting influence on the temperature sensor’s measurements ($A$).

---

### Table 4: Policy violations identified by IoTSeer.

| Devices | Violation Description | Example Apps Triggered |
|---------|-----------------------|------------------------|
| $A$: [GarbageDisposal](a), [SoundSensor](a) | Sound sensor detects un-introduced sound of the garbage disposal and triggers apps | 
| $B$: [CoffeeMaker](b), [Dryer](b), [PressureCooker](b), [TemplarSensor](b) | | 
| $B$: [MotionSensor](b) | of the robot vacuum and triggers apps | 
| $A$: [Oven](a), [Dryer](a), [PressureCooker](a), [TemplarSensor](a) | Temperature sensor reading increases due to un-introduced heat of the coffee maker, oven, dryer, pressure cooker and triggers apps | 
| $A$: [CollisionSensor](a) | of the robot vacuum and triggers apps | 
| $B$: [MotionSensor](b) | due to un-introduced heat of the coffee maker, oven, dryer, pressure cooker and triggers apps | 
| $B$: [SoundSensor](b) | of the AC, washer, and dryer and triggers apps | 
| $B$: [LuminaireSensor](b) | due to both int and un-introduced heat of the coffee maker and triggers apps | 
| $A$: [LightSensor](a), [TemplarSensor](a) | Maintains sensor detects both | 
| $A$: [LightSensor](a) | Windows of the TV and triggers apps | 
| $A$: [Heater](a), [AC](a), [Humidifier](a) | The heater and AC have conflicting impacts on temperature sensor reading | 
| $B$: [Humidifier](b) | The humidifier and AC have conflicting impacts on humidity sensor reading | 
| $B$: [Humidifier](b) | The humidifier and Associated have conflicting impacts on humidity sensor reading due to the dependency. | 

Both grid-based testing and optimization-guided falsification identify all violations. Grid-based testing outputs multiple start times which cause the violation, falsification outputs a single start time for each actuator.

---

### Figure 9: Illustration of (a) $V_2$ violation, and (b) $V_7$ violation.

#### Dependency Policy (D) Violation.** IoTSeer identified a single dependency violation ($V_{11}$) between 4 apps. This violation happens when heater-on ($B_1$) increases temperature and humidifier-on ($B_2$) increases humidity. The dependency between humidity and temperature leads to a conflicting impact on the humidity sensor ($A$).

---

### 6.2.2 Comparison with Previous Work.** We compare the policy violations flagged by IoTSeer with IoT systems in Table 5. MenShen [10], iRuler [60], IoTCom [3], IoTMon [21], that target identifying physical interactions among IoT apps. If we assume these systems correctly map all 26 physical channels of 13 actuation commands in our deployment through manual mappings or NLP techniques, they are able to identify 2 out of 11 violations (individual (I) policy violations $V_1$ and $V_2$). This is because they do not model the physical behavior of devices and do not consider aggregation and dependency. Additionally, they flag 24 false alarms as most physical channels do not individually cause app interactions. To illustrate, they define a physical channel between the temperature sensor and oven; however, in our deployment, oven-on’s individual influence on temperature is not enough to cause an interaction.

---

---
6.2.3 Porting IoTSeer to other IoT Deployments. Users can port IoTSeer to other IoT deployments by following 4 steps. (1) Users need to derive the automata of actuation commands and sensor events and update the physical-channel reference file if their IoT deployment includes apps that automate devices different from the 13 actuators and 6 sensors we considered. Our constructed automata are generic for a family of devices and can be directly used if the automata of new devices change the same physical channels. 5 (2) Users need to update the automata parameters through device spec sheets (e.g., the operation power of oven), which is necessary to obtain the correct physical behavior of actual devices. (3) The physical channel labels (Int/UnInt) of the physical interaction policy templates (Table 1) need to be updated based on the users’ needs. (4) Users need to collect data traces from IoT deployment to tune the automata parameters for precision and to accommodate for the environmental factors (Detailed in Section 6.3).

6.3 Performance Evaluation

Automata Size vs. Time. We evaluate the overhead of IoTSeer’s policy validation algorithms with an increasing automata size. Figure 10 (Left) shows the policy validation time with an increasing number of actuation command automata that influence the same physical channel. Grid-based testing time overhead increases with the number of actuation commands as each policy is validated using a combination of actuator start times, and all violations are identified. On the contrary, falsification maintains a near constant time overhead. The optimization algorithm samples actuator start times and searches for low robustness values for a single violation. This adds a negligible delay, on the order of seconds, to policy validation with an increasing number of actuation commands.

Number of Policies vs. Time. We evaluate the policy validation time with an increasing number of policies. We set the number of actuation command automata to three, and a sensor measures their aggregated influence on a physical channel. Figure 10 (Right) shows the time overhead of both testing and falsification increases linearly with the number of policies. Testing takes ≈ 3X longer than falsification because it reports all violations, while falsification searches for a single counterexample that violates the policy.

Time for Trace Collection. We present the time for trace collection from constructed automata and actual devices for (1) generating the physical-channel reference file, (2) tuning automata parameters, and (3) addressing environmental factors. These steps are required to improve IoTSeer’s accuracy in finding physical interactions among apps and minimizing the false alarms. Here we note that, even without steps 2 and 3, IoTSeer identified 9 out of 11 violations in our experimental house by only using the automata constructed through physics-based modeling. First, to generate the physical-channel reference file (Table 2), we collected 6.5 hours of data traces from 13 actuation commands that impact 6 sensors’ measurements. Second, we recorded 33.3 hours of data traces to tune automata parameters through conformance experiments. This is an improvement over generating the flow functions by system identification solely using device traces, which would require ≈ 650 hours of data traces with different device configurations and distances. Lastly, to address the environmental factors, we recorded 8.3 hours of traces and spent 3.5 hours to define new automata and derive coefficients for integrating them into the composite automaton.

7 DISCUSSION & LIMITATIONS

IoTSeer can be used to validate additional safety and security policies that are based on the use cases of one or multiple devices. For instance, a user may desire to validate a policy that states the alarm must go off in two seconds after smoke is detected. This can be represented with MTL as $\Box_{\text{smoke > threshold}} \rightarrow \Diamond_{\text{alarm = ON}}$. In future work, we will identify such policies through requirements engineering and validate them on the composite automaton.

IoTSeer defines the flow functions of the actuation commands and sensor events through algebraic and differential equations that are well-studied in control theory. However, there may be more complex equations that are a better contextual fit. These can be integrated into IoTSeer by replacing the automata’s flow functions.

Future work will study the optimization techniques such as actuation commands’ monotonic behavior on physical channels to minimize the validation inputs for scalability, and evaluate IoTSeer in other deployments such as safety-critical CPSs.

8 RELATED WORK

There has been an increasing number of works in IoT safety and security. We compare IoTSeer with several recent approaches that focus on the identification of physical interactions among IoT apps. As presented in Table 5, IoTSeer is the first system that integrates the complex properties of the physical behavior of actuation commands and sensor events into the source code of IoT apps (Columns “Physical Channel Properties”). Unfortunately, none of the systems model the complex physical behavior of apps but they manually build physical channel mappings (e.g., a mapping between heater and temperature) [3], generate naive actuator models (e.g., oven increases temperature $1^\circ$F per hour) [10], or use machine learning techniques (e.g., inspection of apps’ text descriptions) [21, 60] to infer interacting apps. Therefore, they identify limited physical interactions and lead to significant false positives, as we compared the number of violations they can identify with IoTSeer in Section 6.2.2.

The systems studied here use different formulas (e.g., LTL) to express the physical interactions and different verification techniques (e.g., model checking) to validate policies on the abstracted
app models (Columns "Security Analysis"). However, their policy representations do not consider timing constraints on the physical channels and physical channel labels. Additionally, their validation techniques cannot be readily used to verify physical interactions since apps exhibit both discrete and continuous behaviors. In contrast, IoTSeer expresses policies with MTL for timing constraints, integrates physical channel labels into MTL, and extends grid-based testing for exhaustiveness and falsification for scalability in policy validation of large-scale IoT deployments. Lastly, none of the systems is able to identify the root cause of the complex physical interactions. IoTSeer reports the root cause of the violation and proposes patches that guide users on how to fix them.

9 CONCLUSION

We present the IoTSeer security service, which identifies the physical interaction vulnerabilities in IoT deployments. Our evaluation in a real house demonstrates that many apps interact with each other, and IoTSeer efficiently identifies all policy violations. IoTSeer is an important step forward in achieving the compositional safety and security of an IoT system’s physical behavior.

REFERENCES

[1] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, FranoI Ivanici, and Aarti Gupta. 2013. Probabilistic temporal logic falsification of cyber-physical systems. ACM Transactions on Embedded Computing Systems (TECS) 12 (2013).

[2] Houssam Abbas, Hans Mittelmann, and Georgios Fainekos. 2014. Formal property verification in a conformance testing framework. In ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE).

[3] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis and runtime monitoring to learn control flow of IoT models. In USENIX Annual Technical Conference (USENIX ATC).

[4] Yaqi Chen, Sayali Kate, Yousra Aafer, Xiangyu Zhang, and Dongyan Xu. 2020. Software-based Realtime Recovery from Sensor Attacks on Robotic Vehicles. In International Symposium on Research in Attacks, Intrusions and Defenses (RAID).

[5] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, and Xinyan Deng. 2018. Detecting attacks against robotic vehicles: A control invariant approach. In ACM SIGSAC Conference on Computer and Communications Security (CCS).

[6] Edmund M. Clarke and E. Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In Workshop on Logic of Programs. Springer.

[7] R. Claussius. 1850. On the motive power of heat and the laws which can be deduced therefrom regarding the theory of heat. Annalen der Physik 155, 4 (1850), 500–524.

[8] Pritam Das, Mehdi Karimi-Babakashi, and Karthik Pattabiraman. 2019. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In Proceedings of the 35th Annual Computer Security Applications Conference.

[9] Wenbo Ding and Hongxin Hu. 2018. On the safety of IoT device physical interaction control. In ACM SIGSAC Conference on Computer and Communications Security (CCS).

[10] Isaac Elshlaffok. 2000. Whys and haws in uncertainty modelling. Springer.

[11] Georgios Fainekos, Barath Hoxha, and Sriram Sankaranarayanan. 2019. Robustness of Specifications and Its Applications to FalsePath, Parameter Mining, and Runtime Monitoring with S-TaLiRo. In International Conference on Runtime Verification. Springer.

[12] Georgios E Fainekos and George J Pappas. 2006. Robustness of temporal logic specifications. In Formal Approaches to Software Testing and Runtime Verification. Springer.

[13] Forum Post. 2015. High utility bills! Your thermostat might be on the wrong wall - Comments. https://tinyurl.com/y39wwee8. (2015). [Online; accessed 25-August-2020].

[14] Forum Post. 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[15] Leon R Glickman and Steven Taub. 1997. Thermal and behavioral modeling of occupant-controlled heating, ventilating and air conditioning systems. Energy and Buildings 25, 3 (1997), 243–249.

[16] GNU Octave 2020. GNU Octave: Scientific Programming Language. https://www.gnu.org/software/octave/. (2020). [Online; accessed 15-May-2020].

[17] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, and Xinyan Deng. 2018. Detecting attacks against robotic vehicles: A control invariant approach. In ACM SIGSAC Conference on Computer and Communications Security (CCS).

[18] Edmund M Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In Workshop on Logic of Programs. Springer.

[19] Mitchel Day, Scott Ahn, and Patrick McDaniel. 2018. IoTSan: Fortifying the safety of open source home automation IoT systems. ACM Transactions on Cyber-Physical Systems 2 (2018), 1–23.

[20] Pritam Dash, Mehdi Karimibiuki, and Karthik Pattabiraman. 2019. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In Proceedings of the 35th Annual Computer Security Applications Conference.

[21] Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Hybrid i/o automata. In Hybrid multi-agent systems. Springer.

[22] Forum Post. 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[23] Leon R Glickman and Steven Taub. 1997. Thermal and behavioral modeling of occupant-controlled heating, ventilating and air conditioning systems. Energy and Buildings 25, 3 (1997), 243–249.

[24] GNU Octave 2020. GNU Octave: Scientific Programming Language. https://www.gnu.org/software/octave/. (2020). [Online; accessed 15-May-2020].

[25] Forum Post 2015. High utility bills? Your thermostat might be on the wrong wall - Comments. https://tinyurl.com/y39wwee8. (2015). [Online; accessed 25-August-2020].

[26] Forum Post 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[27] Leon R Glickman and Steven Taub. 1997. Thermal and behavioral modeling of occupant-controlled heating, ventilating and air conditioning systems. Energy and Buildings 25, 3 (1997), 243–249.

[28] GNU Octave 2020. GNU Octave: Scientific Programming Language. https://www.gnu.org/software/octave/. (2020). [Online; accessed 15-May-2020].

[29] HomeKit 2020. Apple’s HomeKit. https://www.apple.com/ios/homekit/. (2020). [Online; accessed 30-August-2020].

[30] HomeKit 2020. HomeKit Blog - Forum. https://homekit.blog/homekit-forum/. (2020). [Online; accessed 10-September-2020].

[31] AG Jackson, SIP Labute, and J Busbee. 1996. Sensor principles and methods for Smoke Detector Technology. Final Report.

[32] AG Jackson, SIP Labute, and J Busbee. 1996. Sensor principles and methods for Smoke Detector Technology. Final Report.

[33] AG Jackson, SIP Labute, and J Busbee. 1996. Sensor principles and methods for Smoke Detector Technology. Final Report.

[34] Forum Post 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[35] Forum Post 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[36] AG Jackson, SIP Labute, and J Busbee. 1996. Sensor principles and methods for Smoke Detector Technology. Final Report.

[37] HomeKit 2020. Apple’s HomeKit. https://www.apple.com/ios/homekit/. (2020). [Online; accessed 30-August-2020].

[38] HomeKit 2020. HomeKit Blog - Forum. https://homekit.blog/homekit-forum/. (2020). [Online; accessed 10-September-2020].

[39] IFTTT 2020. IFTTT (If This Then That). https://ifttt.com/. (2020). [Online; accessed 18-May-2020].

[40] T. G. Lee and George W. Mulholland. 1977. Physical Properties of Smokes Pertinent to Smoke Detector Technology. Final Report. NIST Interagency/Internal Report (NISTIR) (1977).

[41] Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Hybrid i/o automata. Information and computation 185, 1 (2003), 105–157.

[42] Forum Post 2015. High utility bills? Your thermostat might be on the wrong wall - Comments. https://tinyurl.com/y39wwee8. (2015). [Online; accessed 25-August-2020].

[43] Forum Post 2018. Rooma sets off motion sensor at night? https://tinyurl.com/yyuxsifys. (2018). [Online; accessed 25-August-2020].

[44] HomeKit 2020. Apple’s HomeKit. https://www.apple.com/ios/homekit/. (2020). [Online; accessed 30-August-2020].

[45] HomeKit 2020. HomeKit Blog - Forum. https://homekit.blog/homekit-forum/. (2020). [Online; accessed 10-September-2020].

[46] JF Chang, YZ Chen, and JZ Hu. 1997. Automatic generation of test cases in distributed interactive simulation. In Proceedings of the 14th International Conference on Distributed Simulation (DISSim). ACM.

[47] JF Chang, YZ Chen, and JZ Hu. 1997. Automatic generation of test cases in distributed interactive simulation. In Proceedings of the 14th International Conference on Distributed Simulation (DISSim). ACM.
We note that IoTSeer's implementation can be easily migrated to other software such as GNU Octave [28], SpaceEx [55], Scilab [53] with third-party code converters.

**APPENDIX**

**1 APPENDIX GUIDE**

This Appendix provides the necessary information to reproduce our results. In Appendix 2, we present an example composite automaton developed with Simulink. In Appendix 3, we detail the apps considered in our motivation and experiments. In Appendix 4, we present the implementation of our policy validation techniques. In Appendix 5, we present our flow functions of the studied actuation commands and sensor events. In Appendix 6, we present the impact of actuation commands on physical quantities, the distances between the sensors they impact, and the sensor sensitivity levels. We note that IoTSeer contains all the tools necessary to extend the composite automaton to other apps, and allows for replacing the flow functions of the studied actuation commands and sensor events with ones that may be a better contextual fit.

**2 SIMULINK AUTOMATON EXAMPLE**

The composite automaton of light-bulb-on, TV-on commands and light-detected event is presented in Figure 1. The actuation command and sensor event automata (including their flow functions) are implemented within Matlab functions inside the Simulink boxes.

We note that IoTSeer’s implementation can be easily migrated to other software such as GNU Octave [28], SpaceEx [55], Scilab [53] with third-party code converters.

**3 APPS DEPLOYED IN THE EXPERIMENTS**

Table 1 lists the description of the apps considered in our paper. Ex_App1, Ex_App2, and MalApp1, MalApp2 are the apps used in Section 3 Figure 1. App1, App37 are the apps deployed in our experiments in Section 6 from the official IFTTT trigger-action platform.

**4 POLICY VALIDATION**

We use open-source temporal logic falsification toolbox S-TaLiRo’s [5] dptaliro function as a subroutine into our grid-based testing algorithm and the staliro function into optimization-guided falsification to validate MTL policies on the composite automaton. We use the staliro function with simulated annealing by hit-and-run Monte Carlo sampling for input generation [23]. In both functions, we use a dynamic programming-based algorithm to compute the robustness of MTL policies.

**Parameter Selection.** For grid-based testing, we set fixed distances among sensors and actuators as in Figure 7. We consider apps invoke actuation commands with 10-min intervals, thus, the actuators’ start times are set as a grid to 0:10:60. For falsification, we set ranges for distances among sensors/actuators as ±0.5 m on the placement in Figure 7 to account for possible variations. We set the actuators’ start time ranges as any time in the execution (0 – 60). We define the maximum number of tests to 100 as we do not observe a significant change in robustness after 100.

**5 AUTOMATA FLOW FUNCTIONS**

To determine the flow functions of automata, we first analyze the physical channels each sensor event observes [36, 59]. We then study how actuation commands influence these channels, and how they diffuse in a way that changes the sensor readings [9, 11, 19].

![Figure 1: Simulink automaton of light-bulb-on, TV-on, and light-detected.](image-url)
Table 1: App list and their descriptions.

| ID | Description |
|----|-------------|
| App1 | Activates the heater at a user-defined time |
| App2 | Activates the oven at a user-defined time |
| App3 | Activates the robot vacuum at a user-defined time |
| App4 | Activates the light bulb at a user-defined time |
| App5 | Activates the dehumidifier at a user-defined time |
| App6 | Activates the clothes washer at a user-defined time |
| App7 | Activates the dryer at a user-defined time |
| App8 | Activates the humidifier at a user-defined time |
| App9 | Activates the garbage disposal at a user-defined time |
| App10 | Activates the TV at a user-defined time |
| App11 | Activates the robot vacuum at a user-defined time |
| App12 | Activates the light bulb at a user-defined time |
| App13 | Activates the AC at a user-defined time |
| App14 | Activates robot vacuum when the house mode is set to 'away' or vacation mode |
| App15 | Wipes (thermostat) 'away mode' ON at a user-defined time |
| App16 | Notifies user when sound is detected |
| App17 | Sends a message to the user when temperature is too high |
| App18 | Calls user if smoke is detected |
| App19 | Sends a message to the user when smoke is detected |
| App20 | Turns on the AC based on its power (positive for heater, negative for AC) and q denotes whether the system’s actuation command (on or off) |
| App21 | Sends a message when the air quality is too low |
| App22 | Activates the heater when the air quality is too low |
| App23 | Activates the robot vacuum when the air quality is too low |
| App24 | Activates the light bulb at a user-defined time |
| App25 | Activates the AC at a user-defined time |
| App26 | Activates the oven at a user-defined time |
| App27 | Activates the robot vacuum when the air quality is too low |
| App28 | Activates the light bulb at a user-defined time |
| App29 | Activates the AC at a user-defined time |
| App30 | Activates the robot vacuum when the air quality is too low |
| App31 | Activates the AC at a user-defined time |
| App32 | Activates the robot vacuum when the air quality is too low |
| App33 | Activates the AC at a user-defined time |
| App34 | Activates the robot vacuum when the air quality is too low |
| App35 | Activates the AC at a user-defined time |
| App36 | Activates the robot vacuum when the air quality is too low |
| App37 | Activates the AC at a user-defined time |
| App38 | Activates the robot vacuum when the air quality is too low |
| App39 | Activates the AC at a user-defined time |
| App40 | Activates the robot vacuum when the air quality is too low |
| App41 | Activates the AC at a user-defined time |
| App42 | Activates the robot vacuum when the air quality is too low |
| App43 | Activates the AC at a user-defined time |
| App44 | Activates the robot vacuum when the air quality is too low |
| App45 | Activates the AC at a user-defined time |
| App46 | Activates the robot vacuum when the air quality is too low |
| App47 | Activates the AC at a user-defined time |
| App48 | Activates the robot vacuum when the air quality is too low |
| App49 | Activates the AC at a user-defined time |
| App50 | Activates the robot vacuum when the air quality is too low |
| App51 | Activates the AC at a user-defined time |
| App52 | Activates the robot vacuum when the air quality is too low |
| App53 | Activates the AC at a user-defined time |
| App54 | Activates the robot vacuum when the air quality is too low |
| App55 | Activates the AC at a user-defined time |
| App56 | Activates the robot vacuum when the air quality is too low |
| App57 | Activates the AC at a user-defined time |
| App58 | Activates the robot vacuum when the air quality is too low |
| App59 | Activates the AC at a user-defined time |
| App60 | Activates the robot vacuum when the air quality is too low |
| App61 | Activates the AC at a user-defined time |
| App62 | Activates the robot vacuum when the air quality is too low |
| App63 | Activates the AC at a user-defined time |
| App64 | Activates the robot vacuum when the air quality is too low |
| App65 | Activates the AC at a user-defined time |
| App66 | Activates the robot vacuum when the air quality is too low |
| App67 | Activates the AC at a user-defined time |
| App68 | Activates the robot vacuum when the air quality is too low |
| App69 | Activates the AC at a user-defined time |
| App70 | Activates the robot vacuum when the air quality is too low |
| App71 | Activates the AC at a user-defined time |
| App72 | Activates the robot vacuum when the air quality is too low |
| App73 | Activates the AC at a user-defined time |
| App74 | Activates the robot vacuum when the air quality is too low |
| App75 | Activates the AC at a user-defined time |
| App76 | Activates the robot vacuum when the air quality is too low |
| App77 | Activates the AC at a user-defined time |
| App78 | Activates the robot vacuum when the air quality is too low |
| App79 | Activates the AC at a user-defined time |
| App80 | Activates the robot vacuum when the air quality is too low |
| App81 | Activates the AC at a user-defined time |
| App82 | Activates the robot vacuum when the air quality is too low |
| App83 | Activates the AC at a user-defined time |
| App84 | Activates the robot vacuum when the air quality is too low |
| App85 | Activates the AC at a user-defined time |
| App86 | Activates the robot vacuum when the air quality is too low |
| App87 | Activates the AC at a user-defined time |
| App88 | Activates the robot vacuum when the air quality is too low |
| App89 | Activates the AC at a user-defined time |
| App90 | Activates the robot vacuum when the air quality is too low |
| App91 | Activates the AC at a user-defined time |
| App92 | Activates the robot vacuum when the air quality is too low |
| App93 | Activates the AC at a user-defined time |
| App94 | Activates the robot vacuum when the air quality is too low |
| App95 | Activates the AC at a user-defined time |
| App96 | Activates the robot vacuum when the air quality is too low |
| App97 | Activates the AC at a user-defined time |
| App98 | Activates the robot vacuum when the air quality is too low |
| App99 | Activates the AC at a user-defined time |
| App100 | Activates the robot vacuum when the air quality is too low |
| App101 | Activates the AC at a user-defined time |

Temperature. Actuation commands’ influence on temperature are due to hot surfaces diffusing heat to their surroundings. Therefore, the heat diffusion equation from a point source [31] (which is based on a partial differential equation) is used as their flow function.

\[
\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \quad (1)
\]

\[
T(t, 0) = T_0 \quad (2)
\]

\[
T(t, x) = T_s \quad (3)
\]

In the formula, \(T\) is temperature in K, \(x\) is the distance from the actuator in meters, \(\alpha\) is the thermal diffusivity constant (in \(m^2/s\)), \(\varepsilon\) is the maximum distance from the source, and \(T_s\) is the temperature of the source. We set the thermal diffusivity as \(2.2 \times 10^{-5} m^2/s\) as a constant [32, 52].

On the other hand, HVAC systems (e.g., heater and AC) influence temperature with an air flow (due to convection) to ensure quick heat dissipation. In particular, air flow of these systems enables uniform temperature from their influences [27]. Therefore, we construct their flow functions through ordinary differential equations [30].

\[
\frac{dT}{dt} = T_\Lambda \cdot q \quad (4)
\]

\[
T(0) = T_0 \quad (5)
\]

In the equation, \(T_0\) is the initial temperature, \(T_\Lambda\) is the impact from the HVAC system based on its power (positive for heater, negative for AC) and \(q\) denotes whether the system’s actuation command (on or off). Although these devices cause air flow in the room in order to enable uniform temperature, the unauthorized devices still generate heat which diffuses to influence the sensor measurements. Therefore, both diffusion and convection play a critical role in the temperature measurements.

Relative Humidity. The ratio of the existing water vapor in the air to the maximum capacity of water vapor that can exist in the air, is defined as relative humidity [39], detailed below.

\[
\text{RH} = \frac{w}{w_s} \times 100 \quad (6)
\]

In the formula, \(w\) represents the water vapor in the air, and \(w_s\) represents the maximum capacity of water vapor the air can contain. \(w_s\) exponentially depends on the ambient temperature, related to the Clausius–Clapeyron equation [19]. Therefore, we leverage existing experiments [58] to fit an exponential function to compute \(w_s\) based on temperature. In particular, this function is defined as:

\[
w_s = 2.6055 \times e^{0.0262 \times T} \quad (7)
\]

where \(T\) represents the temperature of the environment in °F.

The actuation command automaton flow functions are implemented as ordinary differential equations that generate or reduce water content in the air.

Smoke. We consider an ionization-based smoke detector that detects the presence of smoke by filtering air through an ionization chamber. When smoke particles enter through the chamber, conductivity decreases and smoke presence is detected [11]. The threshold of these sensors is in obscuration density per meter (OD/m), where a typical sensor’s sensitivity is 0.02 OD/m corresponding to 13 mg/m³ smoke density [11, 40]. Gas particles (such as the particles in smoke) move very fast through the air in room temperature [9]. Therefore, we define the flow function of actuation command automata as an ordinary differential equation.

Illuminance. The influence of light is instant, and therefore, is modeled with an algebraic equation. Its influence follows the inverse square law, as shown below [59].

\[
I_s = \frac{I_s}{4 \times \pi \times \pi} \quad (8)
\]

In the formula, \(I_s\) denotes the luminosity flux of the source.
Sound. Sound is modeled with an algebraic equation due to its high speed of diffusion over air. Sound intensity is modeled with the inverse square law [59]. Therefore, the sound pressure, which is the quantity the sensors measure, is represented with the following formula.

\[ SP_2 = SP_1 + 20 \times \log_{10} \left( \frac{x_1}{x_2} \right) \] (9)

Sound pressure \( SP_2 \) at distance \( x_2 \) can be calculated in decibels (dB) with this formula, where \( SP_1 \) is the sound pressure level at distance \( x_1 \) (the standard value of \( x_1 \) is 1 meter) [61]. When the distance is doubled, \( SP_2 \) decreases by 6 dB.

Motion. There are different types of motion sensors available on the market. For instance, accelerometers detect motion based on the acceleration generated by the source, PIR sensors detect motion based on the infrared heat map of an environment, and laser-based sensors detect motion by generating an invisible laser between two devices and detecting the cut-offs. We consider PIR motion sensors, and model them through their range to detect motion. If the actuators’ movement rate exceeds a threshold and the distance is close enough, motion is detected.

6 DEVICE SPECIFICATIONS

Actuator configurations, distances to sensors they influence, and the details of their influence are presented in Table 2.

For sensors (presented in Figure 2), sensitivity values, and thresholds for detection (e.g., for sound, illuminance, smoke) are set as follows: (i) temperature sensor \( \pm 1.8^\circ F \), (ii) humidity sensor \( \pm 2\% \), (iii) smoke sensor \( 0.0200/\text{m} = 13\text{mg/m}^3 \), (iv) illum. sensor \( 50\text{ lux} \), (v) sound sensor \( 55\text{ dB} \), and (vi) motion sensor \( -1\text{ m} \).

### Table 2: Details of the actuators in the house.

| Device (ID) | Operation (Time) | Distance (m) | Details² |
|-------------|------------------|--------------|----------|
| 1 | set(temp) | temp - 2.8 | hum - 2.8 | set(70 – 82°F) Dep |
| 2 | 10 | temp - 1.5 | smoke - 3 | hum - 1.5 | surrTemp = 10°F smokeGen = 0.5g/min Dep |
| 3 | 15 | temp - 0.9 | hum - 0.9 | | surrTemp = 10°F Dep |
| 4 | 3 | temp - 0.5 | hum - 0.5 | | surrTemp = 15°F Dep |
| 5 | 20 | hum - 1.5 | | vaporRes = 8.5g/min |
| 6 | 25 | temp - 1.7 | sound - 1.8 | motion - 2.5 | vaporGen = 0.1g/min 55 dB at 1 m Vibrations |
| 7 | 30 | temp - 1.8 | sound - 1.8 | motion - 2.6 | surrTemp = 96°F Dep 58 dB at 1 m Vibrations |
| 8 | 20 | hum - 1.8 | | vaporGen = 8.8g/min |
| 9 | 0.2 | sound - 0.8 | | 58 dB at 1 m |
| 10 | 25 | illum - 1.2 | | 400 lumens 62 dB at 1 m |
| 11 | 20 | motion - variable* | | Move towards sensor |
| 12 | 15 | illum - 1.4 | | 815 lumens |
| 13 | set(temp) | temp - 2 | hum - 2 | vaporRes = 18g/min 62 dB at 1 m |
| 14 | | sound - 3.5 | | |

* Robot vacuum moves towards the motion sensor.

† temp: Temperature Sensor, smoke: Smoke Sensor, hum: Humidity Sensor, illum: Illuminance Sensor, motion: Motion Sensor, sound: Sound Sensor

‡ surrTemp: Surface Temperature, vaporGen: Water Vapor Generation Rate, vaporRes: Water Vapor Removal Rate. Dep: The actuator impacts the sensor due to dependency, smokeGen: Smoke Generation Rate.