Discovering IoT Physical Channel Vulnerabilities

Muslum Ozgur Ozmen
Purdue University
mozmen@purdue.edu

Xuansong Li*†‡
School of Computer Science and Engineering, Nanjing University of Science and Technology
lixs@njust.edu.cn

Andrew Chu†
University of Chicago
andrewcchu@uchicago.edu

Z. Berkay Celik‡
Purdue University
zcelik@purdue.edu

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

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

ABSTRACT

Smart homes contain diverse sensors and actuators controlled by IoT apps that provide custom automation. Prior works showed that an adversary could exploit physical interaction vulnerabilities among apps and put the users and environment at risk, e.g., to break into a house, an adversary turns on the heater to trigger an app that opens windows when the temperature exceeds a threshold. Currently, the safe behavior of physical interactions relies on either app code analysis or dynamic analysis of device states with manually derived policies by developers. However, existing works fail to achieve sufficient breadth and fidelity to translate the app code into their physical behavior or provide incomplete security policies, causing poor accuracy and false alarms.

In this paper, we introduce a new approach, IoTSeer, which efficiently combines app code analysis and dynamic analysis with new security policies to discover physical interaction vulnerabilities. IoTSeer works by first translating sensor events and actuator commands of each app into a physical execution model (PeM) and unifying PeMs to express composite physical execution of apps (CPeM). CPeM allows us to deploy IoTSeer in different smart homes by defining its execution parameters with minimal data collection. IoTSeer supports new security policies with intended/unintended physical channel labels. It then efficiently checks them on the CPeM via falsification, which addresses the undecidability of verification due to the continuous and discrete behavior of IoT devices.

We evaluate IoTSeer in an actual house with 14 actuators, six sensors, and 39 apps. IoTSeer discovers 16 unique policy violations, whereas prior works identify only 2 out of 16 with 18 falsely flagged violations. IoTSeer only requires 30 mins of data collection for each actuator to set the CPeM parameters and is adaptive to newly added, removed, and relocated devices.

*Also with State Key Laboratory for Novel Software Technology, Nanjing University.
†This work was completed while the authors were at Purdue University.
‡Corresponding authors.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560644

CCS CONCEPTS

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

KEYWORDS

Smart Homes; Security Analysis; Physical Channel Vulnerabilities

ACM Reference Format:
Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z. Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. 2022. Discovering IoT Physical Channel Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560644

1 INTRODUCTION

With the growing number of IoT devices co-located in an environment, the interactions among IoT apps cause increasing safety and security issues [16, 17, 25, 35, 50, 53]. There are two fundamental sources of app interactions, software and physical. Software interactions occur when IoT apps interact through a common device defined in their source code. Consider an app that turns on the lights when smoke is detected and another app that locks the door when the lights are turned on. These apps interact through a common light device ($\xrightarrow{\texttt{smoke}}$ `light-on` $\xrightarrow{\texttt{light-on}}$ `door-locked`) and makes residents get trapped during a fire.

Physical interactions are another notable (and stealthier) threat; an app invokes an actuation command, and a sensor detects the physical channel influenced by this command, triggering other apps that actuate a set of devices. Consider an app that turns on the heater and another app that opens the window when the temperature exceeds a threshold. These apps interact through the temperature channel (`heater-on` $\overset{\texttt{temp}}{\rightsquigarrow}$ `window-open`). An adversary who exploits the heater controller app can stealthily trigger the `window-open` command and break into the house when the user is not home.

Discovering software and physical interactions has received increasing interest from the security community since they enable an adversary to indirectly gain control over sensitive devices and put the user and environment in danger. Prior works mainly focus on identifying software interactions via app source code analysis [10, 11, 14, 15, 17, 18, 37, 46]. These approaches find interacting apps by matching the device attributes in multiple apps, such as

the `light-on` attribute in the first example. They cannot detect the physical interactions because the app source code does not state the physical channels, e.g., the heater's influence on temperature.

There have been limited efforts to discover physical interactions. These works mainly (1) use pre-defined physical channel mappings between commands and sensor events [5] and (2) leverage NLP and device behavioral models to map the events and commands of apps [12, 20, 50]. However, these approaches have limited expressiveness of physical channels, causing two issues. They lead to over-approximation of physical channels, which are false alarms (e.g., the system flags the temperature from `oven-on` and opens the window, yet the temperature from the oven is not enough to create a physical channel), and under-approximation of physical channels, which are the interactions that the system fails to identify (e.g., the system ignores the motion from `robot-vacuum-on`).

Recent work identifies physical interactions by collecting run-time device states and enforces security rules at run-time [21]. However, this approach has three limitations, which limit its effectiveness. (1) It defines rules based on devices' use cases to prevent physical interaction vulnerabilities. However, such rules do not prevent unintended interactions that subvert the intended use of apps and devices. For instance, a user installs an app that unlocks the patio door when motion is detected to automate their home entry process. This system, however, does not prevent unlocking the patio door even if the motion is detected due to the vacuum robot's movements. (2) As a dynamic enforcement system, it cannot infer the specific command that influences a physical channel at run-time. For example, when the motion sensor detects motion, it cannot determine if this motion is from the vacuum robot or human presence. (3) When a device location changes, it makes wrong predictions about the app interactions, leading to unnecessarily enforcing rules and failing to prevent violations. For instance, if a portable heater is moved away from the temperature sensor, its influence on temperature measurements decreases. Yet, it would predict a higher influence and turn the heater off prematurely.

In this paper, we introduce IoTSeer, which builds the joint physical behavior of IoT apps through code and dynamic analysis, and validates a set of new security policies to discover physical interaction vulnerabilities. IoTSeer first extracts an app's commands and sensor events from its source code. It translates them into physical execution models (PEMs), which define each app's physical behavior. It unifies the PEMs of interacting apps in a composite physical execution model (CPEM). To maximize CPEM's fidelity in different smart homes, it collects device traces to define its execution parameters. IoTSeer supports new security policies that operate on intended/unintended physical channel labels and validates if IoT apps conform to these policies through falsification. IoTSeer addresses the limitations of prior works with formal physical models of apps, new security policies, and validation through metric temporal logic.

We applied IoTSeer in an actual house with 14 actuators and six sensors, automated by 39 apps from popular IoT platforms. We built the PEMs of 24 actuation commands and six sensor events used in the app source code and unified them in CPEM. IoTSeer found 16 unique physical interaction policy violations on different groups of interacting apps. We compared the violations discovered by IoTSeer with existing works that identify physical interaction vulnerabilities and found that they can only identify 2 out of 16 violations with 18 false positives. We repeated the experiments in the actual house and verified that all violations discovered by IoTSeer are true positives. IoTSeer is adaptive to newly added, removed, and updated devices and imposes minimal model construction and validation time overhead. It requires, on average, 30 mins of data collection for each actuator to set the parameters, and it takes, on average, 21 secs to validate a physical channel policy on four interacting apps.

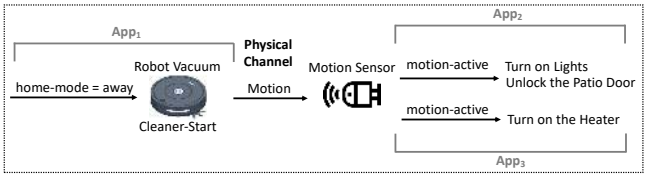


Figure 1: Illustration of physical interactions: the vacuum robot is activated when the user is not home, which unlocks the patio door and turns on the lights and heater.

In this paper, we make the following contributions.

- **Translating App Source Code into its Physical Behavior**: We translate the actuation commands and sensor events in the app source code into physical execution models to define their physical behavior.
- **Composition of Interacting Apps**: We introduce a novel composite physical execution model architecture that defines the joint physical behavior of interacting apps.
- **Physical Channel Policy Validation**: We develop new security policies with intended/unintended physical channel labels. We formally validate the policies on CPEM through optimization-guided falsification.
- **Evaluation in an Actual House**: We use IoTSeer in a real house containing 14 actuators and six sensors and expose 16 physical channel policy violations.
- IoTSeer code is available at
  https://github.com/purseclab/IoTSeer
  for public use and validation.

2 MOTIVATION AND THREAT MODEL

A smart home 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, activating actuation commands (e.g., `door-unlock`). Users install official apps from IoT markets such as HomeKit [28] and OpenHAB [39], and third-party apps through proprietary web interfaces. Another trend for custom automation is trigger-action platforms such as IFTTT [30] and Zapier [52]. These platforms allow users to use conditional statements in the form of if/then rules to integrate digital services with IoT devices. In this paper, we use the term app(s) to refer to both IoT apps and trigger-action rules.

When an actuation command is invoked, it influences a set of physical channels measured by sensors. The command then interacts with apps subscribed to those sensor events, invoking other commands. An adversary can exploit such physical interactions to indirectly control devices and cause unsafe states.

To illustrate, Figure 1 shows a common smart home with four devices. The user installs $\texttt{App}_1$ that starts the robot vacuum cleaner

when the home mode is set to away (or at specific times). An adversary can provide users with App_2 and App_3 that operate correctly in isolation yet exploit the physical interactions to cause unsafe states. When motion is detected (the user enters the home), App_2 turns on the lights and unlocks the patio door, and App_3 sets the heater to a specific temperature value.

In this deployment, the user leaves home and sets the home mode to away, triggering App_1 that starts the robot vacuum cleaner. The movements of the robot vacuum create a physical interaction with App_2 and App_3 since the motion sensor detects the robot vacuum. This results in unlocking the patio door and turning on the lights and the heater while the user is not at home. The unlocked patio door may allow a burglar to break in, the turned-on lights may indicate whether the users are at home or not, and the heater's influence on temperature may trigger other apps (e.g., opening the windows), causing a chain of interactions between multiple apps.

The preceding example shows that the final environment states do not just depend on individual devices but are a result of the physical interactions of multiple devices. Each app is individually safe, yet their unified physical interactions leave users at risk.

2.1 Threat Model

Our threat model is similar to related IoT security works, which focus on app interaction vulnerabilities [17, 20, 21, 50]. We consider an adversary whose goal is to execute undesired device actions (e.g., unlocking the door when the user is sleeping) and cause unsafe system states. The adversary achieves this goal by creating or exploiting physical app interactions.

The adversary can conduct two types of attacks, a mass attack or a targeted attack. In a mass attack, an adversary provides users with apps operating correctly in isolation yet exploits the physical interactions among apps to cause unsafe states on a large scale. The adversary does not target a specific smart home but harms many users and hurts the trustworthiness of an IoT platform. The adversary can conduct this attack by (1) distributing apps on IoT platforms and third-party IoT forums and (2) tricking users into installing apps via phishing and other social engineering methods.

In a targeted attack, the adversary determines a specific smart home to exploit its physical app interactions. First, the adversary discovers an exploitable physical interaction in the target smart home. For this, the adversary remotely learns the devices and installed apps by eavesdropping on the commands and sensor events over network packets and mining their correlations [3, 23]. The adversary can then wait until the physical interactions naturally occur and create unsafe states (e.g., the door is unlocked when the user is not at home) to conduct a physical attack. The adversary can also leverage vulnerable apps to remotely control a set of commands and cause physical interactions [20, 21, 50]. Through this, the adversary can stealthily invoke actuation commands through physical channels even if they cannot directly control them.

The physical interactions might also happen due to the errors in users' creation, installation, and configuration of apps. In such cases, the physical interactions subvert the intended use of IoT devices, leading to unsafe states. This is because IoT users are usually uninformed about the implications of app interactions, as demonstrated by prior works [48, 54].

3 DESIGN CHALLENGES

C1: Correct Physical Interactions. To identify physical interaction vulnerabilities, prior works have used NLP techniques, e.g., the heater is semantically related to temperature [20], manually crafted interaction mappings, e.g., `heater-on` is mapped to the temperature channel [5], and constructed naive device models, e.g., `heater-on` increases temperature by 1℃ in 8 hours [50].

These approaches, unfortunately, discover erroneous interactions or fail to discover them due to over-approximating and under-approximating physical channel properties. Such errors may cause serious consequences. For instance, when the user is not at home, they fail to block `door-unlock` or mistakenly approve `window-open`.

C2: Unintended Physical Interactions. Prior works define security rules based on the use cases of devices to prevent physical interaction vulnerabilities. Such rules do not consider unintended interactions, which occur beyond the intended use of devices and apps, and unexpectedly trigger actions in a smart home. For instance, the user uses App_2 and App_3 in Figure 1 to turn on the light and heater and unlock the patio door when they enter the home. However, when the vacuum robot creates motion, it unintentionally triggers both of these apps and invokes their actions.

Additionally, an actuation command may unintentionally trigger a security rule and subvert its intended use. For example, prior works define a rule that states, "The alarm must sound and an SMS/Push message should be sent to the owner when motion is detected, and home mode is away" to protect the smart home from intruders [21]. This rule can be triggered when App_1 turns on the vacuum robot, which would create panic and unnecessarily bring resources (e.g., police dispatch) to the home.

C3: Run-time Dilemmas. Dynamic systems that examine the device states at run-time [17, 21] cannot infer the influence of an exact command on a physical channel. For instance, in Figure 1, it is unclear to these systems whether the motion is from the vacuum robot or human presence. This challenge becomes more critical when multiple devices influence the same physical channel. For example, if a sound sensor detects the sound from both AC and dryer, the dynamic systems cannot determine if a single device or their aggregated influence changes the sound. This makes them flag incorrect physical interactions.

When an interaction vulnerability is identified, dynamic systems either block device actions or notify users. However, these responses could be dangerous. For example, the door-unlock action might be blocked if there is a fire in the house when the user is not home.

C4: Device Placement Sensitivity. Prior works do not model the impact of the distance between an actuator and a sensor on physical interactions. Intuitively, if the distance between an actuator and sensor increases, the physical influence of a command on sensor readings decreases monotonically. Thus, when a device's placement is changed, the identified interactions may no longer occur, and there may be new interactions that were not previously identified.

This observation causes wrong predictions with false positives (incorrect policy violations and unnecessarily enforcing policies) and false negatives (missing violations and failing to prevent them). This is critical in smart homes as frequent device placement changes may occur with lightweight and portable IoT devices.

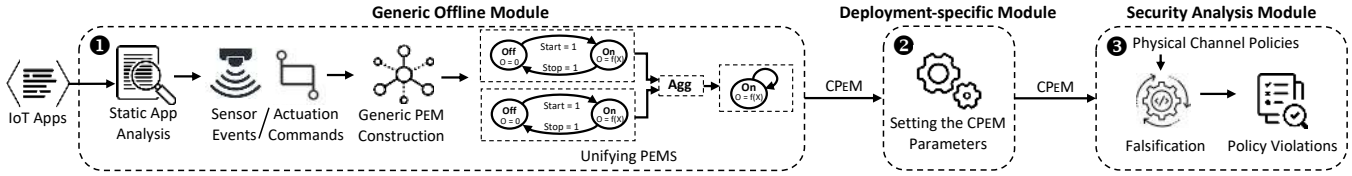


Figure 2: Overview of IoTSeer's architecture

4 IOTSEER DESIGN

To discover physical interaction vulnerabilities, we introduce IoTSeer, which combines app code analysis and dynamic analysis with new security policies, and efficiently addresses the **C1**-**C4** challenges. Figure 2 provides an overview of IoTSeer's modules.

In the *generic offline* module (❶), IoTSeer first extracts actuation commands and sensor events of apps from their source code via static analysis. From this, it builds *physical execution models* (PEMs) for each physical channel a command influences and a sensor measures. Each PEM defines a generic physical behavior of commands and events in hybrid automata with well-studied generic differential and algebraic equations.

IoTSeer then unifies the PEMs in a *composite physical execution model* (CPEM) to represent the joint physical behavior of interacting apps. Our composition algorithm considers a set of physical channel properties (e.g., the aggregation of physical influences) (**C1**) and distinguishes the influence of each command (**C3**).

The offline module delivers PEMs and CPEM that define a generic physical behavior for devices and their composition on the physical channels. However, each smart home may contain devices with different properties (e.g., heater power) and environmental factors (e.g., furniture and room layout).

To address this, in the *deployment-specific* module (❷), IoTSeer extends RSSI-based localization [4, 51] to obtain the physical distance between actuators and sensors. It next collects device traces and leverages system identification techniques to define the execution parameters of CPEM. The resulting CPEM defines the physical behavior of interacting apps for a specific smart home with minimal data collection and addresses the device location changes (**C4**).

In the security analysis module (❸), we first develop security policies to detect unintended physical interactions that cause unsafe and undesired system states (**C2**). IoTSeer then extends optimization-guided falsification to validate if the joint physical behavior of interacting apps conforms to the identified policies. If IoTSeer discovers a policy violation, it outputs the violation's root cause with the interacting apps and physical channels.

Deployment. Our IoTSeer prototype runs in conjunction with the edge device in a smart home (See Figure 3). However, it could be implemented as a software service in the cloud or in a local server.

IoTSeer first obtains the IoT apps and runs its generic offline module (❶). It then collects actuator and sensor traces through the edge device for the deployment-specific module (❷). It next runs its security analysis module and presents users with the policy violations and their root causes (❸).

IoTSeer supports dynamic changes in the smart home, including added, removed, and updated IoT apps and devices (❹). When a dynamic change occurs, IoTSeer reruns its related modules and

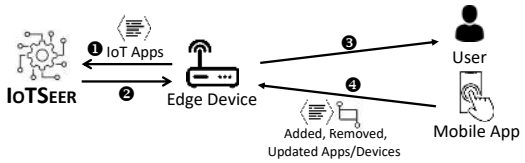


Figure 3: Usage scenario of IoTSeer

presents users with the changed policy violations and their root causes. First, if the user installs a new app or device, IoTSeer runs the generic offline and deployment-specific modules to include the new apps and devices in the CPEM. Second, if the user removes an app or device, IoTSeer removes their PEMs and transitions from the CPEM. Lastly, if an app's configuration or a device's placement changes, IoTSeer changes the CPEM's parameters with the deployment-specific module. After updating the CPEM, IoTSeer runs the security analysis module to identify policy violations.

4.1 Generic Offline Module

To map an IoT app source code to its physical behavior, IoTSeer requires an app's events, actuation commands, and trigger conditions associated with each command. However, IoT platforms are diverse, and each offers a different programming language for automation. For instance, IoT platforms such as OpenHAB enable users to write apps with a Domain Specific Language based on Xbase [39], and trigger-action platforms such as IFTTT implement if-then abstractions [30]. To address this, we leverage existing static analysis and parsing tools for IoT apps [14, 15, 50].

In this way, IoTSeer supports apps from various IoT platforms. These tools model an app's life-cycle, including its entry points and event handlers from its interprocedural control flow graph (ICFG), and extract (1) devices and events, (2) actuations to be invoked for each event, and (3) conditions to invoke the actuations. For instance, given an app *"When the temperature is higher than* 80°*F, if the AC is off, then open the window"*, IoTSeer obtains the `temp > 80` event, the `window-open` command, and the `AC-off` trigger condition.

4.1.1 Constructing PEMs. We translate each command and sensor event of an app to a PEM expressed with a hybrid I/O automaton. This process begins by constructing a separate PEM for each physical channel a command influences, and a sensor event observes with *physics-based modeling*. The physics-based modeling integrates a generic differential or algebraic equation from control theory into a PEM to model each app's physical behavior [31, 49, 55]. This approach is widely used in robotic vehicles [9, 43] (e.g., to predict RV's sensor values) and in autonomous vehicles [13] (e.g., to model the movements of cars and pedestrians).

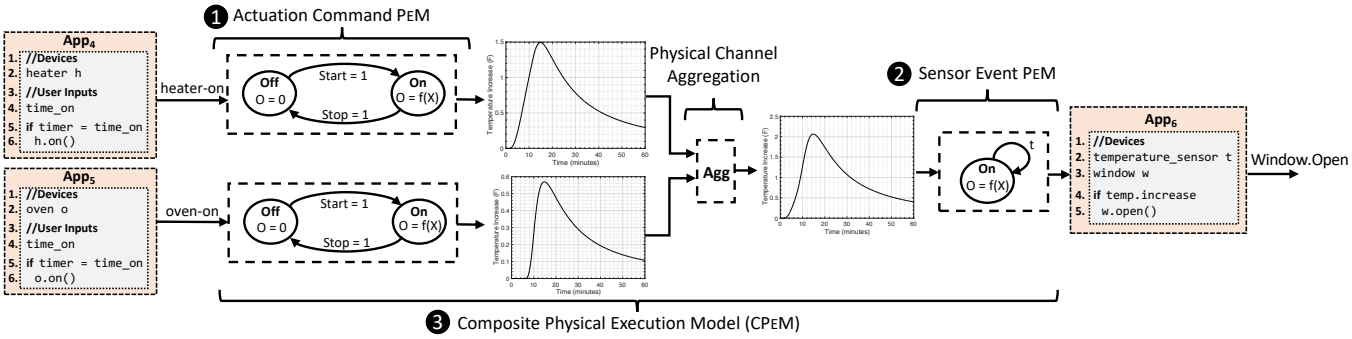


Figure 4: Illustration of PEMs for actuators and sensors, and their CPEM for the unified behavior of three apps (App$_4$, App$_5$, App$_6$).

Determining the physical channels that a command influences requires collecting actuator and sensor traces from the smart home since static app analysis does not reveal the commands' physical channels. IoTSeer initially considers each command may influence all physical channels and removes the over-approximated channels in the deployment-specific module (See Sec. 4.2.2).

PEMs for Actuation Commands. A command PEM defines the discrete and continuous dynamics of a command. The discrete behaviors are an actuator's states (e.g., on/off) for invoking the command from the app. The continuous behavior is an algebraic or differential equation that defines its physical behavior.

Formally, each PEM is a hybrid I/O automaton [34] in the form of $H_a = (Q, X, f, \rightarrow, U, O)$. Here Q is a set of discrete states, X is a continuous variable, f is a flow function that defines the continuous variable's evolution, $(\rightarrow)$ defines the discrete transitions, and U/O defines the input/output variables, as shown in Figure 4-❶. We define the discrete states as $Q = \{on, off\}$, and discrete transitions enable switching between them. The continuous variable defines a command's influence on physical channels (e.g., temperature in °F, sound in dB). The flow function acts on the continuous variable, and the PEM outputs the command's influence.

We define a separate generic flow function for each physical channel. They are *differential* equations for continuous physical channels such as temperature and *algebraic* equations for instant channels such as sound. A flow function takes two parameters as input, device property, and distance from the actuator and outputs the actuator's influence on a physical channel at that distance. These parameters allow us to use the same flow function for different actuators that influence the same channel (e.g., heater-on and AC-on) and the actuators with multiple working patterns (e.g., AC's modes) by setting different parameters.

The property parameter describes the characteristics of a device, such as its operating power. In Sec. 4.2, we show how to set the parameters based on a specific smart home with IoTSeer's deployment-specific module for precision. The distance parameter quantifies the command's influence at different locations (e.g., fan-on's sound intensity at 1 and 2 meters away from the fan). We set this parameter as the distance from the actuator to the sensor that measures its influence (Sec. 4.2). This makes the PEM practical against device placement changes and enables effortless porting of IoTSeer to other deployments with different placements.

Example Actuator PEM. We illustrate a PEM for actuators that influences the temperature channel. The flow function for commands that influence temperature uses the partial differential heat diffusion equation [26], $(\partial T)/(\partial t) = \alpha(\partial^2 T)/(\partial x^2)$ with boundary conditions. Here, T is the environment's temperature (°K), x is the distance parameter (m), α is the thermal diffusivity constant (m^2/s), and the boundary conditions define the actuator's temperature.

Given the device property parameter and the distance to the temperature sensor, the PEM outputs the command's influence on the temperature sensor's measurements over time.

PEMs for Sensor Events. We define an event's PEM as a hybrid I/O automaton (H_s) with a single state, $Q = \{on\}$, and a timed (t) self transition $on \xrightarrow{t} on$, where t is the frequency that a sensor samples its measurements. Figure 4-❷ depicts a sensor event's PEM that only measures physical channels.

The sensor event PEM takes a sensitivity-level parameter, which defines the minimum amount of change in the physical channel (threshold) for a sensor to change its reading. We set the sensitivity level based on the sensors installed in the smart home. A threshold function outputs a sensor reading indicating if the physical channel level is equal to or greater than the sensitivity level. If the sensor measures boolean-typed values (e.g., motion), the PEM outputs a bit indicating "detected" or "undetected" events. If the sensor makes numerical readings (e.g., temperature), it outputs numerical values.

Example Sensor PEM. We illustrate a PEM for a sound sensor that outputs boolean-typed measurements. The threshold function of sound sensor events is defined as $f(sp) = 1$ if $sp > th, 0$ otherwise. where sp is the ambient sound pressure and th is the sensor's threshold (sensitivity level). Here, the PEM outputting 1 means "sound-detected" and 0 means "sound-undetected."

Built-in PEMs. Using the above approach, we have integrated into IoTSeer a total of 24 actuator command PEMs (e.g., heater-on, door-unlock) that influence a total of six physical channels, namely temperature, humidity, illuminance, sound, motion, and smoke, and six sensor event PEMs that measure these channels. The PEMs can be easily extended to define the physical behavior of various devices since their flow functions are generic for a family of devices that influence the same physical channel.

The PEMs allow us to obtain the physical behavior of popular apps used in diverse IoT platforms.

Algorithm 1 Composition of Physical Behavior of Apps

Input: Actuation Command PEMs (H_a), Sensor Event PEMs (H_s), Apps ($\mathcal{L}_{app}$)
Output: CPEM ($\mathcal{M}$)

1: **function** COMPOSITION($H_a, H_s, \mathcal{L}_{app}$)
2: **for** $H_i \in H_a, H_j \in H_s$ **do**
3: UNIFY(H_i, H_j) ▷ Command to sensor event transitions
4: **end for**
5: **for** $app_i \in \mathcal{L}_{app}$ **do**
6: $\langle \mathcal{L}^i_{command}, \mathcal{L}^i_{condition}, \mathcal{L}^i_{event} \rangle$ = STATICANALYSIS(app_i)
7: **for** $s \in \mathcal{L}^i_{event}, a \in \mathcal{L}^i_{command}$ **do**
8: **if** a.condition = true **then**
9: UNIFY(H_s, H_a) ▷ Sensor event to command transitions
10: **end if**
11: **end for**
12: **end for**
13: **for** $a_1 \in \mathcal{L}_{command}$ **do**
14: **if** $a_1 \in \mathcal{L}^i_{event}$ and a_i.condition = true **then**
15: UNIFY(H_{a_1}, H_{a_i}) ▷ Software channel transitions
16: **end if**
17: **end for**
18: **for** $H_j \in H_s$ **do**
19: AGG(H_j.U) ▷ Aggregate inputs of the sensor events
20: **for** $H_k \in H_s$ **do**
21: **if** H_j.O = H_k.U **then** DEP($H_j \rightarrow H_k$) ▷ Dependency
22: **end if**
23: **end for**
24: **end for**
25: **return** $\mathcal{M} = \bigcup (H_a, H_s)$ ▷ Return CPeM
26: **end function**

4.1.2 Unifying the Physical Behavior of Apps. After we build the PEMs for sensors and actuators to define the behavior of each app, we build a separate CPEM to represent their joint behavior.

Algorithm 1 presents our approach to CPEM construction. The algorithm starts with identifying the interacting apps by matching the physical channels of sensor events and commands. First, if a sensor measures a physical channel that a command influences, we add a transition from the command PEM (H_a) output to the sensor event PEM (H_s) input (Lines 2-4). Second, software and physical channels can trigger the event handler of apps and invoke commands if the apps' conditions are satisfied. For physical channels, we add a transition from a sensor event PEM (H_s) to a command PEM (H_a) (Lines 5-12). For software channels, we add a transition from a command PEM (H_{a_1}) to another command PEM (H_{a_2}) if an app invokes a_2 when a_1 occurs (Lines 13-17). The transitions are expressed with a UNIFY operator, which defines the interactions as a transition, $H_a \rightsquigarrow H_s$, $H_s \rightarrow H_a$, and $H_a \rightarrow H_a$. Here, $\rightsquigarrow$ is a physical influence on a channel, and $\rightarrow$ is a software channel.

Figure 4-❸ shows the CPEM of three apps (App_4, App_5, App_6) that automate a heater, oven, window, and temperature sensor. When App_4 invokes heater-on and App_5 invokes oven-on, IOTSEER identifies heater-on's and oven-on's temperature PEMs interact with temperature sensor of App_6. IOTSEER adds the below transitions to the CPEM:

H_a{heater-on} $\rightsquigarrow$ H_s{temp-increase}
H_a{oven-on} $\rightsquigarrow$ H_s{temp-increase}

Another transition from the temperature sensor event PEM to window-open PEM is then added because when the sensor measures an increased temperature, App_6 opens the window.

H_s{temp-increase} $\rightarrow$ H_a{window-open}

Addressing Aggregation and Dependency. A sensor measures the accumulated influence of multiple commands. For this, we define an aggregation operator (AGG), which combines UNIFY(H_a, H_s) operators so that a sensor event PEM takes the aggregated output of command PEMs as input (Lines 18-19). Turning to Figure 4, IOTSEER adds AGG(H^1_a{heater-on}, H^2_a{oven-on}) $\rightsquigarrow$ temp-increase transition to the CPEM. The AGG operator's output is defined based on the physical channel's unit. It is the sum of the command PEM outputs, $\sum_{i=1}^{n}$ UNIFY(H^i_a, H_s), for linear scale channels (e.g., temperature) [55]. The channels in the log scale (e.g., sound) are aggregated after being converted to a linear scale, $10 \times \log_{10}(\sum_{i=1}^{n} 10^{H^i_a/10})$ [36].

Another property of physical channels is that a physical channel (p_j) may depend on another channel (p_i) if a change in p_i affects p_j. Due to dependencies, a sensor event PEM's output may influence another sensor event PEM's readings. The generic PEMs allow us to easily identify dependencies by iteratively taking each sensor event PEM and checking if it is used in the threshold function of another sensor event (Lines 20-23). For instance, when ambient temperature increases, the air-water capacity increases and affects the humidity sensor's readings [33]. To address this, we add a DEP($H^i_s \rightarrow H^j_s$) transition from the temperature sensor event PEM output (H^i_s) to the humidity sensor event PEM input (H^j_s).

4.2 Deployment-specific Module

The actuation command PEMs require distance and device property parameters, and the sensor event PEMs require a sensitivity level parameter. We set these parameters based on the devices installed in the smart home to ensure the CPEM precisely models the physical behavior of the apps.

4.2.1 Setting the Distance Parameter. To determine the distance parameter in PEMs, we initially considered leveraging a recent IoT device localization tool, Lumos [45]. Lumos localizes IoT devices with high accuracy by requiring the user to walk around the smart home with a mobile phone. However, this approach could be inconvenient for smart home users since it requires manual effort.

To address this, we integrate received signal strength intensity (RSSI)-based distance estimation techniques [4, 51] into IOTSEER. Such techniques leverage the inverse proportion between distance and RSSI to estimate the distance between two devices. Although this approach may incur an error in the distance parameter, our evaluation shows that the impact of such errors on IOTSEER's policy violation identification is minimal (See Sec. 5).

4.2.2 Setting the Device Property Parameter. We consider two options for setting the device property parameters based on the installed devices. The first is using the installed device's datasheets. Yet, in our prototype implementation, we realized that the datasheets might be incomplete, or a discrepancy could occur, for example, due to device aging [44]. To address this, we extend System Identification (SI), a learning-based method commonly used by control engineers to estimate parameters or models of physical processes using experimental data traces [32, 42].

SI allows us to estimate device property parameters that ensure the CPEM achieves high fidelity with actual devices. This process requires *fewer traces* than the traditional application of SI as we only estimate parameters instead of complete equations (See Sec. 5). We particularly use (τ, ϵ)-closeness [2] as the fidelity metric. (τ, ϵ)-closeness determines the difference between two traces in their timing (τ) and values (ϵ), where ϵ is referred to as deviation score.

Table 1: Descriptions of intent-based policies to discover physical channel vulnerabilities in Figure 5.

ID*	Formal Representation†	Policy Description	Security Goal
G_1	$\square(\texttt{imp}(\langle \texttt{UnInt}, \texttt{p}\rangle) \leq \texttt{th})$	An actuation command's influence on a physical channel must not unintentionally trigger an app's sensor event and invoke its device actions.	Prevent attackers from creating and exploiting individual unintended physical interactions.
G_2	$\square(\texttt{imp}(\langle \texttt{UnInt}_1, \texttt{p}\rangle, \ldots, \langle \texttt{UnInt}_n, \texttt{p}\rangle) \leq \texttt{th})$	Multiple commands' aggregated influence on a physical channel must not unintentionally trigger an app's sensor event and invoke its device actions.	Prevent attackers from creating and exploiting aggregated unintended physical interactions.
G_3	$\square(\texttt{imp}(\langle \texttt{Int}_1, \texttt{p}\rangle \ldots \langle \texttt{Int}_k, \texttt{p}\rangle) < \texttt{th}) \rightarrow \square(\texttt{imp}(\langle \texttt{Int}_1, \texttt{p}\rangle, \ldots, \langle \texttt{Int}_k, \texttt{p}\rangle, \langle \texttt{UnInt}_1, \texttt{p}\rangle, \ldots, \langle \texttt{UnInt}_n, \texttt{p}\rangle) < \texttt{th})$	If commands' intended influences do not trigger an app's sensor event, their aggregation with other commands' unintended influences must not trigger it.	Prevent attackers from bypassing intended physical interactions.

∗ The IDs correspond to the vulnerabilities in Figure 5 (G_1 for (a), G_2 for (b), and G_3 for (c))
† imp() denotes an actuation commands' labeled influence on a channel p. Multiple channels in imp denotes aggregated influences. th denotes a sensor's sensitivity.

To apply this approach in a smart home, IoTSeer individually activates each actuator and collects sensor measurements. It next runs their PeMs with a device property parameter and obtains sensor traces. It computes the (τ, ϵ)-closeness between the actual device and PeM traces. It conducts a binary search on the device property parameter to obtain the optimal value that minimizes the deviation score. Using real device traces to determine the device property parameters ensures that the impact of environmental conditions (e.g., furniture) on sensor readings is integrated into the CPeM.

From the collected actual device traces, IoTSeer also determines the set of physical channels that a command influences in the smart home. IoTSeer checks if a command does not change the sensor measurements or if its influence is statistically indistinguishable from environmental noise. In such a case, IoTSeer removes the PeMs of those commands from the CPeM.

4.3 Security Analysis Module

We first identify intent-based policies to detect unintended app interactions and device-centric policies to detect the vulnerabilities intended interactions present (Sec. 4.3.1). IoTSeer then leverages falsification to validate the identified policies on the CPeM (Sec. 4.3.2).

4.3.1 Identifying Physical Channel Policies. To properly designate the circumstances under which the physical interactions are a *vulnerability* or *feature*, we define *intended* and *unintended* labels.

We define the physical channel between an actuator and an app as intended if the app is installed to be triggered from that command's influence. For instance, consider a user that installs an app that turns on the AC when the temperature exceeds a threshold. If the temperature from the oven triggers this app, the channel between oven-on and the app is intended, as the oven causes the temperature to exceed the threshold defined by the user.

We define a physical channel between an actuator and an app as unintended if it is unplanned by a system or undesired by a user. For instance, consider a user that installs an app with the goal of unlocking the patio door with the motion from her presence. If the motion from the vacuum robot triggers this app, the channel between robot-vacuum-on and the app is unintended, as unlocking the patio door due to the vacuum robot is not desired by the user.

Intended/Unintended Label Generation. IoTSeer automates generating the interaction labels based on the use cases of apps. It also allows users to change the labels as their needs dictate.

To generate the labels, IoTSeer first checks whether the intended use of an app is related to a specific activity. IoTSeer leverages SmartAuth [47], an NLP-based technique that extracts the activity related to an app from its description.

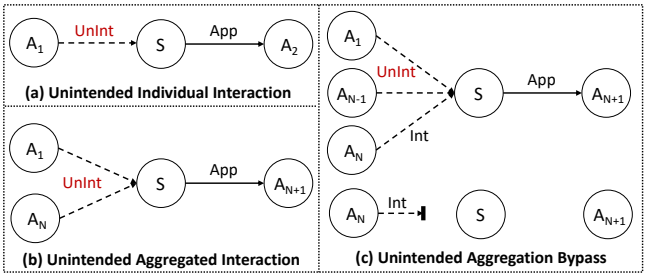


Figure 5: Intent-based physical channel vulnerabilities (A is an actuation command, and S is a sensor event).

Consider an app that states *"open the windows when you are cooking"* in its description, and the app is triggered when the temperature sensor's readings exceed a threshold. SmartAuth outputs that this app is related to the cooking activity. IoTSeer takes the app's activity and checks whether any of the actuators installed in the smart home are semantically related to the activity. For this, IoTSeer uses Word2Vec representations to compute the semantic distance between the activity and the commands. It then assigns intended (Int) label to the commands with a distance lower than a threshold and assigns unintended (UnInt) labels to others. For instance, the cooking activity is semantically related to oven-on and cooker-on commands, and thus, IoTSeer assigns Int to them.

If an app's description does not indicate an activity or none of the commands are semantically related to the activity, IoTSeer assigns the labels based on the apps' sensor events. The apps conditioned on motion or sound sensors' events are used to detect the presence of users and intruders in smart homes. For instance, $\texttt{App}_2$ in Sec. 2, which unlocks the patio door and turns on the lights when motion is detected, is used to be triggered with user presence. IoTSeer assigns UnInt label to all commands for such apps because only the influences from users and intruders are intended for them.

The apps conditioned on temperature, humidity, smoke, and illuminance channels are installed to be triggered when the physical channel's state reaches a specific condition. For instance, an app that turns on the AC when the temperature is higher than a threshold controls the AC based on the ambient temperature level. IoTSeer assigns Int label to all commands for such apps because their intended use only depends on the physical channel conditions, regardless of which commands influence them.

Intent-based Policies. Based on generated labels, we define three *intent-based* policies, as shown in Figure 5, which are used to identify unintended physical interactions that create undesired and

Table 2: Example device-centric policies.

ID	Policy Description	Formal Representation
DC_2	When the home is in the away mode, the window must be closed.	$\square$(mode-away $\rightarrow$ window-close)
DC_3	A device must not open, then close and then reopen (actuation loop) within t seconds.	$\square\neg$(on $\wedge \bigcirc\Diamond_{[0,t]}$(off $\wedge \bigcirc\Diamond_{[0,t]}$on))
DC_5	The alarm must go off within t seconds after smoke is detected.	$\square$(smoke-detected $\rightarrow \Diamond_{[0,t]}$alarm-on)
DC_6	The main door must not be left unlocked for more than t seconds.	$\Diamond_{[0,t]}$door-lock
DC_9	The door must always be locked and lights must be off when the home is in the away mode.	$\square$(mode-away $\rightarrow$ door-lock $\wedge$ light-off)

unsafe system states. We present the security goal of each policy and its expression with Metric Temporal Logic (MTL) in Table 1. We validate the policies on the CPEM in the next section.

The first policy, G_1, unintended individual interaction, states that an actuation command's unintended influence on a physical channel must not trigger an app's sensor event and invoke its device actions (Figure 5(a)). For instance, the robot vacuum's motion must not trigger an app that unlocks the patio door when a motion-detected event occurs. This is because an adversary who can invoke the start robot vacuum action (e.g., through a vulnerable app) can exploit this interaction to indirectly unlock the patio door.

The second policy, G_2, states the aggregated influence from multiple commands must not unintentionally trigger an app and invoke its device actions (Figure 5(b)). Although a command's individual influence may not trigger an app, its aggregation with another command's influence may trigger it. For example, the aggregated sound of AC-on and dryer-on must not trigger an app that sounds an alarm when the sound-detected event occurs, and the home mode is away.

The last policy, G_3, states if the commands' intended influences do not trigger an app's sensor event, their aggregation with other commands' unintended influences must not trigger it (Figure 5(c)). For instance, if the light bulb's Int influence on illuminance does not create a light-detected event and trigger apps, its aggregation with the TV's UnInt influence must not trigger the apps as well.

These unintended physical interactions, by definition, are not features as they are not desired by users. Yet, an adversary can exploit them to indirectly control devices and cause unsafe states.

Device-Centric Policies. While intent-based policies detect unsafe states from unintended physical interactions, Int labeled physical channels can also cause unsafe states. For instance, the heater's intended influence on the temperature sensor may trigger an app that opens the windows when the temperature exceeds a threshold.

To address such violations, we extend the security rules of previous works [15, 17, 21] (and enhance them with time-constrained temporal operators in MTL) to define device-centric policies. We present a subset of device-centric policies in Table 2. For instance, the DC_5 policy states that physical interactions must not prevent an alarm from going off within *two secs* after smoke is detected ($\square((\text{smoke} > \text{th}) \rightarrow \Diamond_{[0,2]}(\text{alarm} = \text{ON}))$, where $\square$ is always, and $\Diamond_{[0,2]}$ is eventually within next 2 secs).

4.3.2 Validating Policies on CPEM. After identified policies are expressed with MTL, IoTSEER executes the CPEM (hybrid I/O automaton) and collects actuator and sensor traces to validate policies.

Algorithm 2 Grid-Testing

Input: CPEM ($\mathcal{M}_{H_a,H_s}$) with command PEMs (H_a) and sensor event PEMs (H_s), parameters (x - distances among devices), inputs (U - apps' activation times $t_0 : \Delta t : t_{end}$), policy (ψ).
Output: P=(inputs, apps, dist, atime, y)

1: **function** GRID_TEST(H_a, H_s, x, U, $\mathcal{M}_{H_a,H_s}$, ψ)
2: **for** j $\in H_s, H_{OP} \subseteq H_a$ **do**
3: **for** Different activation times in H_{OP} **do**
4: **if** $\Phi(\mathcal{M}_{H_{OP},H_S}, x, u) \not\models \psi$ **then**
5: $P \leftarrow P \cup \{x, u, \Phi(\mathcal{M}_{H_{OP},H_S}, x, u)\}$
6: **end if**
7: **end for**
8: **end for**
9: **return** P
10: **end function**

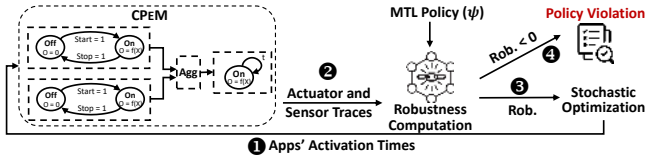


Figure 6: Overview of falsification to find policy violations.

At each execution, the CPEM takes apps' activation times as input, which is the time when the app invokes its commands and the command PEM transitions to the "on" state. The CPEM simulates the unified physical behavior of commands and sensor events and outputs traces of PEMs. The traces (v, t) are composed of a periodic timestamp t, and a physical channel value v. Each command PEM's v shows how much it influences a channel, and each sensor event PEM's v shows its measurements. The traces also include labels (Int/UnInt) and app IDs of commands/events for root cause analysis.

Policy Validation Challenges. The physical channel values and the app activation times are continuous; thus, the CPEM's state space becomes infinite, which makes formal verification approaches (e.g., model checking) undecidable on the CPEM [6, 27, 41].

To address this issue, we initially implemented a grid-testing approach, a commonly applied method for testing CPS and autonomous vehicle software [19, 38, 56]. Grid-testing determines whether the CPEM satisfies a policy under a finite set of apps' activation times– the times that apps invoke actuation commands. Algorithm 2 presents the grid-testing approach on the CPEM for policy validation. We set apps' activation times as a grid ($t_0 : \Delta t : t_{end}$) (Line 3). The algorithm executes the CPEM with a search on activation time combinations. It then validates a policy on each execution's traces from PEMs with a robustness metric (Lines 4-6), where negative robustness values indicate a policy violation.

However, we found that grid-testing does not scale larger analyses with the increasing number of interacting apps and may miss policy violations due to input discretization. To address these, we extend optimization-guided falsification and compare it with grid-testing in identifying violations and performance overhead in Sec. 5.

Optimization-Guided Falsification. Falsification is a formal analysis technique that searches for a counterexample to an MTL policy from a continuous input set [1, 7]. Figure 6 depicts our approach in leveraging falsification to search for interacting apps that cause a policy violation on the CPEM.

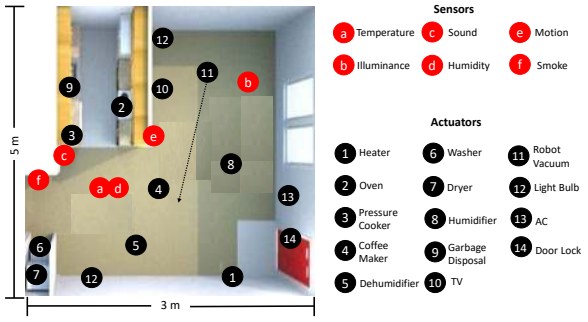


Figure 7: Sensor/actuator layout in the actual house.

Specifically, we use an optimization algorithm to search for policy violations by sampling activation times (❶). We then execute the CPeM and record actuator and sensor traces from PeMs (❷). From the traces, we compute a robustness value that quantifies how close an MTL formula is to the policy violation (❸). Positive robustness values indicate the policy is satisfied, and negative values indicate it is violated. The sampler then seeds another input to the CPeM within the ranges (similar to input mutation in fuzzing [24]). The sampler's objective is minimizing the robustness to find a policy violation (❹). The termination criteria for input generation is when the policy is violated or a user-defined maximum number of iterations is met.

```
"Policy Violation": {
  "Input": robot-vacuum-on,
  "Apps":
    "app_a": e:= timer, a:= robot.vacuum-on,
    "app_b": e:= mot-active, a:=patDoor.unlock,
  "Activation Time": 10,
  "Distance": 2,
  "Physical Channel Values":
    "t = 10": robot-vacuum.move,
    "t = 11": (UnInt) mot-active,
    "t = 11": patDoor.unlock}
```

Listing 1: An example output of policy validation.

When IoTSeer identifies a policy violation, it outputs a quintuple, $O = (\texttt{inputs}, \texttt{apps}, \texttt{dist}, \texttt{atime}, \texttt{v})$, that details the policy violation's root cause. Here, `atime` is the activation time of `apps`, `v` is `Int`, `UnInt` labeled command PeM and sensor event PeM outputs, and `dist` is the distance from actuators to sensors. Listing 1 presents the output of a policy violation when $\texttt{App}_a$ turns on a robot vacuum and interacts with $\texttt{App}_b$ that unlocks the patio door when it detects the robot vacuum's motion. The output further details the violation occurs when `robot-vacuum-on` is activated at minute `10` when it is 2 meters away from the motion sensor. In Sec. 6, we detail how this output can be used to mitigate the violation.

5 EVALUATION

We evaluate IoTSeer in a real home with six sensors and 14 actuators, as shown in Figure 7. To automate the devices, we study three IoT app markets, IFTTT, Microsoft Flow, and SmartThings, and install 39 popular apps. We then invoke each actuation command and collect sensor readings from actual devices in the house for 30 mins to identify the channels each command influences (See Table 3)[1].

[1]We have consulted our university's IRB office and got advised that IRB approval is not required since we do not collect any sensitive information.

Table 3: Physical channels of studied actuators and sensors.

	Sensors					
Actuator (Actuation Command)	**Temp.**	**Illum.**	**Sound**	**Hum.**	**Motion**	**Smoke**
Marley Baseboard Heater (set(val))	✓	✗	✗	✓	✗	✗
Kenmore AC (set(val))	✓	✗	✓	✓	✗	✗
CREE Smart Light Bulb (on)	✗	✓	✗	✗	✗	✗
Instant Pot Pressure Cooker (on)	✓	✗	✗	✓	✗	✗
Mr. Coffee Coffee maker (on)	✓	✗	✗	✓	✗	✗
Sunbeam Humidifier (on)	✗	✗	✗	✓	✗	✗
Easy Home Dehumidifier (on)	✗	✗	✗	✓	✗	✗
Whirlpool Clothes Washer (on)	✗	✗	✓	✓	✗	✗
Whirlpool Dryer (on)	✓	✗	✓	✓	✗	✗
Whirlpool Garbage Disposal (on)	✗	✗	✓	✗	✗	✗
Roborock S4 Robot Vacuum (on)	✗	✗	✗	✗	✓	✗
Vizio 48" TV (on)	✗	✓	✓	✗	✗	✗
Door (unlock)	✗	✗	✓	✗	✗	✗
Whirlpool Oven (on)	✓	✗	✗	✓	✗	✗

✗ means the command does not influence the physical channel the sensor observes and ✓ means the command influences it.

We built a total of 24 PeMs for channels the commands influence (number of physical channels the actuators influence in Table 3) and six sensor event PeMs for the channels they observe (temperature, illuminance, sound, humidity, motion, and smoke). We also use the collected sensor readings to conduct SI and tune the CPeM parameters. Lastly, we assign `Int`/`UnInt` labels between the commands and apps based on the apps' intended uses, as described in Sec. 4.3.1.

We implement grid-testing and our falsification approach using an open-source temporal logic toolbox, S-TaLiRo [8]. We leverage S-TaLiRo's `dptaliro` function as a subroutine in grid-testing and `staliro` function in our optimization-guided falsification to validate MTL policies on the CPeM. We use the `staliro` function with simulated annealing by hit-and-run Monte Carlo sampling for input generation [22]. We use a dynamic programming-based algorithm to compute the robustness of MTL policies. In grid-testing's implementation, we consider apps invoke actuation commands at `10`-min intervals; thus, the apps' activation times are set as a grid to $\texttt{0} : \texttt{10} : \texttt{60}$. In falsification's implementation, we set the apps' activation times as continuous ranges, any time in the execution (`0-60`), and we define the max number of tests to `100` as we do not observe a significant change in robustness after `100`.

We run the CPeM executions on a laptop with a 2.3 GHz 2-core i5 processor and 8 GB RAM, using Simulink 10.0.

5.1 Effectiveness

We validate intent-based and device-centric security policies on the CPeM tuned for our home.

Table 4 presents 16 identified policy violations caused by physical interactions among seven different groups of devices. We compare the violations flagged by IoTSeer with prior works that identify physical interaction vulnerabilities and show they can discover 2 out of 16 violations. We also conduct in-home experiments with real devices and confirm that all 16 violations are true positives.

5.1.1 Intent-based Policy Violations. IoTSeer identified 14 interactions that subvert the apps' intended use, causing unsafe states.

Individual Policy (G_1) Violations. IoTSeer flagged 10 individual policy (G_1) violations that occur due to the physical interactions among three groups of devices. First, the motion sensor detects the presence of the robot vacuum and unintentionally triggers five apps

Table 4: Policy violations identified by IoTSeer and previous works.

Policy	ID	App Interactions	Number of Violations	Violation Description	Existing Work		
					iRuler	IoTMon	IoTSafe
G_1	V_1	robot-vacuum-start $\xrightarrow{\text{motion-det.}}$ door-unlock, light-on, heater-on, TV-on, call-user	5	Robot vacuum's motion *unintentionally* triggers five apps and causes door-unlock, heater-on, light-on, TV-on and call-user actions.	✗	✗	✗
	V_2	garb-disp-on $\xrightarrow{\text{sound-det.}}$ light-on, TV-on, call-user	3	Garbage disposal's sound *unintentionally* triggers three apps and causes light-on, TV-on and call-user actions.	✗	✗	✗
	V_3	TV-on $\xrightarrow{\text{sound-det.}}$ light-on, call-user	2	TV's sound *unintentionally* triggers two apps and causes light-on and call-user actions.	✗	✗	✗
G_2	V_4	Agg(AC-on, washer-on, dryer-on) $\xrightarrow{\text{sound-det.}}$ light-on, TV-on, call-user	3	Aggregated sound from the AC, washer and dryer *unintentionally* triggers three apps and causes light-on, TV-on and call-user actions.	✗	✗	✗
G_3	V_5	Agg(bulb-on, TV-on) $\xrightarrow{\text{light-det.}}$ light-off	1	Aggregated light from the bulb and TV *bypasses* bulb's intended influence and triggers an app that turns off the lights.	✗	✗	✗
DC_8	V_6	sleep-mode-activate $\xrightarrow{\text{sleep-mode}}$ AC-on $\xrightarrow{\text{temp<th° F}}$ light-on	1	Temperature decreases due to the AC's influence and turns on the bulb when the home mode is sleep.	✗	✓	✓
DC_9	V_7	away-mode-activate $\xrightarrow{\text{away-mode}}$ vacuum-start $\xrightarrow{\text{motion-det.}}$ heater-on $\xrightarrow{\text{temp>th° F}}$ light-on	1	Temperature increases due to the heater's influence and turns on the bulb when the home mode is away.	✗	✓	✓

conditioned on the motion-detected event. For instance, this violation occurs when App_{17} starts the robot vacuum when the home mode is set to away, and App_{27} unlocks the door when motion is detected. Second, the sound sensor detects the garbage disposal's (V_2) and TV's (V_3) sound, unintentionally triggering apps conditioned on sound-detected. For example, App_9 turns on the garbage disposal at a user-defined time, and App_{19} notifies the user when sound is detected, creating unnecessary panic.

Aggregation Policy (G_2) Violations. IoTSeer flagged three aggregation policy (G_2) violations among one group of devices. The sound sensor outputs a sound-detected event due to the unintended aggregated influence from AC-on, washer-on, and dryer-on. This, in turn, triggers three apps conditioned on the sound-detected event and causes light-on, TV-on, and call-user actions.

Bypass Policy (G_3) Violations. IoTSeer identified a single bypass policy (G_3) violation. The illuminance sensor measures the aggregated illuminance of light-on and TV-on. The increase in illuminance triggers App_{34}, turning off the lights. However, App_{34}'s intended operation is turning off the lights when the daylight is enough to illuminate the environment, which is semantically related to the light-on action. Therefore, light-on's influence on App_{34}'s sensor event is intended, whereas TV-on's influence is unintended. Since light-on's individual influence cannot trigger App_{34}'s light-detected event but its influence aggregated with the unintended influence from TV-on triggers it, the app's intended use is bypassed.

5.1.2 Device-Centric Policy Violations. IoTSeer identified two device-centric policy violations, V_6 between three apps and V_7 between four apps. In V_6, while the home is in sleep mode, the AC's influence on temperature triggers App_{23} to turn on the bulb. This violates DC_8 since the bulb is turned on when the home mode is sleep. In V_7, two physical interactions occur due to UnInt and Int channels. While the home mode is away, the vacuum's UnInt motion triggers App_{26} to turn on the heater (❶). The heater's Int temperature then triggers App_{23} and turns on the bulb, violating DC_9.

5.1.3 Comparison with Previous Work. In Table 4, we compare the policy violations flagged by IoTSeer with the most applicable approaches, iRuler [50], IoTMon [20], and IoTSafe [21], that run on IoT app source code to identify physical interaction vulnerabilities.

To identify physical interactions among apps, iRuler uses device behavioral models (e.g., AC-on decreases the temperature by 1°C every hour), and IoTMon mines the apps' text descriptions (e.g., finds AC is semantically related to temperature). If we assume they correctly map all physical channels that each command influences in our smart home, iRuler cannot identify any of IoTSeer's violations, and IoTMon can identify 2 out of 16 violations. This is because (1) their policies cannot reason about the intended use of apps, (2) they do not consider the complex physical properties such as aggregation and dependency, and (3) iRuler does not consider device-centric vulnerabilities. Additionally, IoTMon would flag 18 false positives as most commands do not individually cause physical interactions. To illustrate, it defines a physical channel between the temperature sensor and oven; yet oven-on's individual influence on temperature is not enough to cause an interaction.

IoTSafe models the apps' physical behavior through dynamically collected sensor traces and predicts physical channel values at run-time for policy enforcement. Compared to IoTMon, IoTSafe does not flag any false positives since it relies on sensor traces collected from real devices. However, it can also detect only 2 out of 16 violations in our smart home. This is because, as a run-time enforcement system, it cannot infer the influence of an exact command on a physical channel. Additionally, its policy enforcement may create unnecessary panic in our home since the robot vacuum's motion would trigger its policy that sounds an alarm and sends a message to the user when motion is detected.

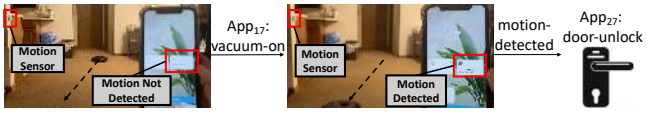


Figure 8: Illustration of the V_1 violation.

5.1.4 In-Home Validation Experiments. We repeated each identified violation in the actual house and confirmed that they (V_1-V_7) are true positives. In-home validation begins by analyzing each violation's root cause through their (`inputs`, `apps`, `dist`, `atime`, `y`) logged by IoTSeer. We activate each actuator involved in the violation using its `inputs` and `atime`. We record the traces and confirm the interacting `apps`. Lastly, we compare traces from PeMs and devices to identify any differences in the time the violations occur.

We observe that policy violations on continuous physical channels (temperature) occur later in the house compared to CPeM. For instance, when `heater-on` is invoked at time `0` in the CPeM, IoTSeer flags a DC_9 violation at second `56`. However, we observe the violation at second $\approx$ `65` in the house. In contrast, violations on instant channels (motion, sound, and illuminance) occur with minor time deviations. These slight timing deviations are expected due to inevitable environmental noise.

Case Study. We present a case study to illustrate a violation and detail an attack scenario demonstrating how an adversary can exploit the observed physical interaction. Figure 8 depicts the V_1 violation; a `motion-detected` event occurs due to the `robot-vacuum-start` command and triggers App_{27} that unlocks the door. To exploit this, an adversary can leverage a vulnerability in the vacuum controller app to start it and stealthily unlock the door. The adversary can also wait until the user sets the home mode to away, which triggers App_{17} that turns on the vacuum and causes the door to unlock. Through this attack, the adversary can break into the house.

5.2 Violations with Device Placement Changes

We change the placement of illuminance, sound sensors, and the TV and use IoTSeer to identify the policy violations that occur with the new placement. We select these devices as they are easily relocated and potentially impact nine policy violations.

IoTSeer correctly identifies that three physical channels that caused policy violations in the initial placement do not occur anymore, and it discovers two new physical channels causing violations. We then evaluate the impact of the distance parameter's accuracy in identifying policy violations and show that IoTSeer only misses a single violation when the distance parameter has a 50% error.

Violations After New Device Placement. Table 5 presents the physical channels that cause policy violations after the device placement changes. IoTSeer identified three physical channels (V_2, V_4, and V_5) that caused policy violations before do not occur in the new device placement. On the contrary, `TV-on` command still creates a `sound-detected` event, unintentionally triggering two apps.

IoTSeer also flagged two new physical channels that cause policy violations, where the washer's and dryer's sound cause G_1 violations since the sound sensor was moved closer to them. We confirmed with in-home experiments that all identified policy violations with the new device placement are true positives.

Table 5: The physical channels that cause policy violations after new device placement, and their tolerance to errors.

Physical Channel	Tolerance to Distance Error
`washer-on` ⇝ `sound-detected`	20%
`dryer-on` ⇝ `sound-detected`	> 50%
`TV-on` ⇝ `sound-detected`	> 50%

IoTSeer's Tolerance to Errors in Distance Parameter. Errors in the distance parameter may occur due to slight deviations in the accuracy of IoT device localization tools and RSSI-based localization techniques. Such errors do not impact IoTSeer's ability to discover the initial policy violations since IoTSeer integrates SI to tune the PeMs based on real device traces. However, the errors may impact IoTSeer's effectiveness in identifying violations after a device's location is changed. Thus, we introduce errors ranging from ±`10%-50%` to the distance parameter when the placements of devices are changed and check if IoTSeer outputs any false positive or negative violations. We select these error rates because IoTSeer's RSSI-based distance estimation gives, on average, a 22.8% error in our setup, and the only error higher than 50% is the estimation between the TV and sound sensor (due to the wall between devices).

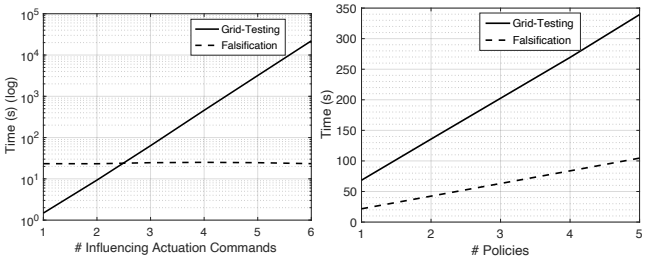
Table 5 presents the maximum error in distance parameters under which IoTSeer can still correctly identify the violations. For instance, if IoTSeer estimates the distance between the washer and sound sensor with a 20% error, it can still correctly identify these devices' policy violations. However, if the error is larger than 20%, IoTSeer would miss the violations, causing false negatives. On the contrary, IoTSeer's dryer and TV PeMs are more tolerant to error, where they correctly identify the violations even under 50% error. This is because these devices' influences on sound are higher, enabling IoTSeer to identify their interactions.

We further checked whether the errors in the distance parameter cause any false positives, where IoTSeer flags violations that do not actually occur. We found that IoTSeer identifies two false positives if the distance error is 50%, (1) the `bulb-on`'s illuminance PeM, and (2) the `AC-on`'s sound PeM. Yet, such high errors are unlikely in practice since IoTSeer integrates state-of-the-art localization techniques.

5.3 Performance Evaluation

5.3.1 Scalability Experiments. We evaluate the policy validation time of IoTSeer's falsification algorithm and compare its results with the baseline approach of grid-testing. Grid-testing discretizes the times when apps activate actuators as a grid and validates policies with all combinations of activation times. Although grid-testing identifies all 16 violations in our experiments, it yields a high time overhead, as detailed below.

CPeM Size vs. Time. Figure 9a shows the policy validation time of the grid-testing and falsification with an increasing number of command PeMs influencing the same channel in the CPeM. Testing time exponentially increases with the number of commands as each policy is validated using a combination of apps' activation times. In contrast, falsification has a near-constant time overhead as it samples activation times and searches for low robustness values for a single violation. This adds a negligible delay, on the order of seconds, with an increasing number of commands.



(a) (b)

Figure 9: (a) #Actuation commands and (b) #Policies vs. time.

Number of Policies vs. Time. We evaluate the validation time with an increasing number of policies. We set the number of command PeMs to three, and a sensor measures their aggregated physical influence. Figure 9b shows the time overhead of both testing and falsification increases linearly with the number of policies. Here, falsification is $\approx 3\times$ more efficient than testing as testing validates the policies with all combinations of apps' activation times.

5.3.2 Time for Device Trace Collection. We present the time for actuator and sensor trace collection from actual devices to tune CPeM parameters. It took 7 hours to record the measurements from actual sensors, which required turning on each actuator and collecting traces from all sensors in the smart home. This is an improvement over generating generic flow functions by SI solely using device traces, which requires $\approx$ 175 hours of data collection with different device properties and distances.

6 DISCUSSION & LIMITATIONS

Mitigating the Policy Violations. IoTSeer identifies policy violations and presents users with a report that details the violation's root cause. However, there is a need to mitigate the policy violations to ensure the safe and secure operation of the smart home. We discuss three methods for mitigation, (1) patching the app code, (2) device placement changes, and (3) removal of apps.

The first mitigation technique is patching the app code to block its commands if the app is triggered due to an unintended influence. This technique adds a code block that guards an app's action with a predicate conditioned on the devices that unintentionally influence the app's sensor event. If a device is unintentionally influencing a channel, the predicate becomes false, preventing the app from issuing its command. For instance, the V_4 violation is prevented by adding a predicate to the apps conditioned on `sound-detected`. The predicate blocks apps' actions if the AC, washer, and dryer are simultaneously turned on, preventing the unintended interactions.

Second, we recommend users increase the distance between the actuator and sensor to prevent policy violations. This is because a command's influence on sensor readings monotonically decreases as the distance between the actuator and sensor increases [49, 55]. For instance, operating the robot vacuum away from the motion sensor (e.g., by setting 'keep out zones') prevents the motion from `vacuum-on` from triggering events. Lastly, removing one or more apps involved in the physical interaction prevents a violation. A user may prefer this method if an app is not critically needed and other mitigation techniques are not feasible.

Table 6: Mitigated policy violations with different methods.

	Policy			
Mitigation Method	**G1**	**G2**	**G3**	**DC**
Patching the App Code	10/10	3/3	1/1	0/2
Device Placement Changes	10/10	3/3	1/1	2/2
Removal of Apps	10/10	3/3	1/1	2/2

For the 16 identified policy violations, we applied the above mitigation methods and evaluated their effectiveness. Table 6 shows the number of policy violations prevented by our mitigation methods. First, patching the app code prevents 14 out of 16 violations as it inserts predicates that guard the unintended physical interactions. For instance, V_1 is patched by adding a condition to the apps triggered by the motion detected event. The condition checks whether the robot vacuum is not on before sending the app's commands. Second, changing the device placement prevents all violations. We validate this through in-home experiments with increased distance between the devices in the policy violations. For instance, increasing the distance from the garbage disposal to the sound sensor prevents V_2 as `garbage-disposal-on`'s sound cannot reach the sensor. Finally, removing the apps involved in the physical interactions prevents all violations, e.g., removing a single app from the three apps that cause the V_6 violation prevents it.

We note that our mitigation methods may prevent desired actions while eliminating dangerous app interactions. First, code patching blocks actions while actuators that unintentionally influence a physical channel are turned on, yet, a user may desire to issue those actions. In such cases, the user can manually activate them and respond to the app interactions. Second, changing a device's placement may be inconvenient for the user, and it may cause other policy violations. However, IoTSeer can identify new policy violations by updating its distance parameters and running its security analysis module. Lastly, removing an app eliminates all of its interactions; however, users may not desire to remove the apps they need. In future work, we will conduct user studies to learn how users perceive the mitigation methods and investigate advanced techniques for automated patching. For instance, we will explore automated distance range discovery through parameter mining [29] to find the specific distance ranges between actuators and sensors that can prevent all policy violations.

Manual Effort Required. IoTSeer requires users' effort in determining the distance parameter in the CPeM and the `Int/UnInt` labels for intent-based policies. First, the users need to confirm that the distances found from RSSI-based localization are correct and provide the distances manually if necessary. This effort is not a significant burden for users since they can provide approximate distances. This is because IoTSeer can tolerate small errors in the distance parameter, as shown in our evaluation. Second, although IoTSeer generates `Int/UnInt` labels between commands and apps, the users may have different intentions than the generated ones. Therefore, they need to check the labels and update them if necessary based on their intended use of the actuators and apps.

Table 7: Comparison of IoTSeer with IoT security systems.

System	Physical Channel Properties					Security Analysis	
	Dist.	Agg.	Dep.	Labels (Int/UnInt)	Composition	Time Constraints	Policy Validation†
IotGuard* [17]	✗	✗	✗	✗	✗	✗	RA
IoTSafe* [21]	✗	✓	✓	✗	✗	✗	RP
MenShen [12]	✗	✗	✗	✗	✗	✗	RA
iRuler [50]	✗	✗	✗	✗	✗	✗	RL
IoTCom [5]	✗	✗	✗	✗	✗	✗	MC
IoTMon [20]	✗	✗	✗	✗	✗	✗	N/A
IoTSeer	✓	✓	✓	✓	✓	✓	F

∗ IoTGuard and IoTSafe are run-time policy enforcement systems.
† RA: Reachability Analysis, RL: Rewriting Logic, MC: Model Checking, F: Optimization-guided Falsification, RP: Run-time Prediction.

Environmental Noise. The environment that the devices operate in may influence the physical channels and impact the app interactions. As IoTSeer leverages SI to tune CPeM parameters, it integrates various environmental impacts such as room layout and furniture. Yet, human activities and uncontrolled environmental noise may also influence sensor measurements. To measure the impact of human activities, we conducted additional experiments while a user was cooking and exercising. We have found that the users do not introduce detectable changes to sensors. We have shown in in-home validation experiments that uncontrolled noise causes the violations to occur at slightly different times.

7 RELATED WORK

In Table 7, we compare IoTSeer with several recent approaches that focus on identifying the vulnerabilities that IoT app interactions present. These approaches can be classified into two categories: run-time policy enforcement and static analysis for IoT apps.

Run-time Policy Enforcement. IoTGuard [17] instruments apps to build dynamic models and enforces policies at run-time. Yet, it cannot correctly identify physical interactions since its models do not include the commands' physical influences.

IoTSafe [21] collects actuator and sensor traces to identify physical interactions and builds physical models for continuous physical channels to predict incoming policy violations at run-time. However, it cannot identify intent-based violations as its policies are only defined based on the use cases of devices, and it cannot determine the specific command that influences a physical channel from examining sensor measurements at run-time. Additionally, IoTSafe does not consider distance in its models and flags incorrect violations or fails to detect a violation when a device's placement is changed. These systems motivate the need for IoTSeer, which can precisely identify dangerous physical interactions before the smart home's run-time operation.

Static Analysis of IoT Apps. Existing static analysis systems do not model apps' complex physical behavior. Instead, they build individual physical channel mappings, generate naive device behavioral models [12, 50], or use natural language processing [20] to infer interacting apps. Thus, they identify limited physical interactions and lead to false positives. As presented in Table 7, IoTSeer is the first to integrate the complex physical properties of commands and sensor events into the source code of IoT apps ("Physical Channel Properties" Columns). Additionally, their validation techniques cannot readily be used to verify physical interactions as apps exhibit both discrete and continuous behaviors. In contrast, IoTSeer extends optimization-guided falsification for scalable policy validation.

8 CONCLUSIONS

We introduce IoTSeer[2], which identifies the physical channel vulnerabilities in smart homes. IoTSeer combines static app analysis with system identification to precisely model the composite physical behavior of apps and uses falsification to validate identified physical channel policies. Our evaluation in a real house demonstrates that many apps interact over physical channels, and IoTSeer efficiently and effectively identifies all policy violations. This paper is an important step forward in achieving the compositional safety and security of an IoT system's physical behavior.

ACKNOWLEDGMENTS

We would like to thank Engin Masazade and Ali Cem Kizilalp for their feedback on the earlier version of this paper. This work has been partially supported by the National Science Foundation (NSF) under grants CNS-2144645, 1901242, and 1910300, DARPA VSPELLS under grant HR001120S0058, Rolls-Royce Cyber Technology Research Network Award, National Natural Science Foundation of China (No. 61702263), and the scholarship from China Scholarship Council (No. 201906845026). The views expressed are those of the authors only.

REFERENCES

[1] Houssam Abbas, Georgios Fainekos, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2013. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS).*
[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] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A Selcuk Uluagac. 2018. Peek-a-Boo: I see your smart home activities, even encrypted! *arXiv preprint arXiv:1808.02741.*
[4] Omotayo G Adewumi, Karim Djouani, and Anish M Kurien. 2013. RSSI based indoor and outdoor distance estimation for localization in WSN. In *IEEE International Conference on Industrial Technology (ICIT).*
[5] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis of interaction threats in IoT systems. In *ACM SIGSOFT International Symposium on Software Testing and Analysis.*
[6] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. 1995. The algorithmic analysis of hybrid systems. *Theoretical Computer Science.*
[7] Yashwanth Singh Rahul Annapureddy and Georgios E Fainekos. 2010. Ant colonies for temporal logic falsification of hybrid systems. In *Annual Conference on IEEE Industrial Electronics Society.*
[8] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer.
[9] Ardupilot SITL 2022. Ardupilot Simulation. https://ardupilot.org/dev/docs/simulation-2.html. [Online; accessed 18-April-2022].
[10] Musard Balliu, Massimo Merro, and Michele Pasqua. 2019. Securing cross-app interactions in IoT platforms. In *IEEE Computer Security Foundations Symposium.*
[11] Musard Balliu, Massimo Merro, Michele Pasqua, and Mikhail Shcherbakov. 2020. Friendly Fire: Cross-App Interactions in IoT Platforms. *ACM Transactions on Privacy and Security (TOPS).*
[12] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. 2018. Systematically ensuring the confidence of real-time home automation IoT systems. *ACM Transactions on Cyber-Physical Systems.*

[2]The development of IoTSeer and its evaluation in a real IoT environment was a highly complex endeavor. This paper's extended version is available with (1) detailed PeM implementations, (2) a complete list of device-centric policies, (3) descriptions of IoT apps installed in our evaluation, and (4) CPeM fidelity experiment results [40].

[13] Carla Physics 2022. Carla - Control and Monitor Vehicle Physics. https://carla.readthedocs.io/en/latest/tuto_G_control_vehicle_physics/. [Online; accessed 18-April-2022].

[14] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2019. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys (CSUR)*.

[15] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT safety and security analysis. In *USENIX Annual Technical Conference (USENIX ATC)*.

[16] Z Berkay Celik, Patrick McDaniel, Gang Tan, Leonardo Babun, and A Selcuk Uluagac. 2019. Verifying internet of things safety and security in physical spaces. *IEEE Security & Privacy*.

[17] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *NDSS*.

[18] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. Cross-app interference threats in smart homes: Categorization, detection and handling. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[19] Anthony Corso, Robert J Moss, Mark Koren, Ritchie Lee, and Mykel J Kochenderfer. 2020. A survey of algorithms for black-box safety validation. *arXiv preprint arXiv:2005.02979*.

[20] 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)*.

[21] Wenbo Ding, Hongxin Hu, and Long Cheng. 2021. IoTSafe: Enforcing Safety and Security Policy with Real IoT Physical Interaction Discovery. In *NDSS*.

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

[23] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. 2021. HAWatcher: Semantics-Aware Anomaly Detection for Appified Smart Homes. In *USENIX Security*.

[24] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*.

[25] Furkan Goksel, Muslum Ozgur Ozmen, Michael Reeves, Basavesh Shivakumar, and Z Berkay Celik. 2021. On the safety implications of misordered events and commands in IoT systems. In *IEEE Security and Privacy Workshops (SPW)*.

[26] Matthew J Hancock. 2006. The 1-D heat equation. *MIT OpenCourseWare*.

[27] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. 1998. What's decidable about hybrid automata? *J. Comput. System Sci.*

[28] HomeKit 2022. Apple's HomeKit. https://www.apple.com/ios/home/. [Online; accessed 30-April-2022].

[29] Bardh Hoxha, Adel Dokhanchi, and Georgios Fainekos. 2018. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *International Journal on Software Tools for Technology Transfer*.

[30] IFTTT 2022. IFTTT (If This Then That). https://ifttt.com/. [Online; accessed 18-April-2022].

[31] AG Jackson, SJP Laube, and J Busbee. 1996. Sensor principles and methods for measuring physical properties. *JOM*.

[32] Karel J Keesman. 2011. *System identification: an introduction*. Springer Science & Business Media.

[33] Mark G Lawrence. 2005. The relationship between relative humidity and the dewpoint temperature in moist air: A simple conversion and applications. *Bulletin of the American Meteorological Society*.

[34] Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Hybrid I/O automata. *Information and Computation*.

[35] Sunil Manandhar, Kevin Moran, Kaushal Kafle, Ruhao Tang, Denys Poshyvanyk, and Adwait Nadkarni. 2020. Towards a Natural Perspective of Smart Homes for Practical Security and Safety Analyses. In *IEEE Symposium on Security and Privacy (S&P)*.

[36] Fedor Mitschke. 2009. Decibel units. In *Fiber Optics*. Springer.

[37] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IoTSan: Fortifying the safety of IoT systems. In *International Conference on Emerging Networking Experiments and Technologies*.

[38] Justin Norden, Matthew O'Kelly, and Aman Sinha. 2019. Efficient black-box assessment of autonomous vehicle safety. *arXiv preprint arXiv:1912.03618*.

[39] OpenHab 2022. OpenHAB: Open Source Automation Software for Home. https://www.openhab.org/. [Online; accessed 30-April-2022].

[40] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z. Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. 2022. Discovering IoT Physical Channel Vulnerabilities. *arXiv preprint arXiv:2102.01812*.

[41] Erion Plaku, Lydia E Kavraki, and Moshe Y Vardi. 2009. Falsification of LTL safety properties in hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

[42] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org. http://ptolemy.org/books/Systems

[43] PX4 SITL 2022. PX4 Simulation. https://docs.px4.io/master/en/simulation/. [Online; accessed 18-April-2022].

[44] Philippe Réfrégier. 2004. *Noise theory and application to physics: from fluctuations to information*. Springer Science & Business Media.

[45] Rahul Anand Sharma, Elahe Soltanaghaei, Anthony Rowe, and Vyas Sekar. 2022. Lumos: Identifying and Localizing Diverse Hidden IoT Devices in an Unfamiliar Environment. In *USENIX Security*.

[46] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *International Conference on World Wide Web*.

[47] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. Smartauth: User-centered authorization for the internet of things. In *USENIX Security*.

[48] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In *CHI Conference on Human Factors in Computing Systems*.

[49] Nikolaos Voudoukis and Sarantos Oikonomidis. 2017. Inverse square law for light and radiation: A unifying educational approach. *European Journal of Engineering Research and Science*.

[50] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[51] Giovanni Zanca, Francesco Zorzi, Andrea Zanella, and Michele Zorzi. 2008. Experimental comparison of RSSI-based localization algorithms for indoor wireless sensor networks. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks*.

[52] Zapier 2022. Zapier: Connect your apps and automate workflows. https://zapier.com/. [Online; accessed 30-April-2022].

[53] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

[54] Valerie Zhao, Lefan Zhang, Bo Wang, Shan Lu, and Blase Ur. 2020. Visualizing Differences to Improve End-User Understanding of Trigger-Action Programs. In *CHI Conference on Human Factors in Computing Systems*.

[55] Alexander Zhivov, Hakon Skistad, Elisabeth Mundt, Vladimir Posokhin, Mike Ratcliff, Eugene Shilkrot, and Andrey Strongin. 2001. Principles of air and contaminant movement inside and around buildings. In *Industrial Ventilation Design Guidebook*. Elsevier.

[56] Aditya Zutshi, Jyotirmoy V Deshmukh, Sriram Sankaranarayanan, and James Kapinski. 2014. Multiple shooting, cegar-based falsification for hybrid systems. In *International Conference on Embedded Software*.