User-in-the-loop Policy Enforcement with Cross-App Interaction Discovery in IoT Platforms

Rui Chen

B.S. Computer Science, University of Wyoming, 2020

B.S. Computer Engineering, University of Wyoming, 2020

Submitted to the graduate degree program in Electrical Engineering and Computer Science Department and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

	Fengjun Li, Chair
Committee members	Bo Luo
	Alex Bardas

Date defended: June 6, 2022

The Thesis Committee for Rui Chen certifies
that this is the approved version of the following thesis :

User-in-the-loop Policy Enforcement with Cross-App Interaction Discovery in IoT Platforms

Fengjun Li, Chair

Date approved:

Abstract

The Internet of Things platforms have been widely developed to better assist users in designing, controlling, and monitoring their smart home system. These platforms provide a programming interface and allow users to install various IoT apps published by third parties. As users could obtain the IoT apps from unvetted sources, a malicious app could be installed to perform unexpected behaviors that violate the user's security and safety, such as opening the door when no motion is detected. Additionally, prior research shows that due to the lack of access control mechanisms, even benign IoT apps can cause severe security and safety risks by interacting with each other in unanticipated ways. An improved access control system is needed to detect and monitor unexpected behaviors from IoT apps to address such threats. In this thesis, we provided a system called IoTDiscover that detects single app conflicts and interaction threats between the trigger-action behavior of an IoT app. The IoTDiscover also generates resolution policies to resolve the detected conflicts by including a user-in-the-loop design. The code analysis and instrumentation will be applied to collect related information used for conflict discovery and policy enforcement, such as the actual trigger-action behavior of an app, configuration information, and runtime trigger event/action information. A policy enforcement module in the system will be encoded with policies, and it will enforce the policies at runtime by blocking actions that violate the policy. We implement and evaluate IoTDiscover with 17 official SmartApps and two malicious SmartApps on the SmartThings platform with five testing cases. As a result, IoTDiscover successfully detected all single app conflicts and interaction threats in each testing case. The resolution policies are generated and enforced effectively by blocking all violated actions.

Acknowledgements

I would like to express my deepest appreciation to my advisor, Fengjun Li, who assisted me through the research process with invaluable feedback and patience. Additionally, I would like to thank all committee members for their consideration for this work.

I would like to thank Zhaohui Wang, who provided a valuable system to improve and assist my research project.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement through my years of study.

This accomplishment would not have been possible without all of your support. Thank you again.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Internet-of-Things (IoT) is a network of smart objects or “Things” embedded with sensors, actuators, software applications, and other technologies connected through the Internet. They can communicate and share data with each other. The rapid development of IoTs technology facilitated the smart objects to be used as the fundamental component in various industries such as transportation, healthcare, agriculture, industrial automation, and especially smart homes. According to the reports, as of 2021, 35 billion IoT devices were installed worldwide [23], and around 43% of U.S. households have owned at least one smart home device [9]. Smart home or home automation consists of different types of IoT devices, including home appliances, lighting, heating, and cooling systems, and various home security and safety system [28]. Such IoT devices can be monitored and remotely controlled through IoT platforms, which is an ecosystem that connects all IoT devices with a hub as a centralized gateway and provides a software environment to allow third parties to develop and publish their IoT apps that assist users in managing their IoT devices. In recent years, a number of smart home platforms have been developed and used in people’s life, such as Samsung’s SmartThings [7], OpenHAB [6], Apple’s HomeKit [4], and HomeAsistant [3].

The proliferation of IoT devices and platforms advanced our living environment to be more convenient, efficient, and autonomous. However, the complex IoT environment also makes IoT security and privacy becomes a serious and challenging problem. In most attacks against IoT environments, attackers exploit the vulnerability of IoT platforms to take control of devices to perform malicious behavior that owners and administrators do not expect. Fernandes et al. [18] provided a study on Samsung’s SmartThings and discovered the overprivileged problem due to the SmartThings permission/capability model design, where a capability defines a set of commands and attributes. The coarse-grained capabilities make the SmartThings platform allow a SmartApp with any given capabilities to gain unauthorized access to the device; for example, given a SmartApp permitted with *capabilities.lock* can access both *lock* and *unlock* commands, where the SmartApp only authorized to access *lock* command. In addition, due to the coarse SmartApp-SmartDevice binding, the SmartApp that is authorized to access one of the capabilities of a selected device can also gain access to all other capabilities of the same selected device. As the IoT platforms allow users to install and authorize third-party IoT apps, which may be obtained from unvetted sources and contains malicious code, attackers can exploit the overprivileged problem to construct malicious app to access unauthorized control of devices in a dangerous manner, such as *unlock the door when the user is sleeping*. While users are relying on app descriptions to acquire app functionality information, they will not understand all functionalities of an app from descriptions, so they will not be aware when a malicious action that differs from the app description occurs.

IoT platforms allow users to install various IoT apps in the same smart home environment. IoT apps developed by third-party are mostly designed based on the *trigger-action* paradigm [27], where the *trigger* is the external information or any device state, such as *when motion is detected* that enables the app to send a command of an action to control a device, such as *turn on the light*. While the action command sent from an app changes the device state, which may trigger other apps, some unanticipated interaction chain may happen between benign IoT apps in a complex IoT environment. For instance, one app has a rule "*turn on the light when it is dark (illuminance below threshold)*", which will enable another app as a rule "*open the window when light is on*". Moreover, since one device can be controlled by more than one app as well as multiple apps can be enabled by the same trigger simultaneously, two apps may send conflicting actions under the same trigger to the same device, such as one app has a trigger-action rule "*turn on the light when motion is detected*", while another app has a rule "*turn off the light when motion is detected*".

In order to address such problems, existing research efforts [29] [24] are applied Natural Language Processing (NLP) to extract information from app descriptions provided by the manufacturer or developer and compare it with the result of code analysis to verify whether the functionality and capabilities of the IoT apps are following the original design goal and identify the overprivileged problem in the apps. Since these works are mostly focused on unexpected behavior of a single app caused by the overprivileged problem, other researchers have been working on discovering cross-app interaction problems with static analysis [13] [16] [11] [22] and applied policy enforcement techniques to capture and monitor the app behavior at runtime based on pre-defined policies [27] [26] [17] [14]. These policies are specified by policy writers or directly from the user's expectation and will be described as a particular policy language. The policy enforcement system will only accept the app to send an action command that satisfies all given policies; otherwise, the action will be denied. However, the policies in these works are pre-defined by policy writers based on a certain IoT environment, which is hidden from the user. When a new device or new IoT app is added to the IoT environment, the existing policy may not be effective for the new device and app. In addition, some efforts allow users to define their policy, which requires the user to have high-level knowledge of the functionality of each app and policy language.

In this thesis, we provided a dynamic policy enforcement system that discovers all single-app conflicts and interaction threats and generates resolution policies that are used to restrict unexpected behaviors. The system consists of four major steps: (1) *app information collection* that consists of code analysis to extract actual trigger-actions and code instrumentation that collects necessary information to be used for conflict discovery and policy enforcement, such as app's settings/configurations, user's decision for actual trigger-actions, and guarded event and action information at runtime, (2) *conflicts discovery and resolution* module takes configuration information and the user's decision of trigger-actions from instrumented app to detect all possible single app conflicts and interactions between each trigger-action, and generate resolved condition-based policies based on different types of conflicts, these rules will be extracted as safety, security and functionality property feature lists and pass to policy generation (3) once the *policy generator* received the property features, it will generated as policy languages that will be encoded in policy enforcement module, (4) *policy enforcement* module enforcs the policies by verifying guarded event and action information from instrumented app against the policies and makes decisions such as “allow” if the event and action passed all policies or “deny” otherwise.

We evaluated IoTDiscover with five testing cases and manually selected several SmartApp in each case. The SmartApps are selected from the SmartThings official marketplace [7], and two of them are malicious apps from IoTBench [5]. As a result, our IoTDiscover system could efficiently detect any types of conflicts and enforce the policies to block any violations.

Chapter 2

Background and Related Works

2.1 IoT Architecture and Security

In general, the Internet of Things or IoT is a network that contains a number of physical devices which are allowed to communicate with other devices and are controlled by IoT applications through the network. The basic IoT architecture shown in Figure 2.1 can be divided into three layers: the Perception layer, the Network layer, and the Application layer. The *perception layer* is also called the sensing layer, which consists of physical IoT devices to collect and transmit information. The physical IoT devices are embedded with low-power nodes such as sensors and actuators that perform different functionalities such as querying temperature, humidity, illuminance, motion, acceleration, etc. [10]. As those nodes are vulnerable to various attacks, attackers can capture or replace the node with a malicious node or inject malicious code or false data into the nodes' memory. The *network layer* is responsible for transmitting information collected from the perception layer to different process systems such as clouds and IoT applications through wireless communication protocols like Wi-Fi, Bluetooth, ZigBee, and Z-Wave. The network layer may consist of IoT hubs, switches, or routers. Similar to the traditional networking layer of the TCP/IP model, the IoT networking layer would perform all the same tasks. Thus, this layer is susceptible to the same attacks as traditional networking layers, such as DDoS attacks [21], MITM attacks [25], and Routing attacks (malicious nodes in an IoT application may try to redirect the routing paths during data transmitting). The *application layer* provides high-quality smart services to the end-user and allows users to control and manage their IoT devices and information. In recent years, different IoT applications have been developed in this layer, such as smart homes, smart cities, industrial au-

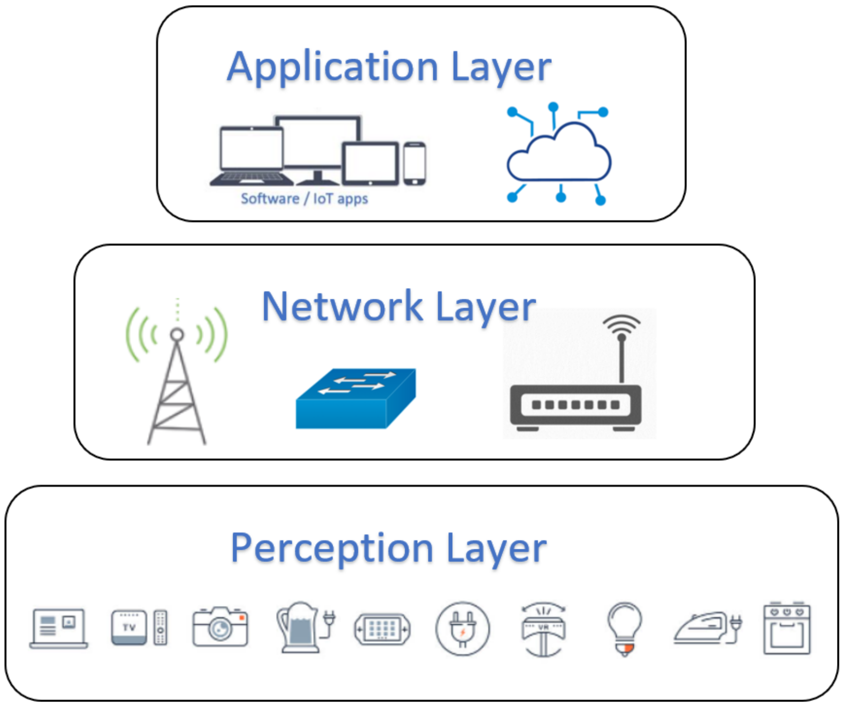


Figure 2.1: IoT Architecture. *Perception Layer* - the sensing layer consists of physical IoT devices, *Network Layer* - transmitting information between perception layer and application layer, *Application Layer* - provides high-quality smart services to the end-user for controlling and monitoring IoT devices.

tomation, and smart healthcare. Most IoT applications provide a platform to allow users to install, develop, and manage their self-designed or third-party developed IoT apps that control IoT devices based on users' expectations. However, due to the lack of access control mechanisms, even benign IoT apps can cause security and privacy problems by interacting with each other in unanticipated ways. More detail about such IoT platform is discussed in Section 2.2.

2.2 Smart Home Platforms

To better assist users in designing, controlling, and monitoring their smart home system, many smart home platforms have been developed to provide an ecosystem that allows users to centralize all home devices and install IoT apps published by third-party. Such platforms typically adopt a similar cloud-backed architecture that contains three components: hub, backend cloud, and smartphone app. In this thesis, we will only focus on the Samsung SmartThings platform, which is one of the most popular smart home platforms.

In Samsung SmartThings, shown in Figure 2.2, the SmartThings Hub is a physical device used to connect all the IoT devices through IoT communication protocols such as ZigBee, ZWave, and Wi-Fi. The Hub is also responsible for bridging the communication between the cloud backend and devices to send commands from SmartApps and update device states to SmartApps. The SmartThings cloud backend provides a Groovy-based sandbox environment to allow the communication between SmartApps and Device Handlers. SmartApps are Groovy-based programs that define automation rules for the home devices, and the Device Handlers are the software-based encapsulation of physical devices and enabled the communication between physical smart home devices and the SmartThings platform [8]. Instead of directly communicating to the physical devices, SmartApps control the device by communicating with Device Handler based on a permission (capability) system. The capabilities have the components of commands and attributes that connected devices can support and specify in Device Handler. For example, a light device with a capability called *capability.switch* has an attributes *switch* and can access two commands *on* and *off*. SmartApps state the capabilities they need during app installation and subscribe to the events from Device Handler as well as send commands to Device Handler to control the device. The SmartThings platform also provides the user interface (UI) with the smartphone app and a Web-based IDE that allow users to manage their hubs, devices, and SmartApps. Users can install different SmartApps from the official SmartThings market or from third parties or develop expected SmartApps by themselves.

2.2.1 Related Works

The security problems of IoT platforms have been broadly explored in many recent research efforts. Fernandes et al. [18] provided a practical security study of the Samsung SmartThings platform and identified several design flaws of the SmartThings permission-based model, such as the overprivileged problem caused by coarse-grand capability and coarse SmartApp-SmartDevice binding, which allows a SmartApp to access unauthorized commands with permitted capabilities. Moreover, they also identified that there exist event leakage and event spoofing problem due to

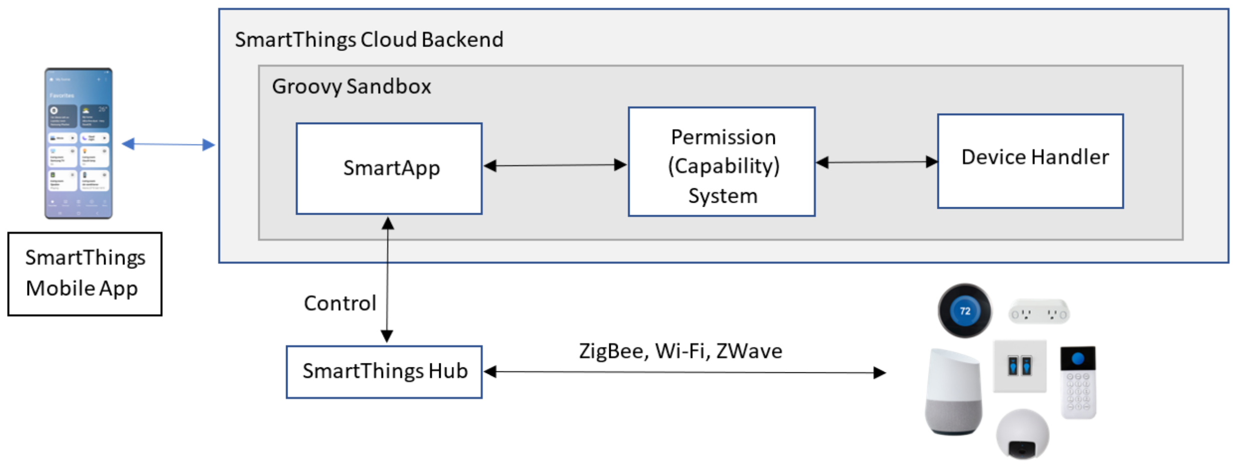


Figure 2.2: Samsung SmartThings Platform Overview.

insufficient sensitive event data protection.

IoT Access Control. To address above problems, Fernandes et al. [19] presented FlowFence, a system based on information flow control with Opacified Computation, and the sensitive data will be explicitly isolated inside the FlowFence-provided sandboxes. Celik et al. [12] provided a static taint analysis tool called SAINT to identify sensitive sources and sinks as well as identify sensitive data flows. To address the overprivileged problem, Jia et al. [20] proposed ContexIoT, a context-based permission system for IoT platforms that identifies fine-grained context action information before execution and prompts the user at runtime with such information to provide contextual integrity as well as request for user's permission. However, the in-context prompt requires the user to be involved in approving permissions at runtime, which does not satisfy the real-time automation of IoT apps. Tian et al. [24] introduced SmartAuth, which automatically extracts code annotations and capability requests of a SmartApp by performing static analysis on the app's source code and natural language processing (NLP) on the app's description provided by the developer. SmartAuth then compares the app's actual functionality from the source code and the functionality described in the app description to detect whether an unexpected functionality is occurring. All these current works are only available in detecting single IoT app misbehaviors, while they are not concerned with interaction behavior between multiple IoT apps in a complex IoT environment.

Cross-App Interactions. Chi et al. [15] introduced the new threat, Cross-App Interference (CAI) threats, which could be caused by unanticipated or malicious interactions between multiple apps on IoT platforms. The CAI threats contain three categories based on the *trigger-condition-action* model of app rules: *Action-Interference Threats*, *Trigger-Interference Threats*, and *Condition-Interference Threats*. To detect the CAI threats, they performed a HOMEGUARD system by using symbolic execution techniques to extract and analyze IoT apps' execution path and implementation. Celik et al. [13] provided a static analysis system, called SOTERIA, that also applies a symbolic execution technique for model checking to verify whether an IoT app or IoT environment violated the safety, security, and functional properties. There are many other existing works based on a static analysis approach to identify potential unexpected behavior caused by the interaction between multiple IoT apps [11] [22]. However, although these works effectively detect potential interference threats of IoT apps, they do not provide mitigation solutions after detected violations to protect apps from these threats.

Policy Enforcement. Policy enforcement techniques have been widely adopted in many existing works to monitor and control unexpected interactions of multiple IoT apps at runtimes, such as IoTGUARD [14], EXPAT [27], and PATRIoT [26]. These approaches usually start with code instrumentations to instrument the source code with a guard to monitor the actions and capture related information of IoT apps at runtime. The policies will be pre-defined by the policy writer [14] [26] or self-defined policy based on the user's expectations [27] and described in policy language. The runtime action information captured by the guard will be sent to a policy enforcement engine that enforces the policy by blocking violated actions and allowing actions that are satisfied with all policies. All these works are lacking in the policy design. IoTGUARD and PATRIoT are pre-defined policies by the policy writer from manually investigating the rules, which relies on the trust of the policy writer to provide benign and correct policies. In addition, the user is unaware of which action is violated since the pre-defined policies are concealed from them. The system of EXPAT allowed users to define policies based on their expectations and provided a policy verification to verify that the users-defined policies were not conflicting. However, this system requires the user to have high-level knowledge of the functionality and interactions of the entire IoT system, which

Name	Single-App Conflicts	Cross-App Interactions	Presenting Recommended Policies	Runtime Policy Enforcement
ContexIoT [20]	$\surd$, ○			
SmartAuth [24]	$\surd$, ○			
HomeGuard [15]		$\surd$		
Soteria [13]		$\surd$		
IoTGuard [14]		○		$\surd$
PatrIoT [14]		○		$\surd$
IoTDiscover	$\surd$, ○	$\surd$, ○	$\surd$	$\surd$

Table 2.1: The comparison of IoTDiscover with other related works, where "$\surd$" in column "single-app conflicts" and "cross-app interaction" denote as the system detects the conflicts, and "○" denotes the system mitigates the conflicts.

could be a challenge to the user since some potential functionality of IoT apps is hidden from the user.

In this thesis, we adopted the CAI threats definition from [15] to define the different interaction threats based on the trigger-action rule patterns. As this work does not provide mitigations when CAI threats are detected, our system will generate resolution policies based on the user's expectations. The PatrIoT [26] is open source and provides expressive policy language and efficient performance with low overhead, and we leveraged the design of policy language and policy enforcement system from PatrIoT. Compared with previous efforts, our work uncovered both the backend app functionality and potential interaction threats to the user and enforced the resolution policies based on the user's expectations at runtime. A summary of the comparison with related works is shown in Table 2.1.

Chapter 3

Threat Model and Problem Statement

3.1 Threat Model

In our threat model, we consider unexpected actions caused by malicious apps or apps with design flaws in a smart home environment. These apps could be installed from unvetted sources or developed by users. We assume: (1) The adversary can exploit the vulnerability of SmartApps, such as overprivileged problem, to access unauthorized action command; (2) The adversary can inject malicious code into an IoT app, or misuse compromised IoT apps to cause unsafe an insecure activities; (3) The adversary does not have the ability to directly access secure devices, such as unlock the front door lock, but the adversary can obtain the information about apps interaction chains and can leverage a compromised app to trigger another app to send action command to secure devices. We also consider that an unintended threats could also happened by multiple benign apps in a complex IoT environment, which directly (electronic channel) or indirectly (physical channel) cause security and privacy issues without an attacker. We assume the IoT devices are trustworthy and consider the adversary's ability to compromise the IoT platform due to vulnerabilities of platforms and software to be out of the scope of this thesis.

3.2 Problem Statement

Any unintended behaviors in an IoT environment can lead to unsafe and insecure situations that violate users' expectations. We consider two problem scenarios that could cause unintended behaviors: (1) *Single-app conflicts*, where an IoT app may implement different functionality as described

```
1  description: "The battery monitor could supervise the battery of your door
       . And when the battery is low, it would send the report to you"
2  input "thebatterymo", "capability.battery", required: true, title:"Where?"
3  input "themotionsensor", "capability.motionSensor", title: "Where?"
4
5  subscribe(thebatterymo, "battery", batteryHandler)
6  subscribe(motionSensor, "motion.inactive", motionHandler)
7
8  def batteryHandler(evt) {
9     sendSMS(phone, "battery low")
10 }
11
12 def motionHandler(evt) {
13    attack()
14 }
15
16 def attack() {
17    def lockState = thebatterymo.currentLock
18    if(lockState != null && lockState == "locked") {
19       thebatterymo.unlock()
20    }
21 }
```

Listing 3.1: Code Snippet of battery monitor [5]. Where the description says "*The battery monitor could supervise the battery of your door. And when the battery is low, it would send the report to you*", but a hidden function will "*unlock the door when the motion sensor detects nobody home*" (line 16-21)

in the app description, and (2) *Multi-app interaction threats*, where IoT apps are individually developed to perform their functions without flaws, but multiple apps in the same environment may have unexpected interactions that will cause some potential safety, security, and privacy threats.

Single-App Conflicts. The user usually relies on app descriptions to learn about app's functionality and install the expected app. However, some real functionalities of an IoT app may not be disclosed to the user in the app descriptions, which causes a discrepancy between the user's perceived functionality and the app's actual functionality [24]. An unexpected action may be triggered surreptitiously without the user's acknowledgment. For example, Listing 3.1 shows a code snippet of a malicious app (battery monitor [5]) described as "*The battery monitor could supervise the battery of your door. And when the battery is low, it would send the report to you*". The user reviewed the app description and installed this app in the smart home. However, a real functionality does

TA_1	Turn on light_1 when motionSensor is active
TA_2	Turn off light_1 when motionSensor is active
TA_3	Turn off light_1 when doorLock is locked
TA_4	Turn on the heater when switch_1 is on
TA_5	Turn on the fan when switch_1 is on
TA_6	Turn on light_1 when motionSensor is inactive
TA_7	Open the window when mode is "home"
TA_8	Set mode to "home" when motionSensor is active
TA_9	Turn off the heater when powerMeter exceeds 3000W
TA_10	Turn on the heater when motionSensor is active
TA_11	Turn on light_1 when TV is on
TA_12	Turn on TV when light_1 is on
TA_13	Open the window when temperature > 70
TA_14	Turn off light_1 when illuminance > 50
TA_15	Turn on light_1 when illuminance < 50

Table 3.1: List of trigger-actions that could cause interaction threats with each other, each interaction threat example will be demonstrated with a pair of two trigger-actions

not disclosed to the user in the app's description, which is a malicious function to "unlock the door when the motion sensor detects nobody home" (line 16-21).

Multi-app Interaction Threats. To help the non-technical user better understand and setup their smart home automation, the IoT apps are generally programmed with a *trigger-action* paradigm that specifies when a trigger event occurs (such as the device state is changed, e.g., *when motion is detected*), one or more action commands will be sent by the app to activate the devices (which changes the device state, e.g., *turn on the light*). As one device could be controlled by more than one app, which leads to the device state change and also enables other apps as a trigger, two or multiple apps can interact with each other and cause unexpected behaviors that violate safety and security. Such unexpected behavior could be conflicting actions and actions triggered by unexpected triggers. In section 3.3, we will describe the details of interaction threats with examples.

3.3 Interaction Threats

To identify the interaction threats of two trigger-actions, we were inspired by the definition of CAI threats [15] and presented our interaction threats based on trigger-action paradigm of IoT

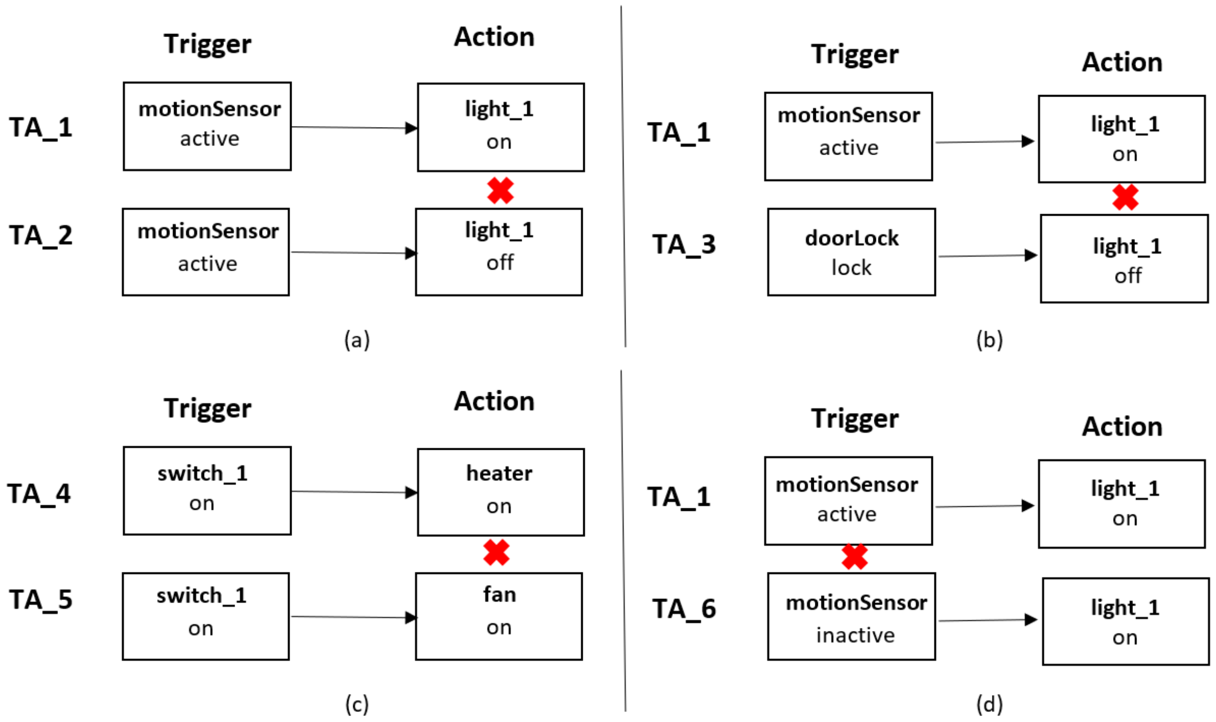


Figure 3.1: Examples of Conflicting Interactions. (a) Same trigger executes opposite actions in electronic channel; (b) two different triggers triggered at same time and executes opposite actions; (c) Same trigger executes opposite actions in physical channel; (d) opposite trigger events ineffectively execute the same action

apps. We consider there are two trigger-actions (TA) installed in the same environment. These two trigger-actions could be specified in two different apps or the same app. We identified two types of interaction threats: *conflicting interactions* and *chained interactions*. To illustrate the examples, we defined 15 trigger-actions shown in Table 3.1.

3.3.1 Conflicting Interactions

To illustrate the conflicting interactions, we consider that two trigger-actions could cause conflicting interactions with three configurations, where (1) two trigger-actions are configured with the same trigger and action devices, (2) two trigger-actions are configured with the same trigger device, but different action device that executed under the same channel, (3) two trigger-actions are configured with different trigger device but activate the same device in opposite commands.

When two trigger-actions are programmed to subscribe with the same trigger device, which is possible that two trigger-actions are triggered simultaneously with the same trigger attributes. In this case, a conflicting action occurs when the two trigger-actions send opposite action commands to the same action device. For example, as shown in Figure 3.1(a), **TA_1** defines "*turn on light_1 when motionSensor is active*", and **TA_2** defines "*turn off light_1 when motionSensor is active*". Both trigger-actions are triggered when the motion sensor detects motion, but sends the opposite command to device light_1, which may cause unpredictable behavior (on/off only, on then off, off then on) and damage to the device. In another case, when two rules are programmed to be triggered by the same trigger device with opposite trigger attributes, the conflict on the trigger is caused when two rules are sending the same action command to the same action device. An example shows in Figure 3.1(d), where **TA_1** and **TA_6** both have the action "*turn on the light_1*", but **TA_1** is triggered by "*when motionSensor is active*" and **TA_6** is triggered by "*when motionSensor is inactive*". As a result, the state of the action device might be immutable when the trigger device is only assigned with two attributes (active/inactive in the example).

In the second configuration, the conflicting interactions could be indirectly caused by the same device when two trigger-actions are operating different devices that act under the same physical channel (e.g., temperature, humidity, and illuminance) with opposite purposes. As shown in Figure 3.1(c), **TA_4** and **TA_5** are both triggered by "*when switch_1 is on*", but **TA_4** sends commands "*turn on the heater*", where the heater is acting under the temperature channel, and the purpose is increasing the temperature, and **TA_5** as "*turn on the fan*", where the fan is also acting under temperature channel but aims to decrease the temperature. This type of conflict could cause energy consumption while two devices are kept active but never reach the final purpose (e.g., temperature decrease to 65F for fan, temperature increase to 75F for heater).

Lastly, there could be a situation where two trigger-actions are triggered by different trigger devices and attributes but send the opposite command to the same action device. In this situation, a conflict action occurs when the trigger is activated at the same time. For example, in Figure 3.1(b), **TA_1** specifies "*turn on light_1 when motionSensor is active*", and **TA_3** specifies "*turn off light_1 when door is locked*", where these two rules are conflicting when motion is detected while the door

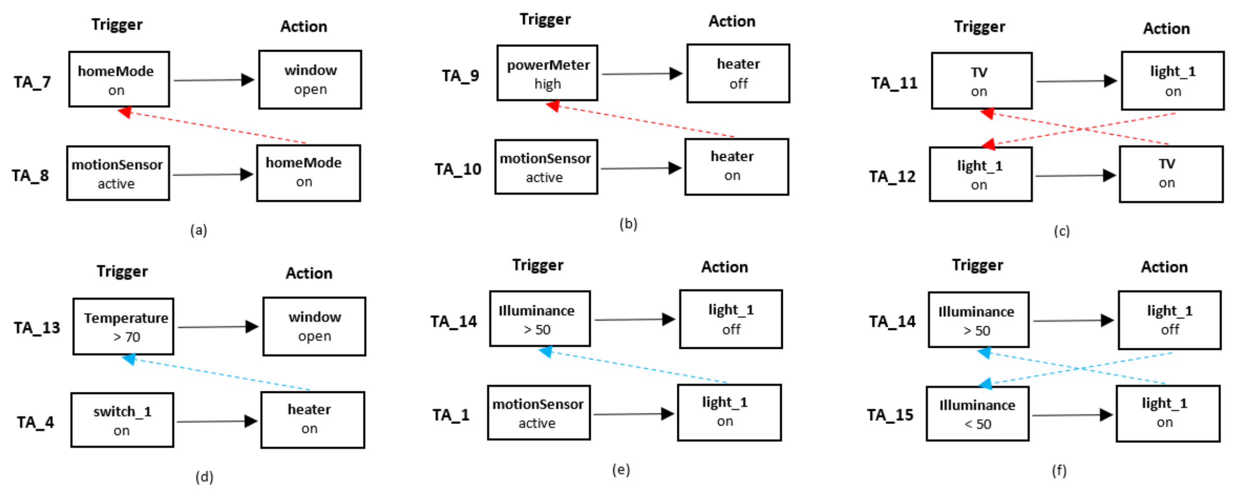


Figure 3.2: Examples of Chained Interactions. (a) & (d) one TA triggers another TA in electrical channel and physical channel, respectively; (b) & (e) *self-disabling*: one TA triggers another TA, and second TA executes opposite action in electrical channel and physical channel, respectively; (c) & (f) *loop-triggering*: two TAs are triggering each other in a loop in electrical channel and physical channel, respectively

is locked at the same time.

3.3.2 Chained Interactions

A chained interaction occurs when a device state is changed by the action command of one trigger-action trig another trigger-action. One trigger-action could directly trigger another trigger-action when the action device of the first trigger-action is the same as the trigger device of the second trigger-action. As shown in Figure 3.2(a), when **TA_8** is triggered when motion is detected and "*mode is set to 'home'*", which triggers the execution of **TA_7**, "*open the window when mode is 'home'*". In this example, the unexpected action "*open the window*" will be executed "*when motionSensor is active*". Similar to the conflict interactions, a chained interaction could also be indirectly caused when the action device of the first trigger-action and trigger device of the second trigger-action are acting under the same channel with the same action goal. For instance, Figure 3.2(d) shows that **TA_4** defines "*turn on the heater when switch_1 is on*", and **TA_13** defines "*open the window when the temperature is above 70F*". The action (*turn on the heater*) of **TA_4** acts under the temperature channel to increase the temperature, which is the same channel of the trigger of **TA_13** (*temperature above 70F*). In this case, **TA_4** will indirectly trigger **TA_13** after a while, and the window will be open when swith_1 is on. These chained interactions lead to unintended trigger-actions that are not desired by the user. Adversaries may leverage these chained interactions to indirectly send action commands to target devices with a malicious app.

Self-disabling and *loop-triggering* are two special cases of chained interactions [15]. The self-disabling happens when two trigger-actions are operated on the same action device with conflict action commands, and the action caused by the first trigger-action directly/indirectly triggers the second trigger-action. Figure 3.2(b) shows an example of self-disabling where the heater is turned on when motion is detected in **TA_10**, and the state of the power meter will change to high, which triggers **TA_9** to "*turn off the heater*" immediately after the heater is turned on. The loop-triggering occurs when two trigger-actions are both triggering each other with the same trigger and action, as shown in Figure 3.2(c). If **TA_12** "*turn on the light_1*" makes the TV state change to "on", **TA_11** will be triggered to "*turn on the light_1*" again, which then triggers **TA_12**. As a result, these two trigger-actions are run into a forever loop which makes the app keep sending action commands. Figures 3.2(e) and 3.2(f) are shown the self-disabling and loop-triggering that happened indirectly under the physical channel, respectively.

Chapter 4

IoTDiscover Design

The major goals of our system is to present the user with any hidden app functionalities that differ from app descriptions and discover potential interaction threats between multiple apps and allow the user to select or specify policies to restrict unexpected actions. In order to achieve our goal, the system contains four major steps: (1) *App information collection*, (2) *Interaction discovery and resolution*, (3) *Policy generation*, (4) *Policy enforcement*.

4.1 App Information Collection

The app information collection consists of two modules: code analysis and code instrumentation. The code analysis is used to extract actual trigger-action behavior from the source code. The code instrumentation collects app information used for conflict discovery and policy enforcement by adding extra logic and functionalities.

4.1.1 Code Analysis

The code analysis is a popular approach to analyzing the source code of an IoT app. In our project, the main purpose of code analysis is to extract information such as the actual trigger-action behaviors from the source code. Our code analysis system leveraged a backward taint analysis approach that constructs all the method calling paths to a device action command call [12]. Before the backward taint analysis, the system will extract the intermediate representation (IR) from the source code of IoT apps. The IR is used to construct an app's entry points, event handlers, and call graphs, as well as model the lifecycle of an app by using backward taint analysis. With this approach, we can get an output that includes the information of the calling path to execute any action call in the app, which also includes the trigger information that trigs the event handler. Then we can process the information of calling paths to extract the trigger-action information such as trigger event and action device and command. In the end, the trigger-action will be formed in a feature list with a format as shown below:

Trigger-Action_ID:
Trigger: <device>, <attribute>
Action: <device>, <command>

4.1.2 Code Instrumentation

Our code instrumentation will inject extra logic and functionalities that collects three types of information: (1) the device and app configuration information as well as user-defined numerical values, such as the temperature threshold that turn on a heater, (2) the user's decision for actual trigger-action behaviors of an app, which extracted from code analysis; (3) the runtime trigger event and action information. The instrumented app transmits the configuration and decision information to the conflict discovery and resolution module to identify any potential conflict or interaction threats based on the user's configuration of actual trigger-action behavior. The runtime trigger event and action will be transmitted to the policy enforcement module once the app is triggered by an event and before the action is executed.

Collecting Configuration and Decision Information. The user is required to specify related information to run the app and control the target device in an IoT environment, such as device name/ID or numerical threshold values. Such configuration information is collected when the user installs or updates an app through an app's user interface. Once the user configured all request information, it is stored in a read-only map of an IoT app. Thus, the code instrumentation needs to insert logic to read this information from the read-only map and transmit it to the conflict discovery and resolution module.

Collecting Runtime Trigger Event and Action Information. A guard will be inserted to the source doe by code instrumentation to monitor and gather action information at run-time. The guard is set at each action command. Once the app is triggered to send a command to access the device, the guard will interrupt the action, transmit the related event/action information to the policy enforcement, and wait for the decision.

4.2 Conflicts Discovery and Resolution

As the problem statement described in section 3.2, a conflict could be categorized into single app conflicts and multi-app interaction threats. In this section, we discuss our approach to discovering such conflicts and provide policy-based solutions to each type of conflict.

4.2.1 Single-App Conflicts Discovery

We consider a single app conflict occurred when an app executes an unauthorized action that does not describe in the app description and is unaware of the user. To discover such conflicts, many existing approaches are using natural language processing (NLP) techniques to extract trigger-action information from the app's free-text description and compare it with the actual trigger-action extracted from the source code [24]. However, due to the complexity of description language semantics, it is challenging for NLP to extract sufficient information precisely. Thus, we will let the user manually discover and decide whether the actual behavior of an app is in conflict or not. The actual trigger-action will be demonstrated to the user with easy understanding sentence, and the user can compare it will app descriptions to identify whether the behavior is different from the description based on their understanding and expectations.

4.2.2 Interaction Threats Discovery

The interaction discovery follows the interaction threats examples we described in section 3.3. We assume two trigger-actions installed in the same environment, and each denoted as $TA_1 = (T_1, A_1, CT_1, CA_1)$ and $TA_2 = (T_2, A_2, CT_2, CA_2)$, where T_i, A_i, CT_i, CA_i are trigger, action, phys-

Category	Basic Pattern	Auxiliary Pattern	ID	Examples
Conflicting Interactions	$A_1 = \neg A_2$	$T_1 = T_2$	**A.1**	Fig. 3.1(a)
		$T_1 \neq T_2$	**A.2**	Fig. 3.1(b)
	$A_1 = A_2$	$T_1 = \neg T_2$	**A.3**	Fig. 3.1(d)
	$A_1 \neq A_2, CA_1 = \neg CA_2$	$T_1 = T_2$	**A.4**	Fig. 3.1(c)
Chained Interactions	$A_1 \rightarrow T_2$	$\sim (A_2 \rightarrow T_1), A_1 \neq A_2$	**T.1**	Fig. 3.2(a)
		$\sim (A_2 \rightarrow T_1), A_1 = \neg A_2$	**T.2**	Fig. 3.2(b)
		$A_2 \rightarrow T_1, A_1 \neq A_2$	**T.3**	Fig. 3.2(c)
	$A_1 \nrightarrow T_2, CA_1 = CT_2$	$\sim (A_2 \nrightarrow T_1), A_1 \neq A_2$	**T.4**	Fig. 3.2(d)
		$\sim (A_2 \nrightarrow T_1), A_1 = \neg A_2$	**T.5**	Fig. 3.2(e)
		$A_2 \nrightarrow T_1, CA_2 = CT_1, A_1 = \neg A_2$	**T.6**	Fig. 3.2(f)

Table 4.1: Interaction Threats Definition. Let $TA_i = (T_i, A_i, CT_i, CA_i)$, *i = 1, 2* denote two arbitrary trigger-actions, where $T_i, A_i, CT_i, CA_i)$ are trigger, action, trigger channel, action channel, respectively. $=$ denotes "same device and command/attribute"; $= \neg$ denotes "conflict with"; $\neq$ denotes "different devices and commands/attributes; $\sim$ denotes "negation"; $\rightarrow$ denotes "triggers directly (electronic channel); $\nrightarrow$ denotes "triggers indirectly (physical channel)

ical channel of the trigger, and physical channel of the action of TA_i, respectively. We use the operator "$=$" to denote the trigger/action of TA_1 and TA_2 are operated on the same device with the same attributes/commands; "$= \neg$" denotes two trigger-actions are operated on the same device but with conflict attributes/commands; and "$\neq$" denotes the trigger/action of two trigger-actions are operated on different devices. Based on the interaction threats examples and referring to the existing work of CAI inference threats definition [15], we define an interaction threat occurs when the trigger/action of TA_1 conflict or interplay with the trigger/action of TA_2 with some explicit and inexplicit interaction channels. In this section, we will formalize the two types of interaction inferences: *conflicting interactions* and *chained interactions*. Table 4.1 shows the summary of the interaction threats we have defined.

Conflicting Interactions. Based on the effects on actions of TA_1, we define three types of basic patterns. The first basic pattern has been defined by CAI threats, where two trigger-actions operate on the same action device but perform conflicting commands simultaneously in the electronic channel ($A_1 = \neg A_2$). In this type, the actions could be triggered by the same trigger event ($T_1 = T_2$, as the example in Figure 3.1(a)), or could be triggered by different trigger events that happened at the same time ($T_1 \neq T_2$, as the example in Figure 3.1(b)). We defined the second basic pattern where the two rules are control two different action devices but perform conflicting commands under the same physical channel ($A_1 \neq A_2, CA_1 = \neg CA_2$). The third pattern we defined it as ineffective triggers where the same action of two rules ($A_1 = A_2$) are triggered by the opposite trigger events ($T_1 = \neg T_2$).

Chained Interactions. In CAI threats, a chained interaction is defined when a device state is changed by an action command of one trigger-action TA_1 triggers another trigger-action TA_2, and TA_2 will be executed after TA_1. In this case, new implicit rule is defined but may not be desired by the user. We consider the chained interaction are happened in electronic channel when TA_1 directly triggers TA_2 based on electronic state changes of a device ($A_1 \rightarrow T_2$), and we consider the chained interactions are happened in physical channel when TA_1 indirectly triggers TA_2 based on the physical environment changes where the action of TA_1 and the trigger of TA_2 are acting under the same physical channel ($A_2 \nrightarrow T_1$ *&* $CA_2 = CT_1$), such as temperature. The definition of CAI threats only defined electronic channel interactions, and we extended two physical channel interactions based on their definition. **T.1** & **T.4** defines the common chained interactions where the action of TA_2 does not have conflict or interaction with the action of TA_1 ($A_1 \neq A_2$). *Self-disabling* and *loop-triggering* are the special cases of the chained interactions. Self-disabling (**T.2** & **T.5**) happened when two rules perform conflicting commands ($A_1 = \neg A_2$) while TA_1 triggers TA_2. Looping-triggering (**T.3** & **T.6**) occurred when the action of TA_1 directly or indirectly triggers TA_2 ($A_1 \rightarrow T_2$ or $A_1 \nrightarrow T_2$ *&* $CA_1 = CT_2$), then the action of TA_2 triggers TA_1 after that ($A_2 \rightarrow T_1$ or $A_2 \nrightarrow T_1$ *&* $CA_2 = CT_1$).

Discover Interactions. To discover the interactions that we described above, we consider a tuple (trigger *t*, action *a*, channel of trigger *ct*, channel of action *ca*) to represent one trigger-action behavior extracted from the code analysis, and each trigger/action is paired with the configuration information collected by instrumented code (such as device ID and user-defined threshold value). Algorithm 1 and Algorithm 2 present the procedure of discovering conflicting interactions and chained interactions, respectively. Let *TA* denote sets of trigger-action behaviors with configured information. The two algorithms take *TA* as inputs and compare each two trigger-actions based

Algorithm 1 Interaction Discovery - Conflict Interactions

```
1:  Input: $TA_P$, sets of trigger-actions with configured information, {t, a, ct, ca}
2:  Output: $IA$, sets of discovered conflict interactions
3:
4:  for $i \in TA_P$ do
5:      for $j \in TA_P$ do
6:          if $i == j$ then
7:              continue
8:          if $i.a ==\sim j.a$ then
9:              if $i.t == j.t \,||\, i.t! = j.t$ then
10:                 $IA \leftarrow \{i, j\}$
11:         if $i.a == j.a$ then
12:             if $i.t ==\sim j.t$ then
13:                 $IA \leftarrow \{i, j\}$
14:         if $i.a! = j.a$ && $i.ca ==\sim j.ca$ then
15:             if $i.t == j.t$ then
16:                 $IA \leftarrow \{i, j\}$
```

on the effects of actions. The output of the algorithms would be a list that contains tuples of interaction pairs (e.g., [(t_1, a_1, ct_1, ca_1), (t_2, a_2, ct_2, ca_2)]). To the end, all potential interactions are discovered and will be presented to the user with recommended solutions, which we will discuss in the section 4.2.4.

4.2.3 Single-App Conflicts Resolution

Once the user discovers a conflict and decides to block the trigger-action, a solution will be generated by defining an access control policy to restrict the action. For example, when the user discovers a trigger-action "*unlock the door when motion is detected*" conflicts with the app description, to restrict the action, a policy "*Deny unlock the door if motion is detected*" is defined to block the action "*unlock the door*". The trigger-actions that are not discovered with any single app conflicts will then be used to identify any potential interaction threats.

Algorithm 2 Interaction Discovery - Chained Interactions

```
1: Input: $TA_P$, sets of trigger-actions with configured information, {t, a, ct, ca}
2:
3: Output: $IA$, sets of discovered conflict interactions
4:
5: for $i \in TA_P$ do
6:     for $j \in TA_P$ do
7:         if $i == j$ then
8:             continue
9:         if $i.a == j.t$ then
10:            if $j.a! = i.t$ then
11:                if $i.a! = j.a$ || $i.a ==\sim j.a$ then
12:                    $IA \leftarrow \{i, j\}$
13:            else if $j.a == i.t$ && $i.a! = j.a$ then
14:                $IA \leftarrow \{i, j\}$
15:        if $i.a! = j.t$ && $i.ca == j.ct$ then
16:            if $j.a! = i.t$ && $j.ca! = i.ct$ then
17:                if $i.a! = j.a$ || $i.a ==\sim j.a$ then
18:                    $IA \leftarrow \{i, j\}$
19:            else if $j.a! = i.t$ && $i.ca == i.ct$ && $i.a ==\sim j.a$ then
20:                $IA \leftarrow \{i, j\}$
```

4.2.4 Interaction Threats Resolution

User-in-the-loop Design. Based on each type of interaction threat, the IoTDiscover system will generate related solutions (e.g., access control policies) to restrict unexpected actions caused by the interaction. As different users have various requirements and expectations for their IoT system, which leads to a challenge in defining access control policies that satisfies all users' requirements. To address this, we include the user in the loop, where the system will first generate some solutions and present them with discovered interaction threats to the user as recommendations, then ask the user to select or specify what they think is most appropriate to their requirements. Once the user defines the appropriate solution, the system will learn from the user's choice and store the new solution defined by the user to make a better recommendation next time.

Resolution for conflicting interactions. The conflicting interaction happens when two conflict actions are executed simultaneously or conflict triggers trig the same device. The main idea to resolve such conflict is to avoid the two trigger-actions behaving at the same time. For interaction

ID	Conditions Description
C.1	Allow light to be turned on only if illuminance below 50 for 10 mins
C.2	Allow light to be turned on only if motion is detected
C.3	Allow light to be turned on only if user is at home
C.4	Allow light to be turned off only if motion is not detected within 30s
C.5	Allow window to be open only if user is at home
C.6	Allow heater to be turned on only if AC is off
C.7	Allow water valve to be turned on only if water leakage is detected with in 60s

Table 4.2: Part of Pre-defined Conditions. The conditions are defined by analyze app/device functionalities and adopted the idea of the safety and security properties of IoT devices from existing works [13] [14] [26]. Each condition will be loaded based on target action device and action command.

threats **A.1**, **A.3**, and **A.4**, the two trigger-actions are subscribed to the same trigger device, which makes the conflict action always executed at the same time (or a device ineffectively controlled by conflict triggers) unless only one trigger-action exist. Thus, we will let the user decide which trigger-action needs to be blocked and generate a policy based on the user's decision, for example, "*Deny* A_1 *if* T_1" when the user decide to block TA_1.

Resolution for chained interactions. The chained interaction happens when an unexpected action is executed under an unauthorized trigger. One solution is to define a policy to deny the unexpected action to be executed when the trigger is happening, for example: "*Deny* A_2 *if* T_1". However, there exist two cases that will make the policy inadaptable and ineffective: (1) the user may allow the chained interactions only if it is under a certain condition, (2) for the special case of loop-triggering, the trigger (action) of TA_1 are usually the same as action (trigger) of TA_2 ($T_1 = A_2$ and $A_1 = T_2$, respectively) (**T.3**), or the trigger device state for T_1 will be changed after A_1 is executed, (e.g., turn on the light will change the illuminance value read by illuminance sensor) (**T.6**).

To address this, we define conditions to only allow a target action to be executed under certain conditions. The conditions are pre-defined based on the use cases of the devices (e.g., a light could be turned on and off), and adopted ideas of safety and security properties of IoT devices from many existing works [13] [14] [26]. For example, condition **C.1** specifies, "*Allow light to be turned on only if illuminance below 50 for 10 mins*", which could avoid the frequent turning of the light on and off for the example in Figure 3.2(f). Table 4.2 shows the part of pre-defined conditions. Once a chained interaction threat is discovered, the system looks for a solution by scanning all the pre-defined conditions and loading the condition list based on the target action device type and commands. For example, the system is looking for a solution to restrict the action "*turn on the light_1*", where light_1 with a device type of light, then the conditions **C.1**, **C.2**, and **C.3** will be loaded. The loaded condition lists will be presented to the user as a recommendation, and the user could select one from existing lists or specify a new condition by following the format and idea of recommended conditions. Once the condition is selected and specified by the user, it will be combined with the target trigger-action to construct a trigger-condition-action based policy. For example, if a condition "*Allow heater to be turned on only if AC is off*" is specified to restrict trigger-action "*turn on the heater when motion detected*", then a new policy "*Allow heater to be turned on when motion detected only if AC is off*" is generated.

4.3 Policy Generator

We adopted the policy language design from the work of PatrIoT [26] to generate the resolution policies into formal policy language that can be enforced by policy enforcement module. In this section, we will introduce the policy language syntax and semantics that are applicable to our policy rules.

4.3.1 Policy Language Syntax

Each resolution policy will be extracted as a *policy block*, where each policy block starts with a **Policy** keyword followed by an identifier such as **Example_1**. The body of each policy block contains a permission statement and one or two condition statements, as shown below:

Policy <identifier>:
ALLOW/DENY <target_clause>
ONLY IF <condition_clause>

In the permission statement, the keyword **ALLOW** or **DENY** are used to identify the permission from user whether to allow or deny the action under a particular condition, respectively. The <target_clause> are the action information such as action_device and action_cmd that extracted from the rule element of policy rules. There are two optional condition statements are start with a keyword **ONLY IF** or **EXPECT**, where **ONLY IF** defined as the target action can only be allowed or denied under the certain condition, and **EXPECT** defined as the target action can only be allowed or denied with the exceptional case under a restriction condition and the **ONLY IF** statement can be disregarded in this case. Based on the policy rules in our work, the **EXPECT** statement will be rarely defined in a policy. The <condition_clause> contains the condition statement extracted from the condition element of policy rules, it can be either a temporal condition or a non-temporal condition. A temporal condition would be a condition where the event has been triggered in a time period, and it is extracted based on the condition event time from the policy rules. The temporal operators such as **SINCE**, **LASTLY**, and **ONCE** are used to identify the past state of a device and can optionally take a within keyword and an additional time interval $[l, r]$, where l and r denote the lower bound and upper bound of the existing time, to specify the condition event occurring time. For example, a condition state that when motion is detected within 30 seconds can be converted to policy language as $LASTLY(state(motion) = active)WITHIN[0, 30]$. A non-temporal condition means the target action would be triggered immediately based on current state of condition devices. Each <condition_clause> could have logical combinations of more than one conditions by using **AND**, **OR** logical operators. While users could define a single app policy rule without a condition statement, in this case, we define the <condition_clause> as the automation unit that only allow/deny certain IoT app to send a target command to control the target device, which restricted the actions caused by overprivileged problem. Below are shown two examples of policy language syntax that converted from policy rules.

Policy Example_1:
ALLOW action_command = on **AND** action_device = ligth1
ONLY IF **LASTLY**(state(motionSensor1) = active) **WITHIN** [0, 30]

Policy Example_2:
ALLOW action_command = unlock **AND** action_device = frontDoorLock
ONLY IF automation_unit = enhanced-auto-lock-door

4.3.2 Policy Language Semantics

The formal semantics of policy language will convert the policy statement that contains permission statement and condition statement of a given policy to a quantifier-free, first-order metric temporal logic (QF-MTL) formula. The there is no order constraints on the policy statements, but different policy statements are combined with a "*deny overrides allow*" conflict resolution mechanism. this means that an action can only be allowed when the trigger event satisfies all the policy statement. As each policy statement will be convert to equivalent QF-MTL formula, the metavariable φ_{action},$\varphi_{condition}$, and $\varphi_{exception}$ can be used to describe the corresponding OF-MTL formulas that captures information of target action (<target_clause>), only if condition, and exceptions of each policy statement. In this case, a allow statement can be explained as $\varphi_{action} \Rightarrow \varphi_{condition} \wedge \neg\varphi_{exception}$, and a deny statement can be explained as $\varphi_{action} \Rightarrow \neg\varphi_{condition} \vee \varphi_{exception}$. Our policy rule may miss the exception statement, and we consider the missing exception statement to be logical FALSE.

4.4 Policy Enforcement

The policies will be encoded into the policy enforcement module as different policy functions. Once the policy enforcement module receives the guarded trigger events and actions information from the instrumented app, the policy will be enforced by matching them with the trigger events and actions. If the trigger events and actions are compliant with all the policies, an "accept" decision is made and sent to the execution engine to execute the action; otherwise, a "deny" decision is made, and the execution engine will reject the action. As the policy enforcement module needs to receive every guarded event and action information from different IoT apps, it is required to preserve a global view of the entire system state and deployed at a place reachable by every guarded trigger events and actions of different apps.

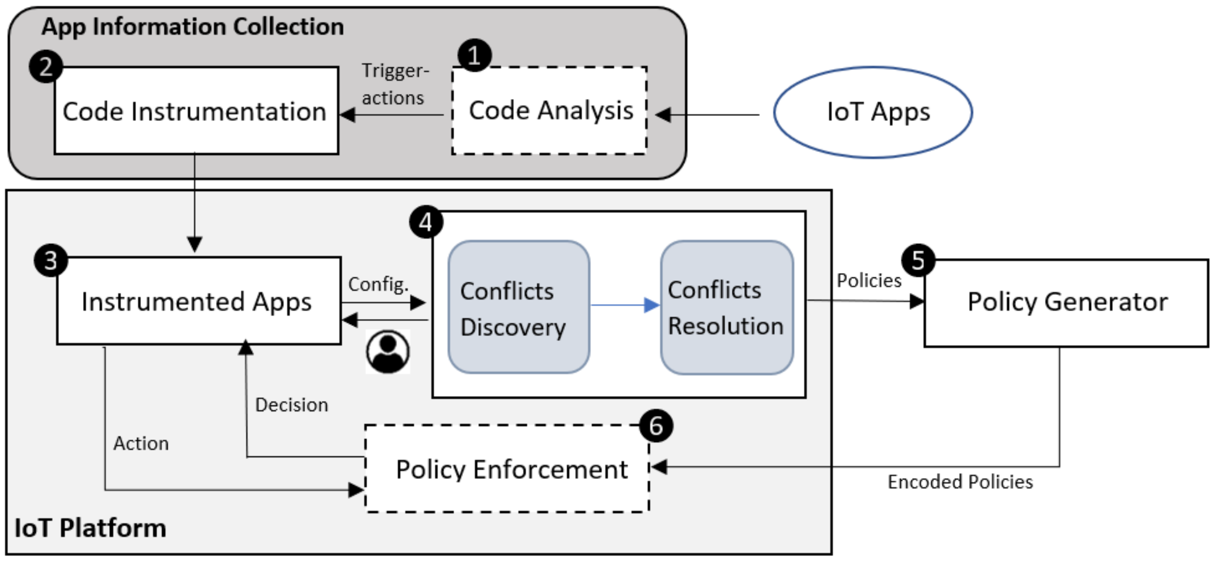


Figure 4.1: IoTDiscover Workflow.

4.5 Summary of IoTDiscover

In this section, we are discussing the integration of all the modules and steps described above. Figure 4.1 shows the workflow of our IoTDiscover system.

A user could browse through the app store or third-party marketplace to select expected apps based on the app's descriptions and tries to install the apps to the IoT platform. Those apps could be installed from unvetted sources where malicious code or unintended functionality could be sneaked into the source code, which is not specified in the app descriptions. The code analysis module analyzes the function flow of the app's source code to extract the actual behavior with trigger events (e.g., temperature below 70F) and corresponding actions (e.g., turn on the heater) of the app (❶). The extracted trigger-action will be present to the user by using code instrumentation in order to gather the user's permission (e.g., allow/deny) for the actual behavior of the app. The code instrumentation adds extra logic to source code that collects app information, including user's settings/configurations of an app, user's permission of actual app behaviors, as well as guarded events/actions information at runtime (❷). The instrumented app will be installed on the platform, and the user sets up the configuration through the app's user interface (UI) and gives permissions by comparing extracted behaviors with app descriptions (❸).

When the instrumented apps are deployed, the user permitted trigger-actions are configured as new rules to control the devices in an app. The interaction discovery and resolution module takes the configured new rules to identify all possible interactions with existing rules and generate solutions for interactions that could be adopted by the user to specify expected policies (➍). Based on the interaction threats described in Chapter 3.3, the interaction resolutions provided three types of solutions to resolve the threats: deny conflicting actions and ineffective triggers, specify condition-based policies to restrict interaction chains, and deny external communication under certain trigger events. The discovered interactions and solutions are presented to the user and request the user's expectations by selecting or permitting given solutions. In the end, the interaction discovery and resolution module collect resolved rules and form them as property lists that will be converted into formal policy languages by a policy generator (➎).

Once the policy languages have been generated, they will be encoded into the policy enforcement module, which identifies policy violations by verifying the app's runtime event and actions against the defined policies (➏). The instrumented app guards each action as well as sends event and action information to the policy enforcement module. If an action violates a policy, the policy enforcement rejects the action; otherwise, the action will be allowed to execute.

Chapter 5

Implementation

We implement our system on the Samsung SmartThings platform by using the SmartThings web-based IDE. And some program processes such as code analysis, code instrumentation, and policy generator are implemented on a Raspberry Pi 4 module.

5.1 Code Analysis

To implement code analysis, we adopted an existing work from our group, provided by Zhaohui Wang [2]. The code analysis would directly work on the Abstract Syntax Tree (AST) representation of Groovy code. Specifically, since the SmartApps for SmartThings platform performs an action by using different method calls. To get all the necessary information to build IR, the AST-Transformation class is used to write a visitor to visit each AST node and extract all entry points, expressions, and method calls inside AST nodes. The code analysis will take the app's source code as input. As the intermediate result, the app's inputs, capabilities, and action list that identified at each entry point. In addition, the sink activity list is also extracted where the app generates external communication with a destination outside of the platform, such as sending SMS, sending the notification, and sending any http requests. We consider these activities could cause unexpected actions in privacy concerns as they may leak users' sensitive activities. To get the information for trigger-actions, we processed the action list and sink activity list to extract the features of trigger-action, including *trigger_device*, *trigger_attribute*, *action_device*, and *action_command*. As the sink activity is not sending any command to any devices, the action_device would be the same as the automation unit, also known as the app's name, and the *action_command* is set as the sink action where the sendSMS or httpRequest function is called. Blow is showing the example of extracted trigger-action information for the code in Listing 3.1.

rule32:
Trigger: <device: thebatterymo>, <attribute: battery>
Action: <device: battery monitor>, <command: sendSMS>

rule33:
Trigger: <device: themotionsensor>, <attribute: motion.inactive>
Action: <device: thebatterymo>, <command: unlock>

5.2 Code Instrumentation

IoT apps are instrumented by code instrumentation before they are installed into the SmartThings platform. Extra code logic will be inserted in order to gather target information. An example of instrumented code is shown in Listing 5.1. SmartThings platform allows a SmartApp to be programmed with a dynamic preference (dynamicPage) where the content of a page could be dynamically generated based on the inputs from previous sections or returned value of in-app function calls. To assist the user to distinguish potential actions that differ from app description, a dynamicPage preference is added to the source code (line 17-26 of Listing 5.1) to create a user interface that displays app descriptions and single app trigger-actions based on the user configured devices and settings. As the conflict discovery and resolution module need to receive configuration and decision information from different IoT apps, it is also needed to prevent a global view of the entire system (similar to the policy enforcement module). To achieve that, a Parent-Child relationship is structured between all SmartApps and the two modules, where the parent app could call any public function inside the child app, and the child app could also call any public function inside the parent app. The conflict discovery and resolution and policy enforcement modules are parent apps, and all instrumented apps are child apps. With this structure, the parent apps are able to get configuration and setting information by calling getChildAppDevices() function (line 13) that is injected by code instrumentation.

```
 1 definition( ...
 2    parent: "ruichenpolicy:PolicyManager",
 3    ...
 4 )
 5
 6 preferences {...}
 7 #Events
 8 subscribe(motion1, "motion.inactive", motionInactiveHandler)
 9
10 def motionInactiveHandler(evt) {
11    parent.verify(app.getLabel(), evt, switch1.getDisplayName(), 'off',
       null) == true ? switch1.off() : log.debug('Invariants violation!')
12 }
13 def getChildAppDevices() {
14    return settings
15 }
16 def SetupPage() {
17    dynamicPage(name:"SetupPage") {
18       #original input section...
19       section("App Description") {
20          #display app description
21       }
22       section("Single-App Policy:") {
23          #display extracted trigger-action
24          #request for permission Allow/Deny
25       }
26    }
27 }
```

Listing 5.1: Instrumented code example. Parent-Child relationship is defined on line 2; Guard is set to capture and send runtime information (line 11); *getChildAppDevices()* function is added to collect the user's configured information (line 13-15); and a dynamicPage is added to create user interface to collect the user's decisions. (line 17-26)

To set the guard to monitor an app's action behavior, the code instrumentation will parse the app's source code to find the actions (method calls) in the pre-compiled action list. Similar to code analysis, the code instrumentation also works on the AST of a SmartApp with AST visitors. Once an action command is visited, it is replaced with a ternary operator, which examines if the function call to the policy enforcement module is evaluated to be true in the condition part. In the end, the AST will be translated back to the source code as instrumented SmartApp.

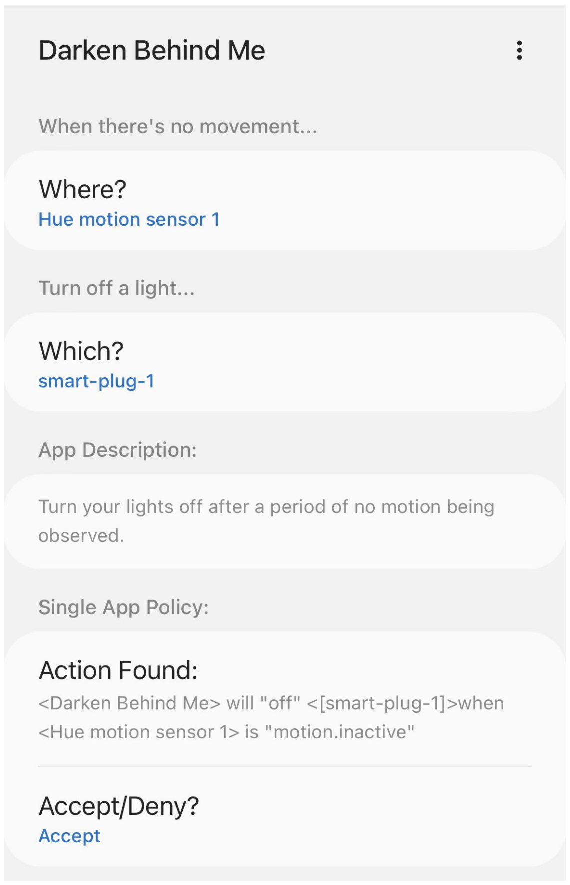
5.3 Conflicts Discovery and Resolution

Configure Trigger-Action List. To implement the conflict discovery and resolution module, a parent app is created during the code instrumentation process and encoded with trigger-actions extracted from code analysis. Since the trigger-action information extracted from code analysis is based on input variable names (e.g., thebatterymo, themotionsensor), where multiple apps could have the same input variable names, but the user can configure them with different devices. Thus, it is necessary to config these input variables with the actual device id or device name that the user selected to be used in each app. The discovery and resolution app first calls *getChildAppDevice()* function from child apps to get the settings and configuration information that contains all input values of certain SmartApp. The information returned by the function is formed as a list with a key of the input variable name and the value of the device name, for example: [“thebatterymo” : “myFrontDoorLock”, “themotionsensor” : “myMotionSensorDevice”]. Then the trigger-actions will be paired with the setting and configuration information as shown below for an example. And the configured trigger-actions can be used to identify any conflicts.

rule32:
Trigger: <device: myMotionSensorDevice>, <attribute: motion.inactive>
Action: <device: myFrontDoorLock>, <command: unlock>

5.3.1 Single-App Conflicts Discovery and Resolution

Discovery. We allow the user to identify whether the actual action of an app conflicts with the app’s description by providing a user interface, as shown in Figure 5.1. The app’s description and trigger-action with configured device name will be present to the user. An input box with *Accept* and *Deny* option is provided after the actual trigger-action display window to request the user to decide whether they expect to allow the trigger-action or not. The input variable name for the user’s decision is based on the trigger-action id. Once the user configures and installs the instrumented app, the decision will be stored and transmitted to its parent app in the form of [“trigger-action_ID” : “decision (Accept/Deny)”] with configuration information. **Resolution.** Once the


Darken Behind Me

When there's no movement...

Where?
Hue motion sensor 1

Turn off a light...

Which?
smart-plug-1

App Description:

Turn your lights off after a period of no motion being observed.

Single App Policy:

Action Found:
<Darken Behind Me> will "off" <[smart-plug-1]>when <Hue motion sensor 1> is "motion.inactive"

Accept/Deny?
Accept

Figure 5.1: The user interface for single-app conflicts discovery and configuration page. The app description and actual app's behavior is presented to user, and the user can compare and select whether they want to allow the behavior or not.

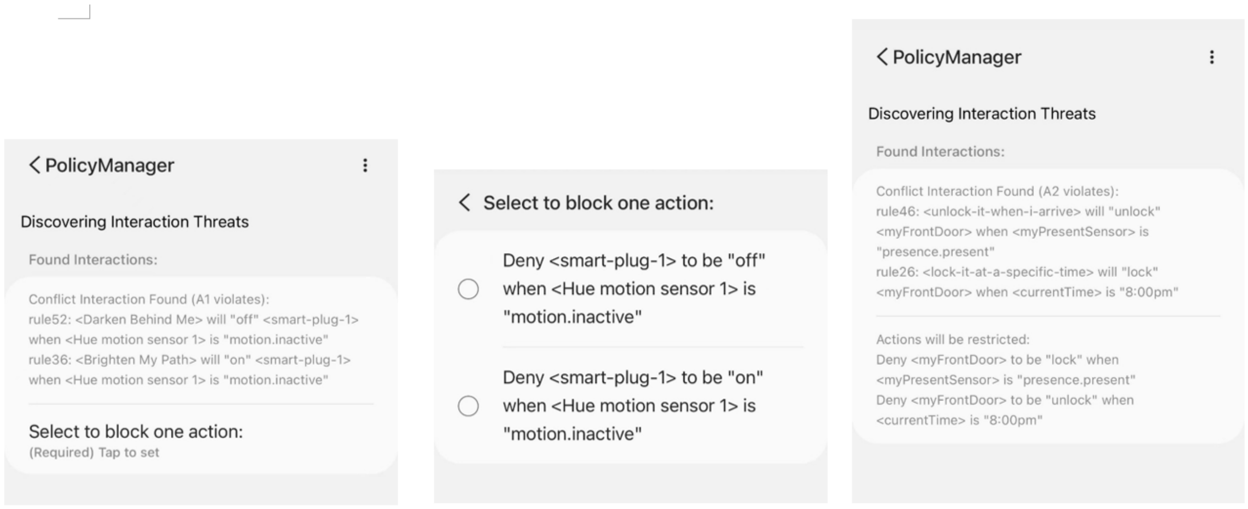
instrumented apps are installed, the decision with trigger-action_ID is sent to the conflict discovery and resolution module. The module then pairs the decision with each trigger-action based on trigger-action_ID. For each trigger-action with a "Deny" decision, it will be moved out from the trigger-action list and converted into a policy. The trigger-action with the "Allow" decision is not changed in the list and will then be used to discover any potential interaction threats between each other.

5.3.2 Interaction Threats Discovery and Resolution

Define Physical Channels. Before we compare each trigger-action to discovery interactions, we need to identify the action and trigger channel of each trigger-action based on the device and command/attribute. Some devices may not have physical effects when the device state is changed (e.g., motionSensor, TV, DoorLocker, etc.), then we define the channel of such devices with **NULL** value. For other devices that could cause physical effects or subscribe to physical environmental changes (e.g., temperatureSensor, light, heater, etc), we consider three common channels: temperature, humidity, and illuminance. We assign each device ID/name with related channels based on the device type; for example, the device "mylight_1" and "mylight_2" are two devices with "light" type and will be assigned under the illuminance channel. Each channel contains two features, either increasing or decreasing. The two features depend on the command/attributes of the device under a certain channel, for instance, a light to be turned on (or illuminance > 50) is under an increasing illuminance channel, and a light to be turned off (or illuminance < 50) is under decreasing illuminance channel. We denote the device action with an increasing feature as 1, and with decreasing feature as 0. Then the channel of a trigger/action will be denoted as [channelName, 0/1], and inserted into the trigger-action list in the forms of (t, a, ct, ca) as described in section 4.2.2.

Discovery. Based on Algorithm 1 and Algorithm 2, a Groovy code program is created and implemented in the SmartThings platform. The program inputs the trigger-action list with channels and filters out interaction threats by verifying whether TA_1 and TA_2 satisfies both the basic pattern and auxiliary pattern. As an example of detecting A.1 type of conflicting interaction, the program first verifies whether TA_1 and TA_2 are sharing the same action device and access conflicting command to the same device, if so, then $A_1 = \neg A_2$ holds. Then the program verifies whether the two trigger-action share the same trigger device and trigger attributes, if so, then $T_1 = T_2$ holds, and **A.1** interaction threats occur. Otherwise, if the two trigger-action are configured with different trigger devices, then $T_1 \neq T_2$ holds and **A.2** interaction threats occur. All discovered interaction threats will be marked with their type and stored in a list.

Resolution. For each discovered interaction threat, the system will generate related policies or search for related conditions based on interaction threats type. The loaded solutions are presented to the user with a description of detected interaction threats through a user interface on SmartThings mobile app. Each type of resolution to interaction threats would be performed as a dif-


< PolicyManager

Discovering Interaction Threats

Found Interactions:

Conflict Interaction Found (A1 violates):
rule52: <Darken Behind Me> will "off" <smart-plug-1> when <Hue motion sensor 1> is "motion.inactive"
rule36: <Brighten My Path> will "on" <smart-plug-1> when <Hue motion sensor 1> is "motion.inactive"

Select to block one action:
(Required) Tap to set

< Select to block one action:

Deny <smart-plug-1> to be "off" when <Hue motion sensor 1> is "motion.inactive"

Deny <smart-plug-1> to be "on" when <Hue motion sensor 1> is "motion.inactive"

< PolicyManager

Discovering Interaction Threats

Found Interactions:

Conflict Interaction Found (A2 violates):
rule46: <unlock-it-when-i-arrive> will "unlock" <myFrontDoor> when <myPresentSensor> is "presence.present"
rule26: <lock-it-at-a-specific-time> will "lock" <myFrontDoor> when <currentTime> is "8:00pm"

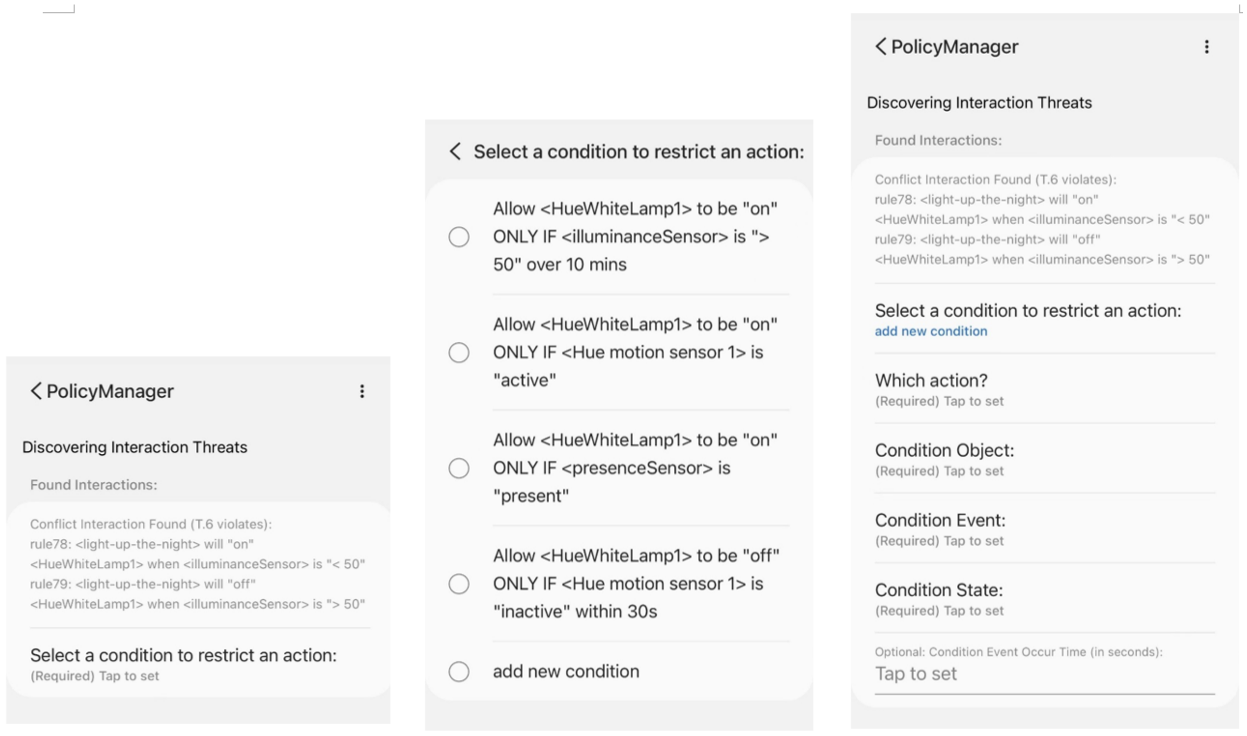
Actions will be restricted:
Deny <myFrontDoor> to be "lock" when <myPresentSensor> is "presence.present"
Deny <myFrontDoor> to be "unlock" when <currentTime> is "8:00pm"

Figure 5.2: The user interface to define resolution for conflicting interactions with user-in-the-loop design. Recommended policies will be generated based on types of conflicting interactions. The UI will display conflicts to the user and request the user to select expected policy.

ferent user interface design. For interaction threats **A.1**, **A.3**, and **A.4**, they can be solved with two options of resolutions, either block TA_1 or block TA_2. Then an input box will be provided to the user to select which trigger-action would be blocked, as shown in Figures 5.2(a) & (b). Since there is only one solution that could solve **A.2** type conflicting threats, the system will generate two policies to avoid the two trigger-actions acting at the same time and only present the policies to notify the user that new policies are created, as shown in Figure 5.2(c).

For chained interaction threats, two types of resolution will be presented, one is a “Deny” policy that will restrict an action triggered by an unauthorized trigger, and another is the list of pre-defined conditions based on the action device type and command. The user can select the “Deny” policy or any conditions to restrict their expectation, as shown in Figures 5.3(a) & (b). If the user is not satisfied with all recommended solutions, a “add new condition” selection would be selected, and the user can specify a new condition; the user interface is shown in Figure 5.3(c).

To demonstrate, we use an example and assume the user is trying to specify a condition “*Allow mylight_1 to be on only if myMotionSensor is active*”. The user is required to first identify which action he/she wants to restrict (*mylight_1.on* in this example). For each target action, three required



< PolicyManager

Discovering Interaction Threats

Found Interactions:

Conflict Interaction Found (T.6 violates):
rule78: <light-up-the-night> will "on" <HueWhiteLamp1> when <illuminanceSensor> is "< 50"
rule79: <light-up-the-night> will "off" <HueWhiteLamp1> when <illuminanceSensor> is "> 50"

Select a condition to restrict an action:
(Required) Tap to set

(a)

< Select a condition to restrict an action:

- Allow <HueWhiteLamp1> to be "on" ONLY IF <illuminanceSensor> is "> 50" over 10 mins
- Allow <HueWhiteLamp1> to be "on" ONLY IF <Hue motion sensor 1> is "active"
- Allow <HueWhiteLamp1> to be "on" ONLY IF <presenceSensor> is "present"
- Allow <HueWhiteLamp1> to be "off" ONLY IF <Hue motion sensor 1> is "inactive" within 30s
- add new condition

(b)

< PolicyManager

Discovering Interaction Threats

Found Interactions:

Conflict Interaction Found (T.6 violates):
rule78: <light-up-the-night> will "on" <HueWhiteLamp1> when <illuminanceSensor> is "< 50"
rule79: <light-up-the-night> will "off" <HueWhiteLamp1> when <illuminanceSensor> is "> 50"

Select a condition to restrict an action:
add new condition

Which action?
(Required) Tap to set

Condition Object:
(Required) Tap to set

Condition Event:
(Required) Tap to set

Condition State:
(Required) Tap to set

Optional: Condition Event Occur Time (in seconds):
Tap to set

(c)

Figure 5.3: The user interface to define resolution for chained interactions with user-in-the-loop design. Recommended conditions will be loaded based on target action device and commands. The UI will display chained interactions to the user and request the user to select expected condition. If the user selected to specify new condition, an UI is provided to the user to input following information: target action, condition object, condition event, condition state, and duration (optional).

input windows will be provided to the user to specify the conditions. The *condition_object* would be the condition device (*myMotionSensor*), and the user can select any device from all already installed devices in the same environment. The *condition_event* would be the state of the condition device, which includes all capability commands or attributes of the device (*active* and *inactive* for motion sensor), and the user can select the desired one (*active* in this example). There exist some devices that may be assigned with numerical attributes, such as temperatureSensor, illuminanceSensor, and humiditySensor. For these devices, the *condition_event* would be a text input box to allow the user to specify the operator $(=, >, <)$ and numerical thresholds. The *condition_state* represented the state of the condition event, which only has two choices: *is* and *is not*, (in this example, *is* is selected). An optional text input is also provided to allow users to define the duration conditions, for example, "*within 10 mins*", or "*over 10 mins*". When a new condition is specified

```
{"permission" : <ALLOW/DENY>,
    "trigger-action" : {"trigger" : [<device>, <attribute>],
                         "action" : [<device>, <command>]},
    "condition" : [<object>, <event>, <state>, <duration>]}
```

Listing 5.2: Resolution policy output format. **permission** defines whether ALLOW or DENY the target action; **trigger-action** contains device and attribute/command information of trigger and action; **condition** specifies the information of the user selected/specified conditions, *NULL* value will be returned if no condition add to the policy.

```
{"permission" : <ALLOW>,
    "trigger-action" : {"trigger" : ["motionSensor_1", "active"],
                         "action" : ["heater", "on"]},
    "condition" : ["AC", "off", "is", NULL]}
```

Listing 5.3: Resolution policy output example with configured information for policy "*Allow heater to be turned on when motion detected only if AC is off*"

by the user, it is uploaded to the condition list and presented as recommended conditions for the next related interaction threats. In the end, the selected or specified conditions will be combined with the target trigger-actions and form a new policy. All the policies adopted by the user will be stored in a policy list and transmitted to the policy generator module to generate policy languages that could be used for policy enforcement.

5.3.3 Resolution Policy Output

The resolution policy generated in the resolution module will be tokenized as abstracted features. This will allow the policy generator module to extract necessary information easily. As shown in Listing 5.2, each policy will be structured as a JSON list that contains three main features: (1) the **permission** would be Allow/Deny the target action to be executed, (2) the **trigger-action** is the target trigger action information that including trigger/action device, trigger/action attributes/-commands, and (3) the **condition** that the user select or specified to restrict the chained interaction threats, which could be **NULL** if there is no condition added to the policy. Listing 5.3 shows an example of the featured policy “*Allow heater to be turned on when motion detected only if AC is off*”. Once all policies are structured in the JSON list, the list will be sent to the policy generator module. Since the generation module is a program implemented outside of the SmartThings platform, the list will be sent through an external communication (sendSMS).

5.4 Policy Generator and Enforcement

Policy Generator. The policy generator is a Python script that runs on a Raspberry Pi 4 module. Once it receives the JSON list from the conflict resolution module, it extracts related information and generates a policy block based on policy language syntax. The keyword **Allow/Deny** is based on the feature "permission", and <target_clause> is filled with "action" with action device and action command, and <condition_clause> contains both "trigger" and "condition". An example of policy language generated for Listing 5.3 is shown below.

Policy P_1:
ALLOW action_command = on **AND** action_device = heater
ONLY IF state(motionSensor_1) = active **AND**
    state(AC) = off

Policy Enforcement. Once the defied policies are generated, the code instrumentation will process again and create a policy enforcement module that is encoded with policies. A parser generated by ANTLR [1] is used to parse the policy language syntax, and the policy semantics will be encoded as different policy functions in the enforcement module. To enforce the policies, a policy decision function is generated to make decisions on whether the action is satisfied with the given policies. The policy enforcement module runs as a parent app in the SmartThings platform in order to prevent the global view of the entire system. In this way, all child apps (instrumented apps) are allowed to call the decision function by passing the related event and action information as arguments and waiting for a response. The decision function then passes the arguments to all existing policy functions and verifies whether the action is violated or not. If the action passes all policy functions, a TRUE decision will be returned and allow the app to execute the action command; otherwise, a FALSE decision is returned, and the action is rejected.

Chapter 6

Evaluation

Testing Cases. Due to the limited number of IoT devices, we present five test cases by manually selecting a few numbers of SmartApp in each test case. A total of 19 SmartApps are selected, some apps may be used in multiple testing cases but with different configurations. The SmartApps are selected from the official marketplace, except two are installed from IoTBentch [5], which includes malicious code and function flaws. The two malicious SmartApps are installed in the first testing case (testing case A). We then run each testing case in IoTDiscover. The experiment is performed on a Raspberry Pi 4 Module, and the instrumented apps are implemented on the Samsung SmartThings Web IDE. 4 physical devices and 6 virtual devices are installed in the same room on SmartThings IDE and the numerical state for virtual devices (e.g., temperature value) is monitored through SmartThings simulator. We also installed the SmartThings mobile app to control and monitor the status of the devices. The overall evaluation result is shown in Table 6.1.

Conflict Detection Result. To evaluate the correctness of the discovery result, we manually go over the source code of each app and record all the trigger-actions that occurred in the app's code. With the recorded trigger-action and configuration information, we analyze single-app conflicts and interaction threats by comparing the trigger-actions. We also run these apps in each testing case to verify the correctness of our manual analysis. After executing all five testing cases, each testing case resulted in different types of conflicts, and one app in a testing case may cause multiple conflicts. 9 types of interaction threats and some single-app conflicts are detected, except **A.1** type of conflicting interaction. That is because we could not find corresponding apps from current resources to form **A.1** type conflict. Thus, to evaluate the effective of our system in this type of conflict, we created one app with simple functionality to "*turn on the light when motion is NOT*

Testing Case	**# of apps**	**# of Single-App Conflicts Detected**	**Interaction Threats Detected**	**# of Policies Generated**
A	6	2	**T.5, T.6**	4
B	5	1	**A.2, T.4**	3
C	5	0	**T.1, A.3**	2
D	4	2	**T.3**	3
E	6	1	**A.4, T.2**	2

Table 6.1: Result summary of evaluating IoTDiscover. All existing conflicts are detected, and the system generated corresponding policies

detected" and test it with an existing app **Brighten-my-path**, which have a functionality to "*turn on the light when motion is detected*". As a result, **A.1** conflict is successfully detected, and related policy is also generated. A summary of all interaction threats and selected single app conflicts is presented in Table 6.2.

Conflict Resolution Result. We use testing case A as an example to demonstrate the conflict resolution results. 6 SmartApps are included in testing case A, and two of them are malicious apps. In this testing case, 2 single-app conflicts and two interaction threats (**T.5, T.6**) are detected. With the 2 single-app conflicts, we selected to block the related trigger-actions, and then 2 "Deny" policies were generated. For interaction threat types **T.5** & **T.6**, 8 recommended conditions are presented for each $light$ actions (4 for $light.on$ and 4 for $light.off$). By manually reviewing the pre-defined condition list, we verified that all the condition lists with related action devices and commands are correctly presented to the user as recommendations. In the end, we selected two conditions for each interaction threat, and a total of 4 policies were generated as a JSON list. The description of the policy results is shown in Table 6.3.

Policy Enforcement Result. To evaluate the policy enforcement, we first execute the apps with violated actions before the policy is executed and recode the app and devices' actions. Then we execute the apps again with violated actions after the policy is enforced. To demonstrate, we use the examples of testing case A. In testing case A, the app Battery Monitor contains a hidden trigger-action "*unlock the door when motion is inactive*", to block this unexpected action, a policy "*Deny door to be unlocked if motion is inactive*" is encoded into the policy enforcement. When we execute the Battery Monitor app after the policy is enforced, we noticed that the action "*unlock the door*" is blocked. Similar to the interaction threats of **T.5**, the device *light*_1 will be immediately turned on after it is turned off when no motion is detected. After a policy "*Allow light_1 to be on only if illuminance is blow 50 for 1 min*", which makes the light to be turned on after the light is turned off within 1 minute. Our analysis of all apps' behavior after the policy is enforced shows that the system correctly enforced all the policy violations in each testing case, including the violations in a single app and the interaction threats. Table 6.4 shows the tested triggers and observed actions before and after the policy enforcement.

Testing Case	App Name	Threats	Threat Type
A	Darken-behind-me	**Unknown trigger-action:** Turn off light_1 when motionSensor is active	Single-app
	Battery-Monitor	**Unknown trigger-action:** unlock myFrontDoor when motionSensor is inactive	Single-app
	Darken-behind-me Light-up-the-night	Turn off the light_1 when motionSensor is inactive Turn on the light_1 when illuminance below 50	**T.5**
	Light-up-the-night	Turn off the light_1 when illuminance exceeds 50 Turn on the light_1 when illuminance below 50	**T.6**
B	Lock-it-at-a-specific-time Unlock-it-when-i-arrive	Lock myFrontDoor when currentTime is 9:00pm Unlock myFrontDoor when presentSensor is present	**A.2**
	Humidity-alert Curling-iron	Turn on humidifier when humidity between 30% Turn on the heater when motionSensor is active	**T.4**
C	Curling-iron	Turn on heater when motionSensor is active Turn on fan when motionSensor is active	**A.3**
	Make-it-so Change-mode-on-unlock	Open the window when mode is "home" Change mode to "home" when door is unlocked	**T.1**
D	Make-it-so Switch-change-mode	Turn on light_1 when mode is "home" Change mode to "home" when light is on	**T.3**
E	Close-the-valve Dry-the-wetspot	Turn off the valve when waterSensor is wet Turn off the valve when waterSensor is dry (user misconfigured)	**A.4**
	Energy-saver Its-too-cold	Turn off the heater when powerMeter exceeds 3000W Turn on the heater when temperature below 70F	**T.2**
	Created-app Brighten-my-path	Trun on the light_1 when motion is NOT detected Turn on the light_1 when motion is detected	**A.1**

Table 6.2: Detected threats, including selected single-app conflicts and cross-app interaction threats

ID	Policy
P.1	Deny light_1 to be off if motionSensor is active
P.2	Deny unlock myFrontDoor if motionSensor is inactive
P.3	Allow turn on the light_1 only if illuminance below 50 over 1 min
P.4	Allow turn off the light_1 only if illuminance exceeds 50 over 1 min

Table 6.3: Resolution policies generated for testing case A, where **P.1** is to resolve single-app conflict in *Darken-behind-me*, **P.2** is to resolve single-app conflict in *Battery-Monitor*, **P.3** is use to resolve interaction threats **T.5**, and **P.4** is use to resolve interaction threats **T.6**

Triggers	**Action Before Policy Enforcement**	**Action After Policy Enforcement**
motion.active	light_1 on then off immediately	light_1 on only
motion.inactive	light_1 off, myFrontDoor unlocked	light_1 off only
set illuminance to 30	light_1 on then off immediately	light_1 on then off after 1 mins
set illuminance to 60	light_1 off then on immediately	light_1 off then on after 1 mins

Table 6.4: Policy Enforcement Result for testing case A. By comparing the action before and after policy enforcement, our result shows IoTDiscover successfully blocked all the violate actions.

Chapter 7

Discussion and Future Works

In this section, we discuss the limitations of our system and future works that could be done on this project.

User-in-the-loop Design. We considered the user-in-the-loop in our system design to allow users to select the best solution to restrict the conflicts. However, in our current recommendation system, the user's option and new conditions are only stored in the condition lists. When a new interaction threat is discovered, the system lists all the condition list that matches the device type and command, which will present a tedious list as the user specifies more conditions. To address this problem, future work could adapt machine learning and provide a model training that learns from the information about the user's choice and expectation in the conflict resolution, and when a new interaction threat is fit in the model, it will automatically recommend the most popular or commonly used conflict resolution.

System Automation. After the resolution policies are generated in the conflict discovery and resolution module, the policies will be sent as SMS message to the policy generator module that is outside of the SmartThings platform. However, the SMS message needs to be manually extracted by copying the SMS message and past to the Python script of policy generator. In addition, as one child app cannot have multiple parent apps, and there are two parent apps created in our system, so the user is required to uninstall the first parent app (conflict discovery and resolution) and then is able to install the second parent app (policy enforcement). These steps make every process to be manually and complicated. A mitigation solution to automate the processes could be enable the external communication between the module inside SmartThings and the module outside the SmartThings through a REST API server. With this solution, the resolution policies could be sent by using an http request, and the policy enforcement module could be executed outside the SmartThings platform, where the instrumented app sends http request with guarded information and wait for the response from the policy enforcement module. We consider these improvements as future works.

Policy verification. Our recommendation system allows the user to select and specify any policy or conditions without verifying them. We assume all the policies generated by the system and the user are validated with no conflicts. However, in a highly complex IoT environment, the user-specified policies may conflict with other policies, or the policy condition could interact with other existing trigger-actions and fail to block unsafe and insecure states. For example, when the user defined a policy "*Allow light to be on when motion is active only if daytime is 9:00pm*", and later a new policy is defined as "*Deny light to be on when motion is active*", where the two policies are conflicted. To address this issue, the policy verification system is needed to verify if the policy is valid or not.

Network-based policy enforcement. In this thesis, we only focused on monitoring the unexpected behavior of an IoT app at the app level. However, many IoT platforms, an example as the SmartThings platform, allow the app to generate communications to external destinations. These communications are unknown to the user and could communicate to a malicious destination. In this case, the app could bypass the app-level policies and gain access to a device through the network level with unexpected behaviors. In addition, many existing researches have been shown that many attacks could be applied at the network level of an IoT environment, such as DDoS attacks. Recent efforts are focused on solving such problems by using Manufacturer Usage Description (MUD), which is an Internet Engineering Task Force (IETF) standard that defines the device profile or MUD file provided by the manufacturer and describes the intended network behavior of an IoT device in detail. By using the Software Defined Network (SDN), the MUD file can be translated into network access policies and enforced in the network. While most works are mainly focused on network-level access controls, the intersection of network level and app level access control could become a future work.

Chapter 8

Conclusion

Due to the increase in the development of IoT devices and IoT apps, the IoT system advanced our living environment to be more convenient, efficient, and autonomous, as well as made the IoT environment to be more complex, which may cause IoT security and privacy problems. Since users usually install an IoT app from unvetted vendors and most IoT platforms are lack access control mechanisms, the unexpected behavior of an IoT app may have occurred in two causes: (1) users are known limited information about an app through the app's description, a malicious action could hide in the source code and execute without user's authorization, (2) even benign apps could unexpectedly interact with each other. While existing efforts to address such threats are either only focused on single app conflicts or based on enforcing pre-defined policies where the user does not know the sources and functionalities. In this thesis, we presented a system, IoTDiscover, that discovers both single-app conflicts and multi-app interaction threats. For every conflict discovered by IoTDiscover, it generates resolution policies to restrict unexpected actions based on the user's choice. A dynamic policy enforcement module is also presented to enforce resolution policies against violated actions. We have evaluated our system with 17 official SmartApps and two flawed/malicious apps in 5 testing cases. As a result, IoTDiscover can detect all conflicts in each testing case and block all the violations by enforcing the resolution policies.

References

[1] Antlr. https://www.antlr.org.

[2] Codeanalysis. https://github.com/cyoki/IoTPrivacy.

[3] Homeasistant. https://www.home-assistant.io/.

[4] Homekit. https://developer.apple.com/homekit/.

[5] Iotbentch. https://github.com/IoTBench/IoTBench-test-suite.

[6] Openhab. https://github.com/openhab/openhab1-addons/wiki.

[7] Smartthings. https://www.smartthings.com/.

[8] Smartthings developer. https://developer-preview.smartthings.com/docs/devices/hub-connected/legacy/.

[9] Smart home device household penetration in the united states in 2019 and 2021. https://www.statista.com/statistics/1247351/smart-home-device-us-household-penetration/, 2022.

[10] AL-FUQAHA, A., GUIZANI, M., MOHAMMADI, M., ALEDHARI, M., AND AYYASH, M. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials 17*, 4 (2015), 2347–2376.

[11] ALHANAHNAH, M., STEVENS, C., AND BAGHERI, H. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis* (2020), pp. 272–285.

[12] Celik, Z. B., Babun, L., Sikder, A. K., Aksu, H., Tan, G., McDaniel, P., and Uluagac, A. S. Sensitive information tracking in commodity {IoT}. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 1687–1704.

[13] Celik, Z. B., McDaniel, P., and Tan, G. Soteria: Automated {IoT} safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 147–158.

[14] Celik, Z. B., Tan, G., and McDaniel, P. D. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *NDSS* (2019).

[15] Chi, H., Zeng, Q., Du, X., and Yu, J. 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)* (2020), IEEE, pp. 411–423.

[16] Ding, W., and Hu, H. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 832–846.

[17] Ding, W., Hu, H., and Cheng, L. Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery. In *the 28th Network and Distributed System Security Symposium (NDSS 2021)* (2021).

[18] Fernandes, E., Jung, J., and Prakash, A. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 636–654.

[19] Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. {FlowFence}: Practical data protection for emerging {IoT} application frameworks. In *25th USENIX security symposium (USENIX Security 16)* (2016), pp. 531–548.

[20] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., PRAKASH, A., AND UNVIERSITY, S. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS* (2017), vol. 2, San Diego, pp. 2–2.

[21] KOLIAS, C., KAMBOURAKIS, G., STAVROU, A., AND VOAS, J. Ddos in the iot: Mirai and other botnets. *Computer 50*, 7 (2017), 80–84.

[22] NGUYEN, D. T., SONG, C., QIAN, Z., KRISHNAMURTHY, S. V., COLBERT, E. J., AND MCDANIEL, P. Iotsan: Fortifying the safety of iot systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies* (2018), pp. 191–203.

[23] STEWARD, J. The ultimate list of internet of things statistics for 2022. `https://findstack.com/internet-of-things-statistics/#:~:text=Reports%20indicate%20that%20there%20will,and%2075.44%20billion%20by%202025`, 2022.

[24] TIAN, Y., ZHANG, N., LIN, Y.-H., WANG, X., UR, B., GUO, X., AND TAGUE, P. {SmartAuth}:{User-Centered} authorization for the internet of things. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 361–378.

[25] WONG, H., AND LUO, T. Man-in-the-middle attacks on mqtt-based iot using bert based adversarial message generation. In *KDD 2020 AIoT Workshop* (2020).

[26] YAHYAZADEH, M., HUSSAIN, S. R., HOQUE, E., AND CHOWDHURY, O. Patriot: Policy assisted resilient programmable iot system. In *International Conference on Runtime Verification* (2020), Springer, pp. 151–171.

[27] YAHYAZADEH, M., PODDER, P., HOQUE, E., AND CHOWDHURY, O. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proceedings of the 24th ACM symposium on access control models and technologies* (2019), pp. 61–72.

[28] ZHANG, W., MENG, Y., LIU, Y., ZHANG, X., ZHANG, Y., AND ZHU, H. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1074–1088.

[29] ZHANG, W., MENG, Y., LIU, Y., ZHANG, X., ZHANG, Y., AND ZHU, H. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 1074–1088.