IoTSafe: Enforcing Safety and Security Policy with Real IoT Physical Interaction Discovery

Wenbo Ding
Clemson University
wding@g.clemson.edu

Hongxin Hu
University at Buffalo
hongxinh@buffalo.edu

Long Cheng
Clemson University
lcheng2@clemson.edu

***Abstract*—The Internet of Things (IoT) platforms bring significant convenience for increased home automation. Especially, these platforms provide many new features for managing multiple IoT devices to control their physical surroundings. However, these features also bring new safety and security challenges. For example, an attacker can manipulate IoT devices to launch attacks through unexpected physical interactions. Unfortunately, very few existing research investigates the physical interactions among IoT devices and their impacts on IoT safety and security. In this paper, we propose a novel dynamic safety and security policy enforcement system called IoTSafe, which can capture and manage real physical interactions considering contextual features on smart home platforms. To identify real physical interactions of IoT devices, we present a runtime physical interaction discovery approach, which employs both static analysis and dynamic testing techniques to identify runtime physical interactions among IoT devices. In addition, IoTSafe constructs physical models for temporal physical interactions, which can predict incoming risky situations and block unsafe device states accordingly. We implement a prototype of IoTSafe on the SmartThings platform. Our extensive evaluations demonstrate that IoTSafe effectively identifies 39 real physical interactions among 130 potential interactions in our experimental environment. IoTSafe also successfully predicts risky situations related to temporal physical interactions with nearly 96% accuracy and prevents highly risky conditions.**

I. Introduction

Internet of Things (IoT) technologies are revolutionizing home automation. The rise of smart home development platforms, such as Samsung SmartThings [38] and Google Android Things [2], enables fast development of smart home IoT applications (apps). Despite the convenience offered by IoT technologies, new safety and security concerns emerge in smart home environments [30], [3]. Recent research reveals many safety and security problems on either IoT devices and platforms, such as device firmware flaws [37], [12], protocol flaws [23], [36], [27], [46], information leakage [22], [9], malicious applications [21], [42], [6], and side channel attacks [35], [33]. In particular, the safety and security problems caused by interactions of IoT apps/devices have attracted much attention recently [18], [11], [29], [10], [14].

The interactions in an IoT environment (*e.g.*, a collection of apps and devices working together to form a smart home environment) can be broadly classified into two categories:

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24368
www.ndss-symposium.org

- *Cyberspace interactions:* Apps interact through a common device or system event in the cyber space (*e.g.*, `switch.on/off` or `home-mode`). For example, an app turns on the light after sunset, and another app unlocks the door when the same light is turned on [11]. These two apps interact via the `light.on` event on the same device and share a common device attribute (*i.e.*, software variable) in the cyber space of an IoT platform. We use the term *cyberspace interaction* to refer to cross-app interactions where multiple apps subscribe/operate the same device.
- *Physical interactions:* One unique feature of IoT is that apps/devices could make impacts on the physical environment. Such physical impacts enable apps that subscribe *different* devices to externally interact with each other through shared physical environment channels (*e.g.*, temperature, luminance, and humidity) [18], [7], instead of through a common device attribute. For instance, a heater-control app turns on a heater, and a temperature-control app opens windows when the temperature sensor detects that the temperature is higher than a pre-defined threshold. In this case, the temperature channel connects the heater and the temperature sensor to generate a *physical interaction*.

Cyberspace interactions can lead to unsafe and insecure states in a multi-app IoT system. To evaluate whether an IoT environment is safe and secure, and operates correctly, Soteria [10] and IoTSan [29] develop a set of reference policies[1] in terms of the safety, security, and functionality in the IoT environment. Then, they use model checking techniques to verify the desired safety/security properties by assuming multiple apps are installed together to operate the same devices. Soteria and IoTSan mainly focus on identifying unsafe or insecure conditions created by cyberspace interactions, *e.g.*, when multiple apps change the same device attributes in conflicting or unexpected manners.

Physical interactions can also leave users in unsafe and unexpected situations, and could potentially be exploited by adversaries to launch IoT attacks. For example, given a temperature-triggered window control app, a burglar may manipulate the temperature to maliciously trigger a window opening action, which can lead to a potential *break-in*. To capture potentially risky physical interaction chains across apps, IoTMon [18] performs a static analysis of 185 official SmartThings apps and identifies 162 hidden inter-app interactions through physical surroundings. However, static analysis based approaches only capture *potential* safety and security problems, rather than *runtime* policy violations in real-world

[1]Policies here represent properties that an app must satisfy to make sure an IoT environment is safe and secure, which can be defined by IoT users or developers.

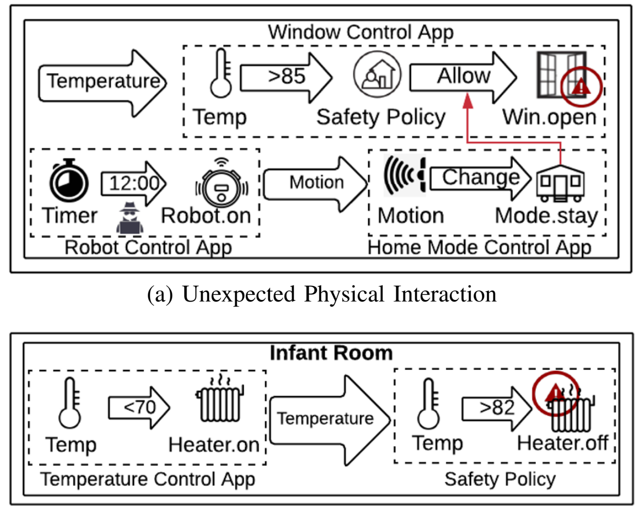
IoT deployments. Different from static analysis of multi-app IoT environments in [10], [29], [18], [14], IoTGuard [11] is a dynamic policy enforcement system for IoT environments. It blocks unsafe and undesired states by monitoring the runtime behaviors of IoT apps. However, like Soteria and IoTSan, IoTGuard mainly considers *cyberspace* interactions in multi-app IoT environments, without capturing real *physical* interactions among different IoT devices. Another limitation of IoTGuard is that it enforces policies when device states are approaching to unsafe conditions (*e.g.*, the temperature is already close to a critical level), which is often too late to prevent from damaging impacts on smart home environments. In this work, we fill this gap and focus on *dynamic physical* interaction control with risky situation prediction in multi-app IoT environments.

There are several challenges in characterizing real physical interactions in smart home environments. Regarding the first challenge, existing static approaches [10], [29], [18], [14] cannot be applied to identify *real* physical interactions. For example, IoTMon [18] uses text mining of app descriptions to identify common physical channels between IoT devices and then discover all *potential* physical interactions among devices. Thus, it cannot be applied to detect *real-time* physical interactions in dynamic physical interaction control. In fact, multiple factors, such as spatial location, device influence range, and environment condition, can affect device physical interactions. For instance, if a heater and a temperature sensor are placed in different rooms, it is likely there is no physical interaction between them. Another major challenge is to consider the *temporal* aspect of physical interactions. Some physical interactions among IoT devices may happen immediately, *e.g.*, turning on a light immediately triggers an illuminance sensor. On the contrary, there exist physical interactions happening *slowly* but *continuously*. For example, only when a toaster keeps running for a sufficiently long time, it will cause a smoke alert. However, it is non-trivial to capture these *contextual* features of IoT physical interactions.

In this work, we present IoTSafe, a dynamic safety and security policy enforcement system for multi-app IoT environments to protect users from unsafe and insecure IoT device interactions in a preventive manner. We design a real physical interaction discovery approach employing both static analysis and dynamic testing techniques to efficiently capture real-world physical interactions among IoT devices. The goal of our dynamic testing is to test run-time physical interactions with respect to both spatial/temporal effects and device/condition restrictions of IoT environments, and meanwhile minimize the overhead (*e.g.*, time consumption) of the whole process. IoTSafe also builds physical models for temporal physical interactions to predict future states and enforces safety/security policies if a risky situation is likely to happen.

Our main technical contributions are summarized as follows:

- We propose an approach to capture *real* physical interactions among IoT devices, which are missed in previous approaches to IoT security. Our method employs both static analysis and dynamic testing techniques to discover run-time physical interactions considering contextual features of IoT environments, and also minimize the overhead of the testing process.
- Our system predicts the risky situations that are caused by temporal physical interactions based on physical models, generates warnings and suggestions to users to prevent unsafe/insecure situations from happening, and thus effectively improves the safety and security of IoT environments.
- We evaluate our system on the SmartThings platform running in *real-world* smart home environments. We identify 39 real physical interactions among 130 static interactions in a typical three-room home structure. For the dynamic policy enforcement, our system successfully predicts risky situations related to temporal physical interactions with nearly 96% accuracy and prevents 53 highly risky conditions with given 36 user policies.



(a) Unexpected Physical Interaction

(b) Continuous Physical Interaction

Fig. 1: Examples of risky physical interactions.

II. Motivation & Threat Model

A. Problem Statement

In an IoT environment, physical interactions among IoT devices can lead to unsafe and insecure situations that could be potentially exploited by attackers [18]. Figure 1(a) shows an example of the inter-app physical interaction in a smart home environment where an attacker exploits a robot vacuum to trigger window opening via the motion channel. Suppose three apps, a robot control app, a window control app, and a home mode control app, have been installed in the smart home environment. The window control app opens the window in a room when its temperature exceeds 85F. The home mode control app sets the home mode to "`STAY`" when a movement is detected. In this example, the temperature is raised above 85F to trigger a window opening action, which may leave home in a potentially unsafe situation, such as beak-in. To eliminate such a potential risk, a safety policy, "Do not open window when no one is at home", is set. Enforcing this policy can prevent the window opening action being triggered by the temperature sensor when no movement is detected at home. However, the attacker can bypass this safety policy by exploiting the robot vacuum or its control app, such as setting the robot vacuum to work at 12:00pm, to change the home mode through an unexpected physical interaction, where the robot vacuum's movement can be detected by a motion sensor. Then, the home mode control app changes the home mode to "`STAY`", which allows the window opening action although no one is at home.

The current policy enforcement systems in IoT environments [11] also ignore *continuous* effects of physical interactions, which may miss or delay the policy enforcement and result in unsafe situations. The *continuous* effect means some devices are still capable of changing environment parameters even after they have been turned off. As shown in Figure 1(b), in an infant room, a temperature control app turns on a heater when the temperature is below 70F. Suppose the user sets a safety policy “Turning off heaters when the temperature exceeds 82F” to prevent that the temperature of the infant room is too high. However, after turning off the heater, the infant room could still reach to a higher temperature, such as 85F, and maintain it for about half an hour (as demonstrated in Section V-C). The *continuous* effect of the heater may lead to unsafe situations, and harm the baby in the infant room.

Therefore, it is crucial to identify *real* physical interactions to correctly enforce the safety and security policies in IoT environments. The existing work, IoTMon [18], performs a static analysis of apps to identify *potential* physical interactions without considering the deployment context information of IoT devices in real IoT environments. Thus, IoTMon cannot be directly applied to detect real-time physical interactions for runtime policy enforcement. In addition, identifying real physical interactions need to consider more complex factors, such as room structure, device influence range, and environment condition.

1) Challenges in Identifying Real Physical Interactions: Enforcing safety and security policies at runtime can mitigate the risk caused by IoT device physical interactions through blocking unsafe and undesired IoT device states/actions. To address such a dynamic policy enforcement problem, the policy enforcement system needs to be aware of *real* physical interactions among IoT devices. We identify four challenges in capturing real physical interactions of IoT devices, as demonstrated in Figure 2.

- **Spatial Context.** Spatial context refers to the location information of IoT devices. The interaction between devices are sensitive to the location or room status. For example, in Figure 2(a), the temperature channel enables an interaction between the heater and the temperature sensor, and possibly leads to an unexpected action of opening the window. However, the interaction between the heater and sensor are highly relying on their locations. A temperature sensor is unlikely to detect the temperature change caused by the heater if they are not in the same room.
- **Temporal Context.** Some physical interactions among devices may happen immediately, while there exist physical interactions happening *slowly*. For example in Figure 2(b), turning on a light immediately triggers an illuminance sensor event. On the contrary, the temperature sensor can only observe increased temperature after a heater running for a sufficiently long time. Intuitively, the temperature, humidity, and smoke physical channels usually take time to remarkably change the physical status.
- **Implicit Effect.** Two types of implicit effects exist in smart home environments. One is that a device action may have *multiple* physical impacts on other devices. For example, in Figure 2(c), the `thermostat.heating` not only increases the *temperature*, but also implicitly reduces the *humidity*. Another type of implicit effects is caused by the physical interactions that current static analysis methods [11], [18] cannot identify. Because the current static analysis methods mainly discover physical interactions based on app descriptions, they can only find physical interactions associated with the physical channels explicitly defined in a device control app. For example, in Figure 1(b), the robot control app mainly explains the cleaning functions of a robot vacuum in its description, but does not provide any description about its potential effect on the *motion* physical channel.
- **Joint Effect.** If there are multiple devices functioning together, the joint physical influence differs from the behavior of individual devices. *e.g.*, when a heater and a thermostat's heating function at the same time as shown in Figure 2(d), the temperature is increased rapidly to trigger the `window.open`.

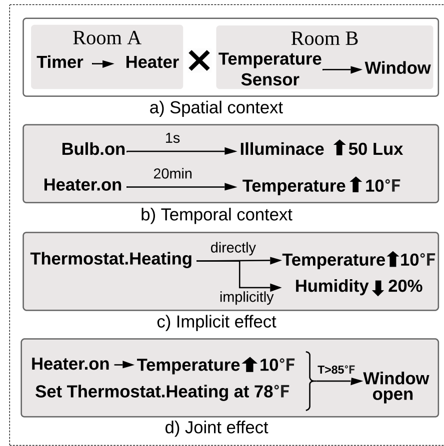


Fig. 2: Challenges of identifying real physical interactions

Physical interactions in a real IoT setting are often context-sensitive, and these contextual factors make capturing real physical interactions non-trivial. Existing works [10], [29], [18], [14] are unable to capture real physical interactions among IoT devices because they do not consider the run-time physical environment of IoT platforms. The physical influence of a device varies based on context, such as environmental attributes and surrounding devices. As such, we need a systematic *dynamic testing* method to identify real and context-sensitive physical interactions among IoT devices in order to support dynamic enforcement of safety and security policies.

2) Challenges in Temporary Physical Interaction Control: One challenge for the temporal effect is that devices (*e.g.*, heaters) may have *continuous* effects after being turned off, which makes the policy violation possible even after the enforcement of a control policy. As shown in Figure 1(b), the temperature in an infant room may exceed 82F after the heater being turned off, which could leave the infant room in a risky situation.

Another challenge for the temporal effect is caused by the long reporting interval of commodity IoT sensors. Most smart home sensors feature low cost and long battery life, which inevitably results in long reporting intervals. For example, regarding the temperature channel, the Aeotec MultiSensor 6 [4] has a report interval from 10 minutes to 1 hour, and Zooz 4-in-1 Sensor's report interval [5] ranges from 8 minutes to 1 hour. With such long report intervals, if a critical policy violation happens, the system may not enforce the policy in a timely manner. Using highly sensitive sensors may avoid the long interval issue, but they are often very expensive with high energy consumption. For example, a sensitive Marcell sensor costs $179 and can only last 2 days.

Because of the continuous effect combined with a low reporting rate of sensors, it is desirable to *model* and *predict* the physical effect of certain IoT devices so as to early assess their physical influences before they make any damaging consequences.

B. Threat Model

In this paper, we mainly consider the indirect attack caused by physical interactions where the failure of the safety and security policy enforcement can happen in the smart home platform. The failure is mainly due to the unexpected or unreal physical interaction of IoT devices that can cause the false/delayed enforcement of safety and security policies.

We assume the unexpected physical interactions are caused by malicious apps or compromised devices under certain constraints, which include: 1) attackers may not have a permission to directly control sensitive apps (a window control app); and 2) sensitive devices (a lock) are possibly more robust and cannot be easily compromised by attackers. By only exploiting physical interactions, attackers can indirectly control sensitive devices through other less-sensitive apps and easily-compromised devices. We assume attackers can launch their attacks through: 1) vulnerable or malicious IoT apps, which contain vulnerable implementations or malicious code that can be easily exploited by attackers to gain access remotely to control devices; and 2) vulnerable devices (a smart plug), which contain design flaws that can be abused to trigger surrounding devices through unexpected physical interactions.

Due to the unawareness of physical interactions, current smart home policy enforcement systems may falsely enforce or delay the enforcement of safety and security policies. For example, when a high temperature is detected, the policy enforcement system may enforce the window opening policy to open windows instead of turning off heaters. It is also possible to have delayed enforcement of safety and security policies due to sensor reporting limitations in the smart home platform. This false or delayed enforcement can also be caused by continuous, implicit, and joint effects of physical interactions, which cannot be directly identified by current static analysis approaches.

In addition, we assume IoT platforms and policy enforcement systems are tamper-proof and trustworthy, and cannot be penetrated. Also, we consider the safety and security policies defined by users or developers are trustworthy. We further assume those safety and security policies are conflict-free.

III. SYSTEM OVERVIEW

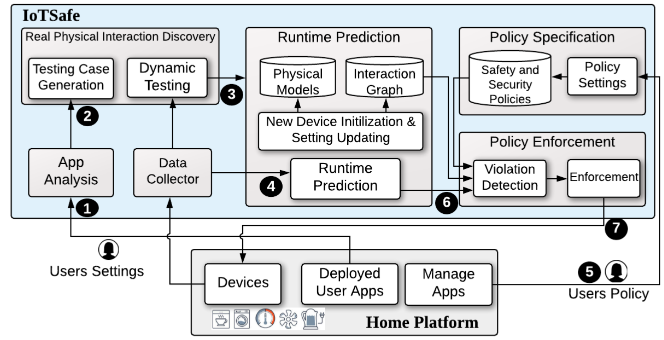


Fig. 3: IoTSafe system overview.

Figure 3 shows the workflow of IoTSafe, which is composed of four major components: i) App Analysis; ii) Real Physical Interaction Discovery; iii)Runtime Prediction; and iv) Policy Specification and Enforcement.

App Analysis: The purpose of this module is to analyze control flows in apps to facilitate IoTSafe's physical interaction discovery through dynamic testing and collect app's user settings/configurations at runtime (❶). The static control flow of typically apps involves trigger conditions, actions, and user settings of apps. The code analysis module extracts trigger conditions and corresponding actions as "static interaction" flows and generates a static interaction graph. It also provides coarse-grained device groups based on the room information of devices (configured by IoT users). At the end, the static analysis module collects user configurations when apps are deployed and generates fine-grained interaction graphs.

Real Physical Interaction Discovery: This module aims at efficiently identifying *real* physical interactions given smart home settings. After apps are deployed, IoTSafe uses app configurations (*e.g.*, a temperature threshold that triggers a heater's action) and room information to generate testing cases (❷). Based on the generated cases, IoTSafe identifies device physical interactions by performing dynamic testing with respect to device/condition restrictions. The dynamic testing includes both sequential testing and parallel testing (❸). The spatial, implicit and joint effects are analyzed by a comprehensive sequential testing process. The parallel testing is introduced to reduce the time consumption of the testing process. IoTSafe generates a directed interaction graph, which includes device states and their transitions. We also provide a method to avoid testing risky devices' behaviors and use online usage data to identify their interactions.

Runtime Prediction: This module initializes and maintains physical models to characterize temporal effects of physical interactions. It also monitors runtime events from the data collector and compares the current status with the physical interaction model to predict future status (❹). During the dynamic testing process, IoTSafe records the temporal interaction data for related sensors. Hence, the modeling process first initializes physical models based on the data collected during the testing phase. For untested devices and newly added devices, IoTSafe uses online usage data to train their interaction graphs and physical models. If the user modifies apps' settings, IoTSafe also updates interaction graphs based on the new locations, trigger conditions, or actions. During normal usage, IoTSafe continues to update the physical models when more device data is received.

Policy Specification and Enforcement: The policy enforcement mechanism uses a control server that is in charge of identifying policy violations by checking app events and actions against a set of user-defined policies. IoTSafe first requires users to setup their policies through a policy management app, which can be based on given policy templates, or write their own policies (❺). The data collection component/app collects runtime information from devices and sends data to both runtime prediction and violation detection modules. Violation detection is based on comparing current/predicted situations with user-defined policies (❻). If a violation is about to happen immediately or within a period, the policy enforcement component takes actions to relieve from the risky situation, *e.g.*, sending warnings or turning off the related heater to avoid a high-temperature situation (❼).

IV. System Design

A. App Analysis

The purpose of static analysis is to identify basic potential interactions for dynamic testing. The static analysis is composed of code analyzer and interaction graph builder. Different from existing works [18], [29], [11], our static analysis targets practically deployed apps in a smart home, where user settings and the room information of devices are known. IoTSafe first performs a code analysis to extract the dependency graph of deployed apps. It also instruments codes for the user setting extraction and policy enforcement module. Then the interaction graph builder uses specific user settings, such as devices id, trigger conditions, and room information to build inter-app interaction graphs.

1) Code analysis and instrumentation: The code analyzer [18] first extracts the inter-procedural dependency of devices within an app by static analysis. Combined with a physical channel analysis, the analyzer could build inter-app dependency between apps and devices. Then it patches the app's source code for runtime data collection, such as sending device ids and user configurations. It also adds extra logic for runtime interaction control [25]. Different from existing work [18], we need to consider the runtime user setting information (*e.g.*, room and device IDs) in the static analysis, which helps optimize the dynamic testing process. Besides, similar to the code instrumentation implemented in [25], [11], we instrument apps with policy enforcement functionalities for physical interaction control.

```
<name: "Turn on It When Water Detected">
def installed()
{   subscribe(waterSensor1, "waterSensor",
    waterHandler)
    info = getdeviceInfo()
    sendRequest(info)
    }
def waterHandler(evt)
{   actions = "switch.on()"
    resp = sendRequest(evt,actions)
    if(resp == 1) {switch.on()}
}
def sendRequest(param)
{   def result = true
    result = httpGet(url, path) {param}
    return result  //Server Response
}
def getdeviceInfo()
{ def deviceInfo = []
    settings.waterSensor1.each{
      deviceInfo << it.displayName
        deviceInfo << it.deviceId}
    return deviceInfo
}
```

Listing 1: An example of app instrumentation.

For code instrumentation, we mainly instrument two sections, 1) the device configuration for interaction graphs, and 2) the app's action commands for runtime policy enforcement, which we have gathered a list of device commands from Samsung SmartThings documents [39]. Similar to ContexIoT [25], our static analysis searches for exact commands from an app's source code and instruments additional control logic to targeted functions and commands. The newly added control logic sends commands and related device information to a control server and waits for responses. Listing 1 provides the pseudo-code of "Turn on It When Water Detected" app as an example to illustrate our app analysis and instrumentation. This app turns on a switch when water is detected by a leakage sensor. In this example, we add extra functions for the data collection in the `installed()` function, which collects user setting information and sends them to the server for the interaction graph generation. The `getdeviceInfo()` function collects device configuration information and passes it to the `httpPost()` function. For the runtime policy enforcement, we instrument additional control logic for the `waterHandler()` function. The instrumented code in the "`sendRequest()` function sends the "switch.on" action and the related water sensor event to the server and wait for a response. It has a return value, which can be used to receive the response from the policy enforcement server. Only if the server approves the following action, the app resumes executing the switching on commands.

2) Interaction Graph Generation: Existing works [18], [29], [11] have proposed methods for building inter-app dependency graphs. Our work considers more information such as user settings, room information to build a more fine-grained interaction graph. The interaction graph generation contains two main functions. First, it extracts specific user settings, device ids, and room information. Second, it divides the whole static interaction graph into separated sub-graphs based on additional user settings.

The configuration information needs to be extracted from each deployed app, including i) device id; ii) numerical values (*i.e.*, thresholds) in trigger conditions; and iii) numerical values for attributes in actions (*e.g.*, cooling temperature settings for thermostat); and iv) room tag information. As shown in Listing 1, we insert the configuration collection code in the `installed()` function of each app. The inserted `getdeviceInfo()` function collects device subscription event information, sends it to the server through the `sendRequest()` function. Based on received configurations, the server can extract runtime user configuration information, such as device id, numerical conditions, and room information. This sendRequest function does not need a response from server as runtime action enforcement.IoTSafe divides the existing static interaction graph into multiple sub-graphs. If a device can interact with devices in multiple rooms(*e.g.*, AC with multiple room tags), then we add them to each room's sub-graph. As shown in Figure 4, the solid lines indicate cyberspace interactions, and the dotted lines are potentially physical interactions. Each node represents a device or a system variable. The trigger condition is also added to each path, such as "Temp>85F". In the end, the static analysis module outputs multiple static interaction graphs, which will be used for dynamic physical interaction testing.

B. Real Physical Interaction Discovery

After generating the static interaction graph, IoTSafe generates testing cases for each group of devices. However, many unreal physical interactions are existing and we also need to identify implicit and joint physical interactions. Hence, we propose to explore a new dynamic testing approach to identify real physical interactions among IoT devices. Different from traditional dynamic testing methods, two following challenges need to be addressed in our dynamic testing for real physical interaction discovery.

- **Temporal Interaction Testing:** Different from traditional software testing, testing physical interactions between IoT devices takes time, especially for capturing temporal interactions. Some slowly-changing physical interactions may need a long time to be identified, such as temperature and humidity. For example, to observe the influence of temperature related physical actions, such as turning on/off a heater, we need to wait for more than 15 minutes before observing any temperature changes. Since there may exist a considerable amount of testing cases, testing all cases one by one will prevent the normal usage for a long time. Hence, testing an IoT platform requires minimum effects on normal usability. To this end, our dynamic testing approach needs to optimize the time cost by choosing the right testing action sequence and parallelizing independent physical testings.
- **Safe & Secure Testing:** Another distinction from software dynamic testing is that our dynamic testing is implemented on real devices and has an influence on real home environments. There are two kinds of restrictions on testing IoT device physical interactions: 1) Device restrictions: some devices, such as locks and sprinklers, may cause a dangerous situation if tested directly. The testing process shall not involve these sensitive devices. 2) Condition restrictions: the environment condition during testing should be within a safe range. If a certain physical condition exceeds the safe range, it may cause damages to the home property, *e.g.*, a high temperature or humidity environment in the storage room. Our system needs to arrange the testing process adaptively to avoid unsafe/insecure conditions.

Our physical interaction discovery module aims at automatically inferring real IoT device interactions through testing the influences of a device's actions on other devices' states. Our dynamic testing process includes two stages, *testing case generation* and *dynamic testing*. The testing case generation module provides all testable cases based on static analysis results. During the dynamic testing, IoTSafe automatically tests devices' actions in a parallel manner and collects the sensors' readings. After that, parameters of physical models are calculated based on these testing data.

1) Testing Case Generation for Grouped Devices: In general, a testing case is a combination of all device states in a group where these devices are in the same room and may interact via physical channels. The room information can be obtained from a device's room tag during the static analysis in Section IV-A. Suppose there are n devices in a testing group, we define $States(D_i)$ to represent all possible working modes for device i (denoted as D_i), such as heating, cooling, fan and off for a thermostat. Let $state(D_i)$ denote a working mode of D_i, where $state(D_i) \in States(D_i)$. Suppose there are n devices in a testing group. A testing case is a combination of a selected device D_i's possible working modes $\{D_1.off, \cdots, state(D_i), \cdots, D_n.off\}$, where all the other devices except D_i are in the off state. The generated testing cases only have one device in the working mode at the same time, which reduces the influence from other devices in the same physical channel. The objective of our dynamic testing is to reduce the overall testing time while covering all possible device actions.

To address the safe/secure testing challenge, we exclude sensitive devices during the testing case generation. Since we still need to capture interactions for restricted devices, which are excluded from testing, we adopt two workaround solutions. The first one is that we provide potential static interactions and let users verify their presence in practice. The second one is to use runtime data to verify static interactions during normal usage, which will be introduced in Section IV-C. To address the challenge of testing temporal interactions, we use the device's working mode instead of specific setting values to generate testing cases. For example, the $States(thermostat)$ has four working modes, including heating, cooling, vent, and off. These modes are extracted from the Samsung SmartThings capability document during the static analysis. Using these modes instead of the specific temperature values, we can reduce its testing cases to 4 cases.

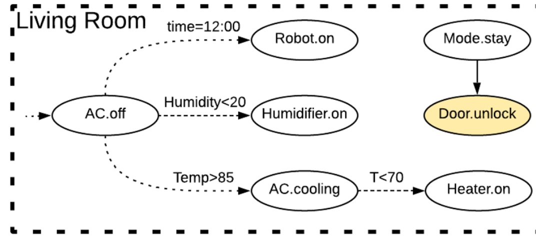


Fig. 4: An example of static interaction graph.

For devices connected to smart plugs, our system views these smart plugs as integral parts of their connected devices. For example, if a fan is connected to a smart plug, we consider this plug as a fan. Since most smart plugs have limited functions, such as turning on or off, IoTSafe views these devices with only two states (on and off).

Initially, all devices are in the off state. The testing case generation starts from randomly selecting a device and choosing a working mode of that device. If the device has more working modes other than on and off, we generate new testing cases by traversing all working modes of that device. Otherwise, in the next testing case, we turn off the previously selected device and turn on another device with a randomly chosen working mode. Note that the testing case generation excludes the sensitive device working modes for safety consideration. Revisit the example in Figure 4, where devices within the same room are included in the static interaction graph. Suppose users define a sensitive device state set S_{avoid} for any action related to locks, and thus the unlock action is excluded from the testing cases. In Figure 4, testing cases contain all following possible states to be tested: AC.off/cooling, humidifier.on/off, heater.on/off, and robot.on/off. One example testing case is (humidifier.off, AC.off, heater.off, robot.on).

2) Dynamic Testing: After generating all possible testing cases, we need to test them to identify the spatial, joint, and implicit effects of physical interactions among devices. To further reduce the testing overhead, IoTSafe optimizes the testing process in two aspects: i) sequential testing; and ii) parallel testing. For safety considerations, IoTSafe allows users to set a safe range for sensitive physical channels, *e.g.*, temperature and humidity. During the dynamic testing, our system monitors physical channel conditions and prevents from exceeding the safe ranges. In Appendix D, we demonstrate the effectiveness/necessity of the safety ranges for the dynamic testing.

To bootstrap the testing, IoTSafe first conducts calibrations for devices to capture normal fluctuations of sensors. We measure sensor readings' differences and time intervals between two adjacent sensor report intervals. Let $SensVal(D_i, state, t)$ denote the sensor reading directly related to device D_i with a specific working mode $state$ at time t. We measure $\Delta SensVal(D_i, state, t_1, t_2)$, which is the sensor reading difference for D_i's current working mode $state$ between two adjacent sensor report intervals at time t_2 and t_1. It represents the normal fluctuation of a sensor. We use it as a baseline to filter false interactions caused by environment noises. We also collect the sensor report interval information in the calibration phase. During the dynamic testing, if the difference of a device's sensor reading is greater than the baseline fluctuation, we identify a real physical interaction.

Sequential Testing. The main purpose of sequential testing is to test all non-parallelizable testing cases and identify devices' spatial, temporal, joint, and implicit physical interactions efficiently. The non-parallelizable cases indicate these cases that may potentially affect the same physical channels. The optimization of sequential testing focuses on reducing testing overhead by choosing the proper testing sequence, which is started by matching devices' trigger conditions from apps. Another optimization is to reduce the conflicting influence between adjacent testing cases.

We first define the testing overhead T_{cost} as the time needed for the whole testing process, which is the sum of testing each case's time cost $\Sigma(t_{case_i})$, where $case_i$ represents a testing case with only one device D_i in the working mode, and t_{case_i} represents the time needed for testing $case_i$. However, due to the continuous effect, the order of testing cases makes a difference on the effectiveness of the testing process. Our objective is to optimize the order of testing cases to minimize the testing overhead T_{cost}. Given a testing case with one specific device D_i in the working mode, t_{case_i} is mainly affected by D_i's working time, denoted by t_{work}, which is related to the associated sensor's report interval.

The testing time of $case_i$ is calculated as the following:

$$t_{case_i} = \begin{cases} t_{work}, & D_i \notin D_{temp}. \\ t_{work} + t_{off}, & D_i \in D_{temp}. \end{cases} \quad (1)$$

In each testing process, we record D_i's working time as t_{work} and the idle time after this device being turned off t_{off}. D_{temp} is the set of devices related to temporal channels, such as temperature, humidity, and air quality. For devices related to non-temporal channels, t_{work} can be relatively short. However, to ensure observing the physical effect from sensors, our system makes t_{work} equal to one sensor report interval. In IoTSafe, t_{off} is chosen to be the same as the related sensor's report interval. Based on our observation, one sensor interval (usually 15 minutes) is enough to identify the continuous effect. Hence, our system records at least two sensor reports for a temporal channel related testing case.

The ordering of testing cases in the sequential testing is based on a comparison between current physical conditions with other untested cases' device trigger conditions, which helps the system to find the most suitable case to continue. For example, if the room temperature is already lower than 60F, then testing the heater makes more sense than testing the AC cooling. IoTSafe uses a greedy algorithm to decide the best suitable cases. The trigger conditions of devices' working states are defined in user apps. The comparison here only considers the trigger conditions that are related to a particular physical channel. For example, if a heater's trigger condition is "when time is 18:00", then our system does not consider this condition. If one device state is related to multiple trigger conditions and then the system compares all of them. If there are multiple cases matching the current home environment, our

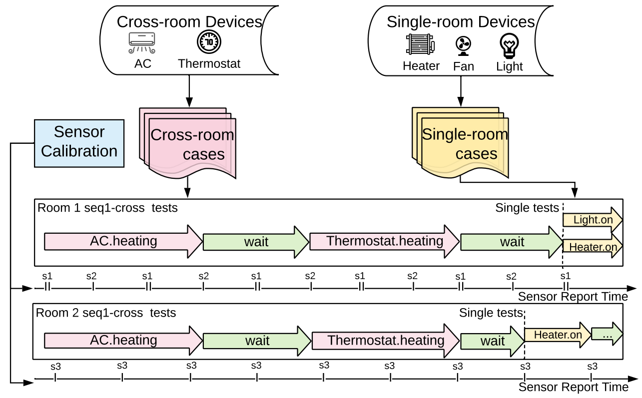


Fig. 5: Parallel testing for single-room and cross-room devices.

tool chooses the case with the closest trigger condition based on the greedy algorithm. If all cases are not satisfied or have no trigger condition, then our system just randomly chooses one to continue the testing.

After the testing starts, IoTSafe chooses the next testing case. Here the main issue is that we need to consider the conflicting influences between device states. For example, if we use the AC heating function right after the AC cooling function, the heating mode may need an extended time for the AC to warm up. Some conflicting working states may even cancel out others' physical influence (*e.g.*, heating and cooling).

Single-room Parallel Testing. To further reduce the testing overhead, IoTSafe also parallelizes single-room testing cases with potentially independent physical channels. As shown in Figure 5, we have three multi-purpose sensors, $sensor_1$, $sensor_2$, and $sensor_3$, which can monitor both temperature and illuminance. The bottom line of Figure 5 shows the sensor report timeline of each sensor, where two individual reports from sensors represent a combined report interval. Each case's testing time needs to cover the combined report interval of all related sensors, which guarantees that the working time covers all sensors' intervals. The interaction channel for the light bulb is illuminance and the heater is the temperature, respectively. Hence, testing cases of lights and heaters can run simultaneously in $Room_1$ (shown as yellow arrows in Figure 5), which reduces the time overhead of the dynamic testing process. The green arrow represents the idle time T_{off} in Equation 1 for the testing case related to temporal channels. IoTSafe first identifies cases that can be parallelized in each room. Then, for testing cases related to an individual independent channel, it runs sequential testing.

One challenge here is that parallel testing may miss device interactions with potential implicit effects. For example, if a heater and a humidifier are tested at the same time, then the heater's humidity influence will be suppressed by the humidifier. To avoid such issues, potentially correlated physical channels (*e.g.*, humidity and temperature channels) are predefined in IoTSafe. We combine testing cases with correlated physical channels into a sequential testing chain instead of parallel testing.

Cross-room Parallel Testing. The cross-room parallel testing is used for reducing testing overhead by parallelizing different rooms' sequential testing. We start from testing cases with any cross-room device (*i.e.*, a device with a multiple-

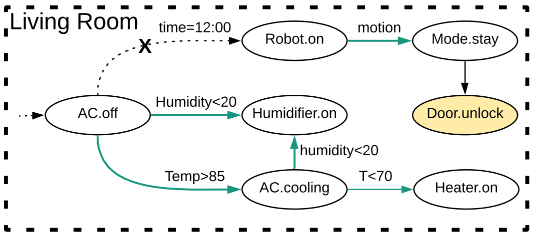


Fig. 6: Dynamic testing results.

room tag) and compare their user triggering settings with current conditions. If any cross-room testing case fits the current condition, we test this case at first. If multiple cases satisfy the current condition, we use a greedy algorithm to choose the most suitable case, which is the same process as the sequential testing. For example, if an AC should be heating at 70F, and cooling at 80F. When the current temperature is 73F, our system chooses the heating testing case instead of the one with cooling action.

The cross-room devices' testings are shown as pink arrows in Figure 5. For cross-room devices, the challenge here is that sensors in different rooms usually report events asynchronously. Hence, we need a synchronization scheme to guarantee the event collection from multiple sensors. We introduce a waiting time to guarantee the coverage for the longest sensor interval. We solve the synchronization problem by choosing a proper starting time, ending time, which leads to an extra waiting time (shown as green arrows in Figure 5). Taking Figure 5 as an example, $Room_1$ has two sensors, and $Room_2$ has one sensor, where $sensor_1$ and $sensor_2$ have longer sensor report intervals than $sensor_3$. IoTSafe first obtains sensors' reporting times (*e.g.*, s_1, s_2, and s_3 in Figure 5) in the sensor calibration process. During the runtime testing, each case needs to last at least one sensor report interval (*i.e.*, getting two adjacent reports) in each room, which is based on the longest report interval of multiple sensors, *i.e.*, $sensor_1$'s interval plus $sensor_2$'s interval. Hence, the pink arrow for the "AC.heating" covers one interval for both $sensor_1$ and $sensor_2$, and it also covers multiple intervals for $sensor_3$ in $Room_2$. For cases interacted by temporal channels, there are also green arrows representing the waiting time T_{off} after the device being turned off. After one cross-room testing is over, the system waits for T_{off} and then starts the next testing case. The system records the difference between the two reports to build the physical influence model of this device. After finishing the cross-room testing cases, the system starts single-room testing cases.

3) Real Interaction Graph: After the testing process, we generate real physical interaction paths by removing unreal paths in the static interaction graph and adding additional implicit interactions. As shown in Figure 6, the solid green arrows indicate real physical interactions while the dotted arrows represent unreal interactions in a smart home environment.

C. Runtime Prediction

Because of the continuous (slowly-changing) effects of temporal channels, a device can still violate safety policies even after the enforcement of a control policy. The low reporting rate of commodity IoT sensors makes such situations even worse, *e.g.*, Zooz 4-in-1 Sensor [5] reports temperature every 15 minutes by default. Therefore, IoTSafe needs an early detection for unsafe physical conditions to ensure the policy enforcement in a timely manner. Our prediction mainly concerns the physical channels of temperature, humidity, smoke, and water level. IoTSafe's prediction module includes two parts: i) the offline physical channel modeling, and ii) the runtime policy violation prediction for proactively alerting dangerous physical conditions.

1) Physical Modeling: We describe our physical models for air-related physical channels, *e.g.*, temperature, humidity, smoke. We first present a general model for these physical channels, and then give detailed parameters for specific channels.

The general model of air-related physical channels [13], [19], [32], [41], [40] can be described as follows:

$$\Delta SensVal(D_i, state, t_1, t_2) = \frac{\Delta Q(D_i, state, t_1, t_2)}{\beta}, \quad (2)$$

where $\Delta SensVal(D_i, state, t_1, t_2)$ denotes the changed sensor value from time t_1 to t_2 for D_i with working mode $state$. It can be calculated from periodical sensor reports. $\Delta Q(D_i, state, t_1, t_2)$ represents the amount of channel quantity change for D_i with working mode $state$ from time t_1 to t_2. For example, for temperature-related devices, it reflects the total amount of changed surrounding heat from t_1 to t_2. For the humidity channel, it means the change of water vapor amount. β is the product of multiple air-related parameters, such as density, room volume, heat or water capacity, etc. Taking the humidity channel as an example, the model needs to calculate the change of water vapor quantity in the air. We model the humidity difference by calculating the water vapor provided by a related device (*e.g.*, humidifier) and calculating β by the air's specific humidity capacity (water vapor pressure) and temperature. Given $\Delta Q(D_i, state, t_1, t_2)$ and β, we can calculate $\Delta SensVal(D_i, state, t_1, t_2)$, which predicts the sensor value at t_2.

We show details of the calculation for temperature related devices. Here, $\Delta Q(D_i, state, t_1, t_2)$ represents the heat gain/loss caused by the operation of D_i during the period between t_1 and t_2. The specific room temperature prediction model for a device (*e.g.*, heater) can be represented as follows:

$$\Delta Q(D_i, state, t_1, t_2) = P_i \cdot (t_2 - t_1), \quad (3)$$

where P_i can be calculated by the influence of a temperature related device D_i, or known by its default power. We can get it from a device's power reading or the connected smart plug's power meter. For the temperature channel, we identify that the room temperature is fairly even. While the humidity channel has a distance factor for specific humidity device distances, which is detailed in Appendix D. We use data collected during the dynamic testing process to generate the offline physical model for each device and estimate the distance factor between devices and sensors.

2) Offline Model Initialization: By collecting all sensors' readings and timestamps during the dynamic testing, we initialize the parameter of each model, which is based on the device working states and related sensors' readings. The data is collected and stored in a cloud server for modeling process, where the server stores all devices' state reports and sensors' readings. For example, an event "turn on the heater at 10am, temperature has been changed from 70F to 73F in

20 minutes" can be represented by a string of "Heater (on, 10:00), TemperatureSensor (70F, 10:00), (73F, 10:20)".

$$min \sum_t (SensVal(\widehat{D_i}, state, t) - SensVal(D_i, state, t))^2 \tag{4}$$

For each model, we calculate its parameters based on Equation (4). $SensVal(D_i, state, t)$ is the measured sensor reading related to device D_i with a working mode $state$ at time t, while $SensVal(\widehat{D_i}, state, t)$ the predicted data based on the physical model. We minimize the mean squared error (MSE) for each device's model and obtain the initial parameters of each model. In Appendix E, we present more details about the physical channel modeling. During the smart home usage, we need to update the model due to adding new devices or changes of the environment context. As new sensor reports are received, we also periodically re-calculate model parameters.

3) Initialization for New Device or Restricted Device: For new devices, our system generates testing cases for a single device to identify the new device's influence on others. We initialize its model and then update the model during normal usage. For restricted devices, we build potential interactions among all related devices within the room. Then we initialize and update its interaction graph and model based on the data from daily usage. User settings refer to the specific trigger conditions and triggered device actions in a smart home app. Users may change the triggered device, the specific action, and the location of the device. When a user updates the trigger condition in an app, the system collects new condition settings and updates the interaction graph. In this case, we perform the new device initialization process.

D. Safety and Security Policy Specification

To properly enforce safety policies, we define our policy language to facilitate the policy enforcement in a smart home environment. These policies are defined based on typical IoT policies with constraints on physical conditions. The policy enforcement introduced in IoTSafe utilizes physical prediction models and considers additional enforcement conditions/actions when a possible violation is detected.

1) Policy Category: As mentioned in Section II-A, current smart home safety properties or policies [11] do not consider the physical influence of devices. To achieve fine-grained control on temporal physical interactions, we present a policy language for both temporal channel-specific interactions and non-temporal interactions. We define the following two types of policies for different interactions.

Control Policies for Temporal Interactions. We define this type of policy to address potential violations that involve temporal interactions. These policies are developed based on use cases of one or multiple temporal channel related devices' physical models to predict dangerous situations, *e.g.*, high temperature, humidity, air quality, or water level. Since a policy contains two parts, the trigger condition and the enforced action, we first let users define conditions that they want to avoid or maintain as triggers. Then, the system identifies conditions with temporal effects and enforces policies when potential violations are predicted based on physical models.

Control Policies for Instant Interactions. These policies are developed based on the use cases of one or multiple devices with non-temporal physical interactions, *e.g.*, motion and illuminance. We define this type of policy to address potential violations related to devices' instant physical influence. This policy is simply combined with one or multiple device states as trigger conditions and changing other devices' status for actions. Such a policy also defines specific behavior of sensitive platform variables, *e.g.*, time, location, and home mode. We first let users define a device status as a trigger condition (*e.g.*, motion.detected), and then define a targeted device status for an action (*e.g.*, light.on).

2) Policy Definition: We describe the detailed definition of our policy language, which supports both temporal and non-temporal (instant) interaction controls. We illustrate the syntax of our policy language in Figure 7. Users can refine existing policy templates or define new policies using this policy language. Expression (E) could be formed over device id (d), device state (s), and value (v) . The expression is a list of device states. Multiple expressions can also be composed together (E∘E), which are used by policies to combine multiple actions together. The violation detector collects expression states and uses them as a predicate. The enforced action contains device ID (d), device actions (a), or policy actions. The device ID is the targeted device id's string. The device action is a command needed for the device to implement (*e.g.*, on/off). The policy actions contain approving/denying apps' requests, and implementing an action (*e.g.*, heater.off during high-temperature alert, or sending a warning to users).

Expression
E ::= | d | s | v
Predicate
P ::= $E \circ E \mid P|P \mid P\&P \mid \neg P$
Action
A ::= implement|approve|deny
Policy
C ::= A | if P: C else : C | $(C|C)$

Fig. 7: The syntax for expressing safety and security policies.

For example, in Figure 1(b), the policy is "Turning off the heater and send a warning when the temperature is above 85F". IoTSafe uses the temperature physical model to obtain the predicted temperature value. If the predicted temperature is above 85F, the policy enforcement module will trigger this policy and the heat.off action, even the sensor has not reported the violated reading. According to the syntax, the trigger condition expression contains three elements, the device ID is "sensor1", the state is "value", and the trigger condition value is "85". The conditions are "device_id == sensor1" and "sensor1.value > 85". It contains two "implement()" actions, which are the "heater.off" and "push (temperature warning)". If no violation is predicted, the system does not enforce any actions, but only skips the policy and records this temperature report.

```
if (device_id==``sensor1'')&&(sensor1.value>85):
    implement(heater.off);
    implement(push(temperature warning));
```

Fig. 8: An example policy "Turn off the heater and send a warning when the temperature is above 85F".

E. Policy Enforcement.

The dynamic policy enforcement is implemented in two aspects. The first one is wildly adopted by existing works [25], [11]. The system instruments an "if" condition and an "http request" before all commands in an app. If the server approves such action under the current context, the server will send back an "approve" message to the app. Otherwise, this action will be denied.

The second enforcement scheme targets policy enforcement with prediction. IoTSafe uses a monitor app to send all devices' events to the server and waits for responses. It uses an "http request" to ask for enforcement actions periodically (*e.g.*, every minute), which ensures timely enforcement for any predicted violation. If a violation is predicted, the enforcement module sends related policies' commands to the monitor app. The data collector first receives runtime data from the monitor app, which includes all events from devices. Based on reported data, if a violation is predicted, our system considers it as an expected risky situation. Hence it sends a warning to users and triggers corresponding actions.

Our policy enforcement module has additional enhancements for policies. The first one is trigger condition enhancement. It searches for all related devices from the interaction graphs, which can trigger this policy. Then it asks users for more fine-grained trigger condition definition to avoid unexpected triggering, *e.g.*, the robot tr use case 2 in Section V-D. The second policy enforcement targets similar actions from other devices. If the system detects multiple devices that can interact with the same physical channel, such as heaters and AC.heating, the system asks users to enforce policies on other devices' actions. In the case of Figure 1(b), if multiple heating devices are working, the system will not only implement the "heater.off", but also ask users whether it needs to turn off other heating devices.

V. Implementation and Evaluation

We have implemented a proof-of-concept prototype system of IoTSafe based on the Samsung SmartThings platform [21]. We studied a total of 45 official SmartThings applications [17] with respect to cyberspace and physical interactions to demonstrate the effectiveness of IoTSafe. We selected 21 representative IoT apps (based on their functions and user studies [16], [28]) and deployed them in our smart home environment with various configurations. We set up the policy enforcement server based on a Web server running on the Google cloud platform [24]. It is a lightweight server and can be deployed on small devices, such as a Raspberry Pi. We chose Google cloud because it provides a public IP address that our app can communicate with. All apps send events related information, such as sensor reports and action requests, to the policy enforcement server.

The implementation of IoTSafe includes three main parts: i) the static analysis tool; ii) the policy enforcement server, and iii) three special SmartThings apps. The static analysis tool instruments apps and extracts static interactions. The server provides functions such as collecting and storing devices' states, policy storage/consultation, and physical condition prediction. The special SmartThings apps have three different purposes/functionalities: as the data collector, runtime monitor, and policy manager. The collector provides the device deployment information and user settings for dynamic testing. The monitor subscribes to all devices' events and sends them to the server during dynamic testing and normal usage. It also takes corresponding actions when the server sends commands. The policy manager helps users setting his/her policies and gives hints about potentially related devices.

Users are involved only in two steps: i) choosing testing/restricted devices, and ii) setting user policies. The dynamic testing process and model training process are fully automatic and users only need to choose the avoided devices. To simplify the policy specification process for users, we provide a set of 36 pre-defined policy templates for users in Table IX. For the policy specification module, our tool provides three options: (1) users can directly enable some general safety policies (such as policies #1 to #6 in Table VIII), which have been pre-defined in IoTSafe; (2) users can customize all 36 reference policies, listed in Table IX, by changing corresponding device actions, which only requires operations through the user-friendly UI of our tool; (3) users can also define new policies. After the safety policies are defined, the prediction and policy enforcement modules work in an automatic manner.

A. Smart Home Testbed Setting

Our system focuses on the real smart home/indoor environment, which typically has an average of 11 devices, according to a Deloitte US report [1]. In order to make our testbed more realistic, we deployed 23 different types of IoT devices and a total number of 33 devices in 7 testing groups, which include real deployments in single room and three-room apartments. Table IX in Appendix C shows the detailed apps and configurations of these 7 testing groups. The three-room apartment includes a living room, a bedroom, and a bathroom, which simulates a multi-app smart home environment as shown in Figure 9 (for Group 4 in Table IX). In addition, we extracted user routines from real-world IoT user datasets [28] [16], and built corresponding IoT apps in three groups (*i.e.*, Groups 5-7). We tested these three groups based on current home layouts to demonstrate the effectiveness of IoTSafe for practical smart home settings.

Some common types of IoT devices were deployed in multiple places in our experiments. For example, window switches (#18) in Figure 9 were deployed in both the living room and the bedroom with different device IDs. Since these IoT devices were deployed in three different rooms, their influence areas were constrained by the home structure and deployment locations. We deployed 21 official SmartThings apps (ST1-ST21 listed in Table IX) in our experimental environment (where the detailed settings can be found in Appendix B). We defined 36 safety and security policies for physical interaction control with considering general IoT safety and security requirements.

For each testing group, we setup the baseline by manually examining physical interactions between devices. To this end, each device worked separately for a period (two sensor report intervals) to identify possible temporal device interactions and then waited for another sensor interval to examine its continuous effect. The longest sensor report interval we chose is 15 minutes. Based on our experimental results, this interval is long enough to identify devices' temporal influences.

For the policy enforcement, we enabled 14 general safety and security policies from Table VIII in Group 4 and Group 7. The monitor updates devices' states to the server per minute and takes actions based on the server's response. IoTSafe enforces proper policies based on devices' states and the predicted conditions. We conducted two case studies based on the settings of Group 4 and Group 7 in Table IX.

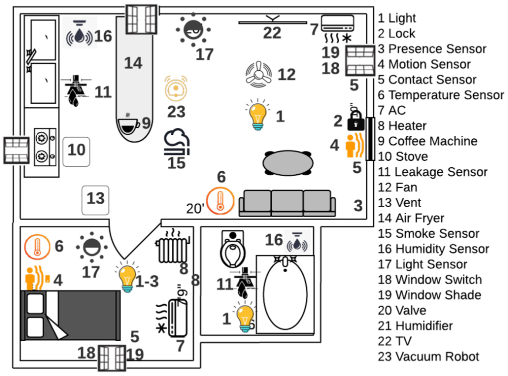


Fig. 9: One of our smart home testbeds for Group 4 and 7 where 33 devices were deployed in three different rooms.

B. Real Physical Interaction Identification

The objective of this experiment is to identify real physical interactions among devices. We compared our results against IoTMon [18], which derives potential physical interactions based on static analysis. IoTMon discovers all potential physical interactions by stitching apps if they share common physical channels. The *real* physical interactions are identified by IoTSafe through both static analysis and dynamic testing.

TABLE I: Physical interaction discovery.

Group ID	Potential Interact. [18]	Real Interact.	Joint Interact.	Implicit Interact.	Temporal Interact.	FP of Static Analysis	FN of Static Analysis
1	17	12	12	6	12	5	6
2	13	5	1	1	0	8	1
3	41	19	18	9	16	22	9
4	130	39	38	17	26	91	17
5	32	14	8	4	4	18	2
6	46	17	6	8	13	29	8
7	42	22	11	6	18	20	6

Table I lists the results of physical interaction discovery. It also reports the False Negative (FN) and False Positive (FP) results of the static analysis approach [18]. The implicit interactions are actually false negatives of static analysis, which are interactions identified by dynamic testings but ignored by the static analysis. With an increasing number of apps in a smart home environment, the ratio of false positives to the total interactions is increasing. This is because the potential physical interactions do not consider physical environment constraints and thus introduce many false static physical interactions, which are potential interactions identified by the static analysis but not detected by the dynamic testing. These false positive results are caused by limitations of the static analysis, such as wrong interaction analysis, missing room layouts and device distance.

As shown in Table I, we list potential interactions derived from the static analysis, real interactions identified by IoTSafe, and detailed categories of these physical interactions. Group 1 and Group 2 are both single room environments (the detailed apps and device information are listed in Table IX). Group 3 is a two-room environment, which contains a living room and a bedroom, and our dynamic testing shows that there are more than 50% false positive physical interactions derived by static analysis due to the increasing number of devices. Group 4 has the same structure as Group 3, but it is deployed with more devices and apps. It involves 21 apps, containing 130 potential physical interactions. However, we only found 39 real physical interactions, which contain 17 implicit interactions (*e.g.*, the opening window may change the ultraviolet level). These implicit interactions are interactions from devices with multiple-channel influences, like AC and stove. Group 5 has the same home layout as Group 2, which involves 14 apps, containing 32 potential physical interactions. We found 14 real physical interactions, out of which 4 implicit interactions were identified. In Groups 6 and 7, where we added more apps and devices than that of Group 3, the number of real interactions is increased to 17 and 22, respectively. The number of implicit interactions is 8 and 6 respectively, which are mainly caused by devices such as windows, AC, and stoves.

TABLE II: Interaction discovery broken down in Group 4.

Physical Channels	Potential Interactions [18]	Dynamic Testing	Online Updating	Implicit Interactions
Temperature	56	10	2	5
Humidity	20	3	2	6
Smoke	4	1	1	1
Motion	6	3	2	2
Illuminance	42	4	1	1
Water	2	1	0	0
Ultraviolet	0	0	2	2

In Table II, we show the detailed physical interactions identified in Group 4, which has the most complex device interaction scenario. The temperature and illuminance channels have nearly 50% unreal interactions derived by the static analysis. Overall, we identified 17 implicit interactions, *e.g.*, window shades may have an influence on the temperature. For the temperature channel, there were 56 potential interactions among devices. During the dynamic testing, we identified 10 real interactions. For the humidity channel, we identified 6 implicit interactions, mostly related to devices that could change the temperature. These devices change humidity as well as the temperature at the same time. We also found humidity is highly sensitive to the spatial distance between sensors and devices (*e.g.*, humidifier), which is shown in Appendix E.

For the false positive interactions in Table II, these are interactions identified by the static analysis but not in dynamic testings. Taking the smoke channel as an example, the static analysis gives 4 potential interactions but only one is real, which leads to 3 false positives. Based on the similarity between keywords, the static analysis approach gives potential interactions between the heater and the smoke channels. However, since the heater could not cause the smoke problem in the dynamic testing, we consider this interaction as a false positive.

TABLE III: Summary of implicit and unreal interactions of Group 4. ✓represents real interactions identified by IoTSafe, ◯ represents implicit interactions identified by IoTSafe, and ✗ represents unreal interactions derived by the static analysis [18].

Devices	Temperature	Humidity	Smoke	Motion	Illuminance	Ultraviolet	Water
AC	✓	◯					
Heater	✓	◯	✗				
Vent	✓	✓					
Fan	✓			✗			
Window	✓	✓		◯	✗	◯	
Radiator	✓	◯	✗				
Humidifier		✓					
Coffee Machine		◯					
Robot				◯			
Stove	◯	◯	✗				
PC	◯						
TV					◯		
Air Fryer	◯		✓				
Light					✓		
Shade	◯				✓	◯	
Valve							✓

We further summarize the related devices with implicit and unreal interactions in Table III. There exist devices having implicit effects that cannot be captured by the static analysis, *e.g.*, coffee machine to the humidity sensors, stove to the temperature sensor, and TV to the illuminance sensor. Most devices that change the temperature will also change the humidity. This is because temperature and humidity are physically related with each other and can be changed at the same time.

C. Runtime Prediction

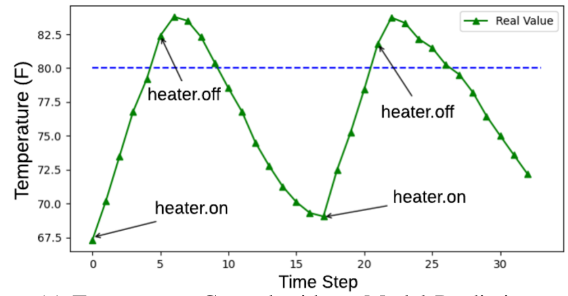
We measured the effectiveness of our modeling method for temporal channels and we mainly focused on the temperature-related devices. We list our experimental results in Table IV. The average error is the average difference between predicted temperature values and the sensor's actual readings at runtime. We measured 5 times of temperature sensor readings and the predicted values for each group. The room temperature was 72F on average, which was used as a baseline to calculate the error percentage. The sensor report interval means the report intervals of used sensors. For example, two types of temperature/humidity sensors have 8 or 15 minutes of report intervals. Group 1 has only one room and all devices are temperature related. It has the smallest error (1.0F) because of its one-room layout and the small room size. Group 2 has no temperature related device. Since Groups 3 and 4 have more rooms and more environmental interference, their errors are higher than that of Group 1. Group 5 has the same layout as Group 2, but with more temperature-related devices, and Group 6 and 7 have the same layouts as Group 3. Hence, their model prediction results are similar. In general, such an average error is small and IoTSafe would not miss the violation prediction of risky situations.

We set 80F as the temperature threshold for policy violation detection. IoTSafe achieves 93% accuracy for temperature related violation prediction in Group 1 (15 out of 16 risky situations in total). The only missing case was because the window was kept open while the heater was working, which makes their joint effects harder to predict than common scenarios.

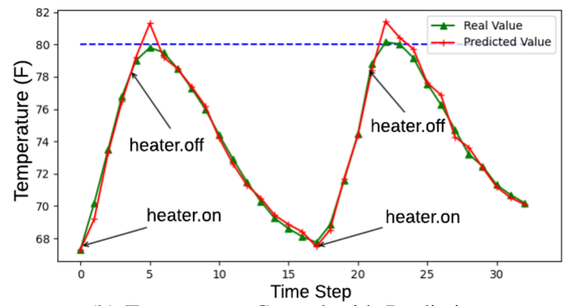
TABLE IV: Effectiveness of the temperature channel modeling.

Device Group	Sensor Time Intervals (min)	Average Error	Error Percentage
1	15	1.0F	1.4%
2	15	N/A	N/A
3	8,15	1.4F	1.9%
4	8,15	1.9F	2.6%
5	8,15	2.2F	3.1%
6	8,15	1.7F	2.4%
7	8,15	2.3F	3.2%

Taking Group 4 as an example, the system contains 170 sensor reports from Group 4. We set the temperature above 80F as a risky situation and there were 55 risky reports. IoTSafe detected 49 out of 55 risky situations before they could happen, and missed 7 of them. The false negative was because the ground-truth temperature was just on the boarder-line of above 80F, while the predicted result was slightly lower than 80F. If we introduced a 3-minute delayed time window, IoTSafe could give 4 more violation warnings and predict 53 risky situations. The system also gives 8 false negatives and 107 real negative results, which were mainly because of the borderline issue (5 times) and the environmental interference (3 times). In summary, the accuracy is about 96% for the true positive rate and 7% for the false negative rate. Figure 10(b) shows an example of the predicted temperature values compared to the real sensor readings.



(a) Temperature Control without Model Prediction.



(b) Temperature Control with Prediction.

Fig. 10: Prediction results

TABLE V: Physical model accuracy based on prediction time.

Physical Channel	Prediction Lead Time (minute)	Average Error	Error Percentage
Temperature	45	3.3F	4.6%
Temperature	30	1.5F	2.1%
Temperature	15	1.0F	1.4%
Humidity	45	7.1%	11.8%
Humidity	30	3.4%	5.7%
Humidity	15	1.3%	2.2%

We also evaluated the lead time on the prediction accuracy in Table V, where the lead time is defined as the advanced time that prediction is conducted before the actual sensor readings report a violation. We evaluated the temperature and humidity channels based on the prediction lead time with 15, 30, and 45 minutes. We tested the model's accuracy based on related devices in Group 4 and compared the prediction accuracy with different lead time. The error increases with an increase of the prediction lead time, especially for the humidity channel. Comparing with the temperature, humidity is very sensitive to device states and changes rapidly if some device states or environment conditions (*e.g.*, air pressure) changed. The humidity changed more rapidly than the temperature, making the value error percentage higher than that of the temperature. However, the average error is still small and it should not affect the policy enforcement.

D. Case Studies

We conducted two case studies to demonstrate the effectiveness of the policy enforcement in IoTSafe against indirect attacks that exploit temporal physical interactions.

Use Case I: We first examined the effectiveness of IoTSafe in enforcing safety policies at runtime by running a malicious app based on the smart home setting of Group 4 in Table IX. In Figure 11, an attacker exploited two compromised IoT devices to open the window. The attacker first manipulated the vacuum robot to trigger the "STAY" home mode (*i.e.*, someone is at home) in order to bypass the security policy "Do not open

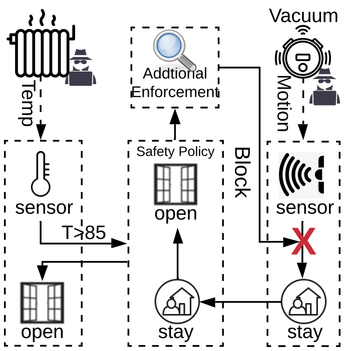


Fig. 11: Attack scenario in use case I.

window when no one is at home". Then he/she manipulated a heater to raise the temperature to trigger a window opening action. IoTSafe first detected the interaction between the robot and the motion sensor. Then, our system gives hints to the user to set additional trigger conditions. For example, if the user sets an additional condition as "Do not trigger the family mode when the robot is working", the home mode app will be blocked by the policy. As shown in the red cross in Figure 11, our system can block malicious manipulation of the home mode.

Use Case II: To demonstrate the effectiveness of physical modeling based prediction of risky conditions, we conducted the second case study to compare results from two enforcement schemes based on the smart home setting of Group 7 in Table IX. We introduced a malicious heater control app that keeps heaters on all the time. We first tested a sequence of heater/AC.heating cases without the prediction based on physical models. Assume the user set the policies "The temperature should be below 80F. Otherwise, turn off all heating devices." and "Open the window when thetemperature is above 84F." As shown in Figure 10(a), the policy enforcement without prediction was delayed around 11 minutes and the temperature reached 82.4F. Even after turning off the heating devices, the temperature still reached 84F and the high temperature could remain 5 report intervals (75 minutes). In Figure 10(b), we tested the same sequence with the prediction based on the physical model. IoTSafe turned off the heater 18 minutes ahead and the home environment was kept from the high temperature situation.

E. System Performance

Dynamic Testing Performance. We study IoTSafe's runtime overhead of dynamic testings. We compared our parallel testing design against a non-parallel testing method in terms of the time consumption considering four groups of apps. The non-parallel testing does not do parallel testing, which means only one device can be tested at a time in the whole home. It also does not have parallel testing among different channels' testing cases during the single-room testing. Our experimental results are shown in Table VI.

The sensor report intervals (by default in SmartThings) are 5 minutes for light-related sensors, 8 or 15 minutes for temperature, water and humidity sensors, and other sensors (smoke, motion) are instant to report any changes immediately. For Group 1, it has 17 potential interactions and costs 150 minutes to test 5 temperature related devices. Most of the time was consumed by the 15-minute report interval from temperature sensors. For the non-parallel testing, it took 150 minutes because this is a single room environment and only one temperature channel was tested. Group 2 has four illuminance related apps, and there are 13 potential interaction paths. The testing process took 40 minutes on average (for 5 real interactions). Group 3 has five temperature related apps and two illuminance related apps, which resulted in 41 interactions. The time consumption for the whole testing process was 90 minutes, while the non-parallel testing took 135 minutes. This is a multiple-room and multiple-channel environment, IoTSafe can save time with the parallel testing. The time consumption for testing Group 4 using our approach was 120 minutes, which was reduced by almost 1.5 hours. In Group 5, it costed 110 minutes for our method. It saved 55 minutes because different physical channels' testing cases in the single room could be tested simultaneously. Group 6 and Group 7 have similar results as Group 4. Since they have similar apps and devices, IoTSafe took a similar time to complete the testing than that of Group 4. The time consumption of them is less than that of Group 5 because IoTSafe implemented the parallel testing for cases in different rooms and different channels. Hence, our method can reduce the time consumption for 1-2 hours on average.

TABLE VI: Time consumption of the dynamic testing.

Group ID	None-Parallel (minute)	Our Method (minute)
1	150	150
2	40	40
3	135	90
4	215	120
5	165	110
6	180	105
7	190	135

New Device Initialization Overhead. We evaluated our physical model initialization for newly added devices. The device initialization is for devices that are newly added or moved after the original model initialization or have testing restrictions. We did not run the entire dynamic testing process for a newly added device, but only trained its model based on daily usage data. Figure 12 shows the experimental results of adding new devices into an existing home environment. The X-axis shows the number of newly added devices in the smart home environment, and the Y-axis shows necessary sensor reports for initializing their models. The error bar represents the maximum/minimum reports needed in each set of testing.

To initialize one single device's model, IoTSafe needs to wait until this device being used one-time in the environment. Hence, the initial time overhead is just one sensor report interval, which is the same as the dynamic testing. With the increasing number of devices, the overhead increases exponentially because of the interference from each other increases. Hence, IoTSafe needs more data to initialize devices' models. For a normal room with four associated devices, it usually needs 2-3 sensor reports (around 1 hour) based on the runtime states of all related devices. The uncertainty of the report amount was because, during normal usage, these devices might work at the same time, which makes it harder to know the influence of the new devices.

Server Runtime Overhead. For evaluating the overhead of the runtime policy enforcement in IoTSafe, we tested the average response time of one policy enforcement in each group. The response time is calculated as the time from requesting an action to the enforcement of the action. We ran such tests for 20 rounds in each group, and the time consumption for

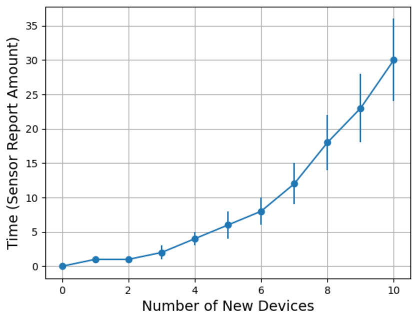


Fig. 12: Time consumption of new device initialization.

Groups 1-3 is 230ms on average, which will not affect timely enforcement.

VI. Related Work

Static Security Analysis. Many research efforts have been undertaken by applying static analysis techniques in identifying and improving IoT security and safety. Since it does not require the dynamic execution of a program, it scales better when performing a large-scale study. Fernandes *et al.* [20] identified several security-critical design flaws (*e.g.*, event leakage and event spoofing) in the SmartThings platform. They discovered that over 55% of SmartApps in the store are over-privileged (as of 2016). In addition, the SmartThings platform does not sufficiently protect event data that carry sensitive information, resulting in event leakage and event spoofing vulnerabilities. SAINT [9] tracks information flows from sensitive sources to external sinks to find sensitive data flows. SOTERIA [10] and IotSan [29] apply model checking to verify user-defined safety, security, and functional properties. Different from SOTERIA, IotSan focuses on revealing flaws in the interactions between sensors, apps, and actuators, *e.g.*, verifying conflicting and repeated commands from multiple apps that subscribe the same sensor. Neither SOTERIA nor IotSan considers the *physical* interactions in their analysis. IoTMon [18] discovers potential physical interaction chains across IoT applications and assesses the security/safety risk of each discovered cross-app interaction. However, IoTMon is unable to capture runtime policy violations in real-world IoT deployments.

Runtime Policy Enforcement. Access control is a fundamental and challenging problem in IoT. In many instances, static analysis of IoT apps provides useful information to guide the runtime enforcement. SmartAuth [42] ensures the app's runtime behavior is consistent with its static analysis model learned from the code description. SmartAuth mitigates the security risks of over-privileged IoT apps. To address the potential permission abuse and data leakage issues, FlowFence [22] enforces developer-specified information flow policies, while blocking all other undeclared flows. HoMonit [45] defines a normal traffic behavior model by analyzing an app's source code and user interface. It detects malicious behaviors of apps at runtime.

IoTGuard [11] supports the detection of policy violations in multi-app environments by means of reachability analysis. However, their reachability analysis is based on a static merging of trigger-action rules extracted from individual apps. The lack of physical reachability analysis leads to substantial false positives for app interactions through physical channels, resulting in over-alarmed and unnecessary action rejections (which may also hurt the usability). In this work, we fill this gap and specifically focus on dynamic security and safety assessment with respect to real physical interactions in multi-app IoT environments.

In Table VII, we compare our work with recent studies in multiple perspectives, *e.g.*, scope and method. Helion [28] provides a method to extract normal routines from users' daily data. It could predict normal behavior and give alarms when an abnormal event happens. Cobb *et al.* [16] provided a comprehensive overview about risky IFTTT applets for real users. Peeves [7] proposed a method to build normal event sequences, which can be used to detect malicious behaviors. These studies provide a measurement for risky behavior detection from real users' perspective. However, they do not consider physical interactions among devices and do not provide a runtime policy enforcement mechanism.

TABLE VII: The comparison of IoTSafe with other IoT systems.

System	Inter-app Analysis	Real Physical Interaction	Runtime Policy Enforcement	Physical Condition Prediction	User Policy Correction
IoTSafe	✓	✓	✓	✓	✓
IoTGuard [11]	✓		✓		
Peeves [7]	✓	✓			
Helion [28]	✓		✓	✓	
iRuler [43]	✓				✓
IoTMon [18]	✓				
Menshen [8]	✓				✓
HomeGuard [15]	✓				✓

iRuler [43] uses SMT and model checking to detect vulnerabilities among IFTTT rules, especially in rule configurations of IoT deployments. Menshen [8] builds and checks the Linear Hybrid Automata (LHA) model for user rules to find violations and generate fix suggestions for non-expert users. HomeGuard [15] uses SMT models to check rules and configurations to find cross-app interference threats among smart home apps. Salus [26] leverages formal methods to localize faulty user-programmable logics and uses model checking tools or SMT tools to debug these logics. ProvThings [44] automatically instruments IoT apps and device APIs to a centralized audit IoT platform and generate data provenance that provides a holistic explanation of system activities. These four studies can identify inter-app interactions based on static analysis, but they cannot capture real physical interactions during runtime. Moreover, none of these systems can predict violations and perform policy enforcement in a preventative manner.

VII. Discussion

In this section, we discuss the limitations of IoTSafe and potential solutions to address these limitations.

Physical Interaction Discovery: Identifying physical interaction based on static analysis suffers from the overestimation issue (*i.e.*, high false positives). We point out that it is necessary to discover real physical interactions through dynamic analysis (*e.g.*, dynamic testing). To the best of our knowledge, this is the first study on dynamic testing based physical interaction discovery in IoT. Physical interactions among IoT devices are subject to both spatial and temporal contexts of the smart home environment, such as deployed locations of IoT devices, and even the outside environment. In this work, we do not consider the impact of the outside environment and temporal dynamics on physical interactions (*e.g.*, time and seasonal impacts). We leave the enhancement of our approach by considering both outside environment impact and temporal dynamics to our future work.

Dynamic Testing Safety and Optimization: One distinctive feature of the dynamic testing for identifying physical interactions from the traditional software testing [12] is that our dynamic testing process has physical impacts on the smart home environment (*e.g.*, temperature changes). Changing the physical environment may lead to risky situations and cause safety issues during the testing. Hence, certain functions of devices cannot be tested or can only be tested in a restricted range, which may miss some physical interactions (*e.g.*, a toaster to the smoke channel). Another challenge is the testing overhead, which is related to the ordering of exploring/testing possible paths. As part of our future work, we will formulate this problem as an optimization problem to minimize the testing overhead and ensure the testing safety.

Human Interaction: IoTSafe mainly considers interactions between devices and does not consider direct users' interactions in the environment. A limitation of IoTSafe is that it requires a dynamic testing period, which can be interfered by human activities. While human activities (*e.g.*, walking, turning on devices) may cause false-positive interaction results during the dynamic testing. The prediction also does not consider the physical influence from human activities. In our future work, we plan to address this challenge on how to identify and predict influences from human activities.

Applicability and Generality: Although we instantiate IoTSafe on the SmartThings platform, our design can be potentially applied to other IoT platforms, such as the OpenHAB [31] platform. The dynamic testing process is platform-agnostic, and thus it can work across different IoT platforms. IoTSafe requires a real-world physical interaction discovery phase, which takes time and incurs certain overhead. In addition, any false positive of the policy enforcement (*e.g.*, a benign action is incorrectly blocked) may hurt usability. In our future work, we plan to perform user studies to evaluate the usability of the dynamic testing process and policy enforcement in IoTSafe.

VIII. Conclusion

In this work, we have presented IoTSafe, a novel IoT dynamic security and safety assessment with physical interaction discovery. To our knowledge, this paper proposed the first dynamic testing method for IoT physical interaction discovery. Based on the identified physical interactions, IoTSafe generates devices' physical models, which predict incoming risky situations and blocks unsafe device states. We have implemented a prototype of IoTSafe on the SmartThings platform. We have evaluated IoTSafe in a simulated smart home environment. IoTSafe identifies 39 real physical interactions out of 21 applications. IoTSafe also successfully predicts 96% highly risky conditions in our experiments.

Acknowledgment

We thank our shepherd, Adam Bates, and the anonymous NDSS reviewers for their valuable comments. This work is supported in part by the National Science Foundation (NSF) under the Grant No. 2031002, 1846291, 1642143, and 1700499.

References

[1] American home connected devices amount. https://variety.com/2019/digital/news/u-s-households-have-an-average-of-11-connected-devices-and-5g-should-push-that-even-higher-1203431225/, 2020. [Accessed 07-01-2020].

[2] Android Things. https://developer.android.com/things, 2020. [Accessed 05-01-2020].

[3] Homeowner's Blood 'Ran Cold' as Smart Cameras, Thermostat Hacked, He Says. https://www.nbcchicago.com/news/national-international/my-blood-ran-cold-as-smart-cameras-thermostat-hacked-homeowner-says/6523/, 2020. [Accessed 012-01-2020].

[4] MultiSensor 6. https://aeotec.com/z-wave-sensor/, 2020. [Accessed 05-01-2020].

[5] Zooz 4-in-1 Sensor. https://www.getzooz.com/zooz-zse40-4-in-1-sensor.html, 2020. [Accessed 05-01-2020].

[6] S. Birnbach, S. Eberz, and I. Martinovic. Peeves: Physical event verification in smart homes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19).*

[7] S. Birnbach, S. Eberz, and I. Martinovic. Peeves: Physical event verification in smart homes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1455–1467. ACM, 2019.

[8] L. Bu, W. Xiong, C.-J. M. Liang, S. Han, D. Zhang, S. Lin, and X. Li. Systematically ensuring the confidence of real-time home automation iot systems. *ACM Transactions on Cyber-Physical Systems*, 2(3):1–23, 2018.

[9] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity iot. In *27th USENIX Security Symposium (USENIX Security)*, pages 1687–1704, 2018.

[10] Z. B. Celik, P. McDaniel, and G. Tan. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2018.

[11] Z. B. Celik, G. Tan, and P. McDaniel. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Proceedings of the 23th Network and Distributed Security Symposium (NDSS)*, 2019.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 22nd Network and Distributed Security Symposium (NDSS)*, 2018.

[13] Z. Cheng, W. W. Shein, Y. Tan, and A. O. Lim. Energy efficient thermal comfort control for cyber-physical home system. In *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 797–802, 2013.

[14] H. Chi, Q. Zeng, X. Du, and J. Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. *CoRR*, abs/1808.02125, 2018.

[15] H. Chi, Q. Zeng, X. Du, and J. Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 411–423. IEEE, 2020.

[16] C. Cobb, M. Surbatovich, A. Kawakami, M. Sharif, L. Bauer, A. Das, and L. Jia. How risky are real users'{IFTTT} applets? In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 505–529, 2020.

[17] S. Community. Samsung smartthing applications. https://github.com/SmartThingsCommunity/SmartThingsPublic, 2017.

[18] W. Ding and H. Hu. On the safety of IoT device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 832–846. ACM, 2018.

[19] O. S. En, M. Yoshiki, Y. Lim, and Y. Tan. Predictive thermal comfort control for cyber-physical home systems. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, pages 444–451. IEEE, 2018.

[20] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (IEEE S&P'16)*, pages 636–654.

[21] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, May 2016.

[22] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016.

[23] B. Fouladi and S. Ghanoun. Honey, I'm home!!-Hacking Z-Wave Home Automation Systems. *Black Hat USA*, 2013.

[24] Google. Google cloud console. https://console.cloud.google.com/, 2019.

[25] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proceedings of the 21st Network and Distributed Security Symposium (NDSS)*, February 2017.

[26] C.-J. M. Liang, L. Bu, Z. Li, J. Zhang, S. Han, B. F. Karlsson, D. Zhang, and F. Zhao. Systematically debugging iot control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, pages 133–142, 2016.

[27] N. Lomas. Critical Flaw identified In ZigBee Smart Home Devices. https://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/.

[28] S. Manandhar, K. Moran, K. Kafle, R. Tang, D. Poshyvanyk, and A. Nadkarni. Towards a natural perspective of smart homes for practical security and safety analyses. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 482–499. IEEE, 2020.

[29] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel. Iotsan: fortifying the safety of iot systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 191–203. ACM, 2018.

[30] A. Omar, L. Chaz, s. A. Mano, and M. Fabian. Sok: Security evaluation of home-based iot deployment. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[31] OpenHAB. openhab - features - introduction. http://www.openhab.org/features/introduction.html.

[32] W. R. Ott. Mathematical models for predicting indoor air quality from smoking activity. *Environmental Health Perspectives*, 107(suppl 2):375–381, 1999.

[33] G. Petracca, Y. Sun, T. Jaeger, and A. Atamli. Audroid: Preventing Attacks on Audio Channels in Mobile Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 181–190. ACM, 2015.

[34] S. O. A. Repository. Smartthings official app repository. https://github.com/SmartThingsCommunity, 2019.

[35] E. Ronen and A. Shamir. Extended Functionality Attacks on IoT Devices: The Case of Smart Lights. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 3–12. IEEE, 2016.

[36] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*, pages 195–212. IEEE, 2017.

[37] V. Sivaraman, D. Chan, D. Earl, and R. Boreli. Smart-Phones Attacking Smart-Homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, pages 195–200. ACM, 2016.

[38] S. Smartthing. Smart home. intelligent living. https://www.smartthings.com/.

[39] S. SmartThings. Samsung smartthings capabilities. http://docs.smartthings.com/en/latest/getting-started/first-smartapp.html.

[40] N. H. Son and Y. Tan. Simulation-based short-term model predictive control for hvac systems of residential houses. *VNU Journal of Science: Computer Science and Communication Engineering*, 35(1), 2019.

[41] A. TenWolde and C. L. Pilon. The effect of indoor humidity on water vapor release in homes. 2007.

[42] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague. SmartAuth: User-Centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pages 361–378, 2017.

[43] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1439–1453, 2019.

[44] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*, 2018.

[45] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, pages 1074–1088.

[46] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1133–1150, Santa Clara, CA, Aug. 2019. USENIX Association.

APPENDIX

A. Safety and Security Policies

We list 36 reference safety and security policies in Table VIII. For the general safety and security policies, users can directly enable them in IoTSAFE without device-specific settings. Our system implements these policies to all related devices within the same capability. Table VIII also gives reference policies that users need to modify or assign specific devices to enable such policies. The policies will be only triggered by the associated sensors or devices.

B. Smart Home Testbed Setting

We installed 21 official SmartThings (ST) apps selected from the SmartThings GitHub repository [34], which enables automated tasks for the smart home environment, as shown in Table IX. In the table, we list the setting (apps and their configurations) for each testing group. We tested these apps in seven groups with different configurations, which are set according to app instructions and common sense. Each group includes a set of apps that may be installed together in a smart home environment. Groups 1-4 use apps and settings based on our experiences and Groups 5-7 are generated based on the user survey data from prior work [28], [16].

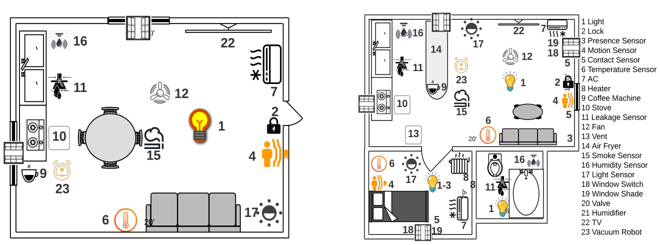
The first group (Group 1) contains 5 temperature-related apps. We use this group to test the effectiveness on identifying temperature-related physical interactions and detecting policy violations between devices. The second group (Group 2) contains 4 light management apps. It aims at learning interactions between illuminance related devices, such as bulbs, shades and light sensors. The third group (Group 3) contains 2 light-related and 4 temperature-related apps. It is used to test cross-channel physical interactions. The fourth group (Group 4) contains 21 apps, which are used to test interactions related to multiple physical channels. Apps in Group 5-7 are generated from real users' routines from existing user studies [28], [16]. We choose routines that are strongly/somewhat agreed by real users. Group 5 contains 18 apps, including temperature, motion, smoke, and light control apps. Group 6 contains 19 apps, including home mode, smoke, motion, temperature and light control. Group 7 contains 21 apps, including temperature, motion, home mode, smoke, humidity, and leakage control apps. We configure apps with multiple devices based on the descriptions of apps and the locations of devices as shown in Figure 9.

C. Home Layouts

Figure 13(a) lists the room layout and device locations for Groups 1, 2, and 5. The differences between groups are that Group 2 does not use temperature related devices, such as AC and heater. Figure 13(b) shows the room layouts for Groups 3, 4, 6 and 7. The detailed devices and apps deployment can be found in Table IX.

TABLE VIII: Safety and security policies (✗ means a physical channel is involved in the policy, T represents temperature, H represents Humidity, S represents smoke, W represents water leakage, SM represents soil moisture, MO represents motion, and I represents illuminance)

No.	Policy Description	T	H	S	AQ	W	SM	MO	I
General safety and security policies									
1	All electrical appliances should be turned off when smoke is detected.			✗					
2	All window and blinds should be closed when home mode is away.	✗	✗						✗
3	All lights should be on when home mode is vacation.	✗	✗						✗
4	Open the window when a related room's temperature is detected above the threshold and there is people home.	✗	✗						
5	Turn off the humidifier, if a related room's humidity exceeds the threshold.		✗						
6	Turn on all air purifiers when air quality is below a threshold				✗				
7	Temperature should not exceed up-threshold when people are at home.	✗						✗	
8	The water valve must shut off when the water/moisture sensor detects leak and no smoke detected.					✗	✗		
9	When smoke is detected, an SMS/Push message should be sent to the owner.			✗					
10	The sprinkler valve should be ON when detecting smoke.			✗		✗			
11	When there is water leakage, an SMS/Push message should be sent to the owner					✗	✗		
12	When smoke is detected, the lights must be turned on in night mode, and the door must be unlocked if someone is at home.			✗				✗	✗
13	The alarm must sound when smoke or CO is detected.			✗					
14	The alarm must sound and an SMS/Push message should be sent to the owner, when motion is detected and home mode is away.							✗	
Device-specific safety and security policies									
1	A water valve should be opened when smoke is detected			✗		✗			
2	Turn off the gas stove, if the smoke is detected.	✗	✗	✗					
3	Turn off the heater, if the smoke is detected.	✗	✗	✗					
4	The AC should be turned off when smoke is detected.	✗		✗					
5	Turn on the vent when temperature is detected to above the threshold.	✗	✗						
6	An AC is turned to heating when temperature is detected to be below the lower threshold	✗	✗						
7	Turn on the heater, if the temperature is below the threshold.	✗	✗						
8	The AC should be turned to cooling if temperature exceeds the upper threshold.	✗	✗						
9	An humidifier is turned ON when humidity is detected to below the threshold.		✗						
10	An vent is turned ON when humidity is detected to above the threshold.	✗	✗						
11	An air purifier is turned ON when air quality is below a threshold				✗				
12	Turn off the valve, if the water level exceeds the threshold.					✗			
13	Turn on the valve, if the water level is below the threshold.					✗			
14	The garden sprinkler should be ON when soil moisture is detected to be low					✗	✗		
15	The garden sprinkler should be OFF when soil moisture is detected to be exceeded					✗	✗		
16	Temperature should be within a predefined range when people are at home.	✗	✗					✗	
17	The heater should be turned off when temperature is above a threshold and no one is at home.	✗	✗					✗	
18	Soil moisture should be within a predefined range					✗	✗		
19	A water valve should be closed when a water sensor's state is wet.					✗	✗		
20	The valve must be closed when water sensor is wet and when the water level threshold specified by a user is reached.					✗	✗		
21	The HVACs, fans, switches, heaters, dehumidifiers must be off when the humidity and temperature values are out of the threshold specified by the user(e.g., a particular degree above/below the threshold of temperature and humidity).	✗	✗	✗					✗
22	The sprinkler system must not be on when it rains, and when the soil moisture below a threshold defined by a user. Flood sensor must activate the alarm when there is water.					✗	✗		



(a) Groups 1, 2, 5 Layout (b) Groups 3, 4, 6, 7 Layout

Fig. 13: Room layouts.

D. User Restrictions

To demonstrate the effectiveness of our dynamic testing with user restrictions, we generated a testing case to compare results of two testing methods. In Figure 14, we first tested a sequence of humidity-related testing cases without any user restriction. Each case turns on a humidifier for 10 minutes. We have two sensors with report intervals of 8 minutes and 15 minutes, respectively. Then, we tested the same sequence with a restriction "humidity $< 85\%$". In the first case, the humidity was raised to 98% after 10 minutes. With the user restriction, the system stopped the testing process when a sensor reading of 88% humidity was reported, keeping the home environment from risky physical situations.

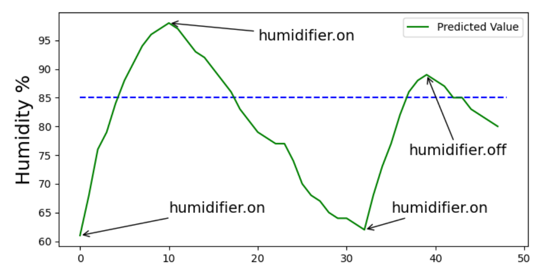


Fig. 14: Example of user restriction function.

TABLE IX: Detailed applications for each group.

Group	App Type Description	Amount
1	Home mode controls temperature devices	2
	Temperature sensor controls heaters, AC, fan, and windows	3
2	Home mode controls lights and locks	2
	Motion Sensor controls Lights	1
	Light Sensor controls Window shades	2
	Light Sensor controls lights	1
3	Home mode (away) controls window(close), AC(off), Heater(off)	1
	Home mode (home) controls AC.cooling and AC.heating	1
	Temperature Sensor controls window, heater, AC and alarm	2
	Light Sensor controls Window shades	2
	Light Sensor controls Lights	1
4	Humidity Sensor controls windows	2
	Presence sensor controls Home mode	1
	Home mode controls AC, windows, and heaters	2
	Motion sensor controls lights	2
	Light sensor controls Lights	1
	Temperature sensor controls heaters, AC, windows, alarms and fan	3
	Humidity sensor controls humidifier	1
	Timer controls Home mode	2
	Home mode controls lights and TVs	1
	Smoke Sensor controls alarms and locks	1
	Home mode and motion sensor control alarms	2
	Home mode controls coffee and lights	1
	Leakage Sensor controls valves	1
5	Home mode and Timer controls lights, AC, coffee, and alarms	6
	Carbon monoxide sensor controls home mode	1
	Motion sensor controls home mode	1
	Light Sensor controls lights	2
	Temperature sensor controls heaters, AC, windows, alarms and fan	3
	Timer controls home mode and lights	3
	Smoke Sensor controls alarms and home mode	1
	Home mode and motion sensor control alarms	1
6	Home mode and Timer controls lights, AC, coffee, and alarms	6
	Home mode and motion sensor control alarms	2
	Presence sensor controls Home mode	2
	Light Sensor controls lights	2
	Temperature sensor controls heaters, AC, windows, alarms and fan	3
	Timer controls home mode and lights	1
	Home mode controls lights and TV	1
	Smoke sensor controls alarms, locks and home mode	1
	Leakage sensor controls home mode and plugs	1
7	Presence sensor controls Home mode	2
	Home mode and motion sensor control lights, alarms	2
	Home mode and Timer controls lights, AC, coffee, locks, heaters, and alarms	6
	Motion Sensor controls lights	1
	Light Sensor controls lights	2
	Temperature sensor controls heaters, AC, windows, alarms and fan	3
	Timer controls home mode and lights	2
	Smoke sensor controls alarms, locks and home mode	1
	Leakage sensor controls home mode and plugs	2

E. Physical Modeling

Here, we provide more details of the physical models. For the air related physical channel modeling, we mainly focus on building models for the temperature and humidity. We start from the general model, which is presented in Section IV-C. The general modeling of the air channels is listed as follows:

$$\Delta SensVal(D_i, state, t_1, t_2) = \frac{\Delta Q(D_i, state, t_1, t_2)}{\beta}, \quad (5)$$

As mentioned in Section IV-C, $\Delta Q(D_i, state, t_1, t_2)$ represents the amount of channel quantity changes of device D_i from time t_1 to t_2. The heat gain/loss Q from temperature-related devices mainly has two types: 1) heat flow from devices like the AC or window fans; and 2) heat radiation from devices like heaters. For the heat radiation, the parameter Q_i can be calculated by using its power or connected smart plug's power multiplied by its working time. However, for devices that we cannot directly know its power or for devices that have airflow abilities, *e.g.*, AC, we use the following equation [40] to capture the heat from its air flows. These heat flows also depend on several environmental parameters, including the room temperatures, outside temperature, and specific attributes of devices.

$$\Delta Q_{flow} = \rho \cdot C_{air} \cdot V_{air} \cdot (T_{set} - T_{room}) \cdot (t_2 - t_1) \quad (6)$$

Equation (6) describes the calculation of ΔQ_{flow} for air flow heating and cooling devices, where ρ is the density of the air, V_{air} is the air volume flow rate in ft^3/min, and C_{air} is the specific heat capacity of the air. T_{set} is the output air temperature of the devices and T_{room} is the current room environment temperature. For window fans, T_{set} is the outside room temperature. $(t_2 - t_1)$ is the working time of devices. We list the detailed values of these parameters in Table X in our experiments. Other air related physical channels like air quality, smoke, and humidity can also be calculated in a similar way.

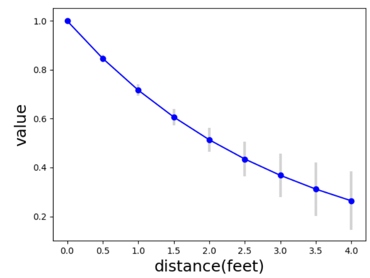


Fig. 15: Distance factor value for a humidifier.

We also build a model for the relative humidity channel. The humidity also has two sources: 1) devices like humidifier or kettle, and 2) air flow devices. The humidity gain/loss from devices like a humidifier can be represented as follows:

$$\Delta Q_{humidity} = \gamma_d \cdot H_i \cdot (t_2 - t_1) \quad (7)$$

where $\Delta Q(D_i, state, t_1, t_2)$ is the changed weight of water vapor in the air. H_i is the water vapor rate of the device, which can be calculated by readings from surrounding sensors, or known by devices' default parameters [41]. We also can estimate the vapor evaporation rate from the humidifier's power meter/manual. γ_d is the distance factor, which represents the distance between the vapor generation device and the location of testing point (*e.g.*, a humidity sensor). We define this parameter because the humidity is sensitive to the distance for these type of devices (*e.g.*, kettle and humidifier). The temperature is more evenly distributed in the room so it does not need this factor. The parameter values of this factor with the increasing distance is shown in Figure 15.

The humidity gain/loss ΔQ_{Hflow} from a device like an AC or window fan can be represented as follows [41]:

$$\Delta Q_{Hflow} = \rho \cdot H_{air} \cdot V_{air} \cdot (H_{set} - H_{room}) \cdot (t_2 - t_1) \quad (8)$$

We do not use the distance factor γ_d here because such devices can evenly change the humidity in a room. Q_{Hflow} represents the air humidity gain/loss amount from the airflow, which is the product of airflow volume V_{air}, air density ρ and absolute air humidity capacity H_{air}. The absolute air humidity capacity H_{air} is a physical attribute that changes with the temperature. Hence our system stores a list of its default values for different temperature ranges. H_{set} is the relative humidity of the airflow and H_{room} is the relative humidity of the room.

TABLE X: Common parameters in physical modeling.

Parameters	Values
Density of air	1.2 kg/m^3
Heat capacity of air	1.005 kj/kg°C
Absolute humidity capacity of air(68F)	17.3 g/m^3
Air flow rate of AC	150 ft^3/min
Air flow rate of window fan	100 ft^3/min
Evaporation rate of a humidifier	2.57g/min

In Table X, we list default parameters that are used for calculating our physical models. For example, the air density and capacity are used to calculate β in Equation (5).