PrivacyGuard: Exploring Hidden Cross-App Privacy Leakage Threats In IoT Apps

Zhaohui Wang
The University of Kansas
zhwang@ku.edu

Bo Luo
The University of Kansas
bluo@ku.edu

Fengjun Li
The University of Kansas
fli@ku.edu

Abstract

The increasing use of the Internet of Things (IoT) technology has made our lives convenient, however, it also poses new security and privacy threats. In this work, we study a new type of privacy threat enabled by cross-app chains built among multiple seemingly benign IoT apps. We find that interactions among apps could leak privacy-sensitive information, e.g., users' identification, location and tracking, activity patterns, etc. To tackle this challenge, we introduce PrivacyGuard, which extracts cross-app chains in the form of trigger-condition-action rules and identifies the corresponding privacy leakage risk with an inference probability. PrivacyGuard supports a fine-grained categorization of privacy threats to generate detailed alerts about privacy leakages. We evaluated PrivacyGuard on a dataset with 2,101 SmartApps, 2,788 IFTTT rules, and 2,086 OpenHAB rules, respectively. The results show that PrivacyGuard could uncover hidden privacy leaks that existing studies fail to detect. For example, 7.67% chains constructed by two seemingly benign IoT apps could leak at least one type of privacy information, while over 80% of the leaks involved privacy information regarding *Localization & Tracking* and *Activity Profiling*.

Keywords

IoT security and privacy, privacy leakage, IoT app privacy

1 Introduction

IoT cloud platforms such as Samsung's SmartThings [60], IFTTT [42], Apple's HomeKit [5], Google Home [33], and OpenHAB [56] have been utilized to handle diverse types of devices from different vendors and facilitate their interactions by through automation rules. While embracing the connectivity and convenience introduced by IoT devices and applications, there are increasing concerns regarding security and privacy risks [8, 14, 15, 31, 44]. A tremendous amount of IoT devices have been deployed in our everyday life, collecting various types of personal attributes (e.g., age, height, weight, address, location, etc.) and highly sensitive information about the users (e.g., blood pressure, heart rate, pin code of door lock, door state, etc.). Many IoT devices continuously sense diverse data types in the smart environment and stream them to external servers, where the fine-grained data could be used to profile user behaviors. For example, inferential profiling is used to develop insight into a user's health status and movement patterns from data sensed by Fitbit [28] or occupants' location and state from smart thermostats [29]. However, recent surveys [82] reported that many users were unaware of the privacy risks in home automation and trusted IoT manufacturers to protect their privacy.

Sensitive data leaks lead to potential privacy invasion. Previous studies showed that such leaks could be caused by malicious IoT applications [8, 14, 15], unauthorized access through IoT frameworks [31, 44], exploitation of customized automation rules [39, 51, 66], or app-level traffic analysis [1, 50]. For example, a study on the SmartThings framework identified multiple privacy threats due to malicious SmartApps stealing sensitive user data [14], where 138 out of 230 SmartApps were found exposing at least one sensitive data via the Internet or messaging services. However, its focus on single IoT apps makes it impossible to uncover potential privacy leakage risks due to interactions enabled by automation rules between multiple apps.

In this paper, we extend the scope from single-app to cross-app privacy threats by considering privacy leakage that can be inferred from cross-app interactions enabled by user-installed automation rules. Such rule-based automation is supported by many IoT platforms, e.g., SmartThings, IFTTT, OpenHAB, and HomeKit, using the *trigger-condition-action* (TCA) paradigm where an application contains at least one trigger-action path that can be executed when a pre-set condition is met. In smart homes with multiple IoT apps, one malicious app could manipulate another app's execution state by activating its trigger events or satisfying its rule conditions. This can be achieved *directly* or *indirectly* by manipulating action events or physical environment channels shared between two apps. In these platforms, the adversary could deliberately plant two malicious apps, one with direct access to private data (e.g., triggered by an event associated with private data) but no external-facing interfaces while the other with external interfaces but no access to any private data. As shown in Section 2.2, both apps are seemingly benign to bypass existing single-app privacy leakage detection. Once installed in a smart home, they form cross-app chains between themselves or with other benign apps to disclose users' private data to the adversary. We refer to it as *cross-app privacy leakage*.

Cross-rule privacy leakage was first observed on trigger-action platforms when interacting with IFTTT (if-this-then-that) rules [42]. Existing studies [22, 39, 51, 66] often take an information-flow approach, where a privacy leakage is defined as any data flowing from a higher security level (e.g., *private world*) to a lower security level (e.g., *public world*). For example, [66] would report a *private* → *public* confidentiality violation for an IFTTT applet *if you take a new photo, post it to Twitter*, because a personal photo (private data) would be disclosed to a Twitter post (public access) via this rule. However, these approaches provide little information about data leakage types (e.g., location, health, personal, etc.), which is useful for assessing the leakage severity. Besides, they mainly focus on the

Proceedings on Privacy Enhancing Technologies 2025(1), 776–791

https://doi.org/10.56553/popets-2025-0040

chains formed between IFTTT rules, which are deterministic, but not across IoT apps, which are inferrable with a probability. In this work, we detect privacy violations and return informative alerts about the types of private data being leaked, the relevant channels or interactions, and the leakage conditions and likelihood.

To better alert users, we present a new framework for privacy leakage detection, named PrivacyGuard. This framework supports finer-grained privacy threat identification across multiple platforms, including SmartThings, IFTTT, and OpenHAB, to help users take appropriate precautions based on the types of sensitive information at risk and their specific privacy concerns. To inform users about the likelihood of privacy leaks, we quantify the probability of privacy inference, allowing users to evaluate the possibility of privacy threats and make more informed decisions regarding app permissions and usage. This enables users to be more cautious and avoid integrating high-risk app combinations. To identify potential leaks across multiple apps, we generate device profiles, infer device types and states associated with each trigger-condition-action rule, capture app interactions with cross-app rules, build cross-app chains, calculate inference probabilities, identify types of privacy leakage threats, and provide a graphical user interface for users. PrivacyGuard aims to provide a comprehensive solution for privacy analysis and privacy threat detection in IoT apps, helping developers and users make informed decisions to safeguard user privacy. The main contributions are summarized as follows:

- We identified a new cross-app privacy leakage risk in IoT apps and performed a systematic study on the inference threats.
- We formalized the cross-app chaining problems and defined the trigger-condition-action relations to associate the apps.
- We inferred the device's profile from its usage context, ensuring accurate modeling of devices with different sensitivity levels.
- We computed the probability of privacy inference, which measures both *direct exposure* and *implicit inference* threats.
- We evaluated PrivacyGuard using large-scale real-world datasets consisting of 2,101 SmartApps, 2,788 IFTTT applets, and 2,086 openHAB rules, and demonstrated its effectiveness.

2 Background and The Problem

2.1 Background of IoT Platforms

IoT platforms such as SmartThings, IFTTT, and OpenHAB use IoT apps to interact with IoT devices and execute actions, enabling users to establish event-based rules where automations are triggered by specific events or conditions defined by the user.

Basic Concepts and Terminology. IoT sensors collect inputs from the physical environment, while the actuators control physical objects based on the received commands. Based on their functionalities, we have simple sensors/actuators typically dedicated to a single usage, and multi-linking sensors/actuators, which can be linked to multiple devices. For instance, a humidity sensor is a simple sensor that measures only the humidity, and a button sensor is a multi-linking sensor controlling different devices. Meanwhile, an IoT device can be linked to multiple sensors/actuators, e.g., a smart lock with a built-in camera has a camera sensor, a lock sensor, and a lock actuator. In this work, we do not differentiate a simple sensor/actuator from its attached device, e.g., a humidity sensor and a humidity device are used interchangeably.

Listing 1: SmartApp snippet of *elvis-has-left*.

```
input "door", "capability.contactSensor", title: "door closed?"
input "thermo", "capability.thermostat", title: "thermostat"
input "mode", "enum", options: ["Auto", "Heat", "Cool"]
def installed() { subscribe(door, "contact.closed", handler) }
def handler(evt) { runIn(60, check) }
def check() {
    if (mode == "Auto") { thermo.auto() }
    else if (mode == "Heat") { thermo.heat() }
    else if (mode == "Cool") { thermo.cool() }
}
```

Listing 2: IFTTT snippet of *colorful-inside-temperature*.

```
{ 'id': '/triggers/tado_heating.temperature_above_threshold',
 'name': 'Temperature rises above threshold',
 'description': 'Fires when temperature rises above a value',
 'service_slug': 'tado_heating',
 'service_name': 'tado Heating' },
{ 'id': '/actions/if_notifications.send_notification',
 'name': 'Send a notification from the IFTTT app',
 'description': 'Send a notification to your devices',
 'service_slug': 'if_notifications',
 'service_name': 'Notifications' }
```

Listing 3: OpenHAB snippet of *Turn-light-on-if-presence*.

```
Switch presence "home presence" <presence>
Switch light "room light" <light>
rule "when arrive home turn on the light"
when Item presence changed
then
    if(presence.state == OFF) { sendNotification("email", "msg") }
    else if(presence1.state == ON) { light.sendCommand(ON) }
end
```

A device's *state*, represented by one or multiple attributes, reflects its current status or action. In SmartApps, a device is associated with *capabilities*, each consisting of attributes and commands, defining how an app interacts with the device. We can modify a device's attributes using the commands, directly impacting the device state. For instance, a device with the *lock* capability has a *lock* attribute. Executing the *unlock* command changes the device state to *unlocked*.

IoT Platforms and Apps. Many users deploy multiple IoT platforms simultaneously. We consider three popular platforms, SmartThings, IFTTT, and OpenHAB, to provide a comprehensive and practical analysis of privacy leakages in IoT environments. They use different programming languages and trigger-condition-action paradigms to interact with IoT devices. To analyze cross-platform interactions, we convert and normalize IoT apps into a unified format, from which we derive potential cross-app interactions. This involves identifying the trigger-condition-action paradigms, the device accessed or controlled by each app, the device states, and the commands/methods that each app is permitted to perform.

SmartApps are written in Groovy[1]. It utilizes the subscription or scheduling functions to subscribe to events. The *trigger* can be a device event or a pre-defined event (e.g., timer, mode, or app touch), which activates a handler function to execute an *action* when the *condition* specified in the handler function is satisfied. Listing 1 shows an example app called *elvis-has-left*, which involves the *door* and *thermostat* devices. It subscribes to the door's *close* event, which acts as the trigger, and controls the thermostat. IFTTT applets and OpenHAB rules follow a simple "if-this-then-that" structure. For example, the applet in Listing 2 contains a trigger event "tado_heating.temperature_above_threshold" and

[1]The SmartThings platform has an ongoing transition from Groovy to Node.js. As the main logic of Groovy and Node.js SmartApps is very similar, the rule extraction and app analysis methods used in this work can be easily extended to Node.js SmartApps.

an action "if_notifications.send_notification". While IFTTT uses a plain JSON format and explicitly defines the trigger and action endpoints, OpenHAB adopts a Java-compatible domain-specific language, where the triggers are defined as statements in the *when* block, and the conditions and actions are located in the *then* script block. For example, in Listing 3, the trigger is the presence event and the actions are determined by its state. Besides, OpenHAB defines *things*, *items*, and *channels* to represent physical entities (e.g., devices, web services) as well as their properties.

We also use text information in IoT apps, such as app descriptions, prompts, and code annotations, to derive their functionalities and identify the associated devices. For example, a SmartApp has a definition and preferences block to inform users about its purpose and associated devices, while IFTTT applets and OpenHAB apps use app names and descriptions to outline their functionalities.

Cross-Rule Chain. Trigger-action rules can be chained together [2, 39, 66, 70]. For example, IFTTT applets can be directly linked if (1) one rule's action matches the trigger of another, called *explicitly chaining* [70]; or (2) one rule's action and the other rule's trigger are connected to the same physical medium (e.g., temperature), known as *implicitly chaining*. A chain may have two or more TCA rules.

Taint Sinks. IoT apps define specific method calls to transmit data externally, known as *taint sinks*. Popular sinks include messaging call interfaces for notifications within the mobile app or SMS messages to recipients, internet services API for HTTP requests between the app and external servers, and online network API for emails and social media updates that expand the app's data reach to online networks. For instance, email.send_me_email in IFTTT, sendSms() in Smartthings, and sendNotification() in OpenHAB.

2.2 The Problem and Our Motivation

When multiple apps are installed in an IoT environment, their interactions may result in cross-app chains with potential privacy leakage. Take apps in Listings 1 and 2 as example, which are real-world apps in the SmartAppZoo [71] and an IFTTT applet [76] datasets. Listing 1 is to set the thermostat once the door is closed for a certain time and Listing 2 will send a push notification when the temperature is above a threshold. Sharing the temperature channel, two apps form a cross-app chain so that Listing 1 can activate the trigger of Listing 2. If both apps are malicious and installed by the victim user, the notification will be sent to the attacker, allowing him to infer the status of the door.

However, existing detection approaches often fail to detect such leakage through cross-app interactions with taint sinks. First, they can bypass static and dynamic detection approaches such as SAINT [14] and IoTWatch [8] since there is no taint sink in Listing 1 nor taint source in Listing 2. Besides, according to information-flow-based approaches [22, 66], both devices (*door* and *thermostat*) are associated with a *restricted physical* label, while the *notification* has a *private* label. Therefore, the interactions between two apps involve information flows without any violations, i.e., from *restricted physical* to *restricted physical* and from *restricted physical* to *private*. Finally, some detection approaches such as SafeChain [39] consider only privacy leakages due to publicly observable data (e.g., lights on/off) but not through the taint sinks (e.g., notifications). Therefore, they cannot identify the cross-app leakage if the taint sink is in the last app on the cross-app chain. This new privacy threat due to IoT app interactions calls for a systematic study on potential privacy leakage risks.

To tackle this problem, we face three design challenges: ***(1)*** *How to describe privacy considerations and identify device-specific sensitive data precisely?* SAINT [14] considers all device states and state variables sensitive, while [66] and [39] rely on a privacy policy to assign privacy labels *private*, *public* and *other* to device attributes. Both schemes are too coarse-grained to provide a precise privacy categorization. ***(2)*** *How to discover cross-app chains based on device usage and identify sensitive data exposure?* Existing privacy leakage approaches [22, 39, 66] focus solely on analyzing IFTTT applets, neglecting the examination of multiple IoT platforms. Additionally, they failed to account for device context and usage, where the same capability may control devices at varying sensitivity levels. Finally, ***(3)*** *How to draw privacy inference?* This requires the calculation of the inference probability to accurately measure the likelihood of leakage, as well as the utilization of chain combinations to draw precise inferences regarding user privacy.

2.3 Threat Model

We consider an IoT environment with multiple IoT apps installed, including both benign and malicious apps. The latter has seemingly benign triggers and sinks chosen by an attacker. They could form cross-app chains among themselves or with other benign apps to infer users' private data and send it to unauthorized recipients.

We assume that the victim user would install at least one malicious app. First, by not acting as a data sink itself or not requesting access to any private data, the malicious apps appear benign under existing privacy checks. Non-technology-savvy users may accidentally install and misconfigure malicious apps. Besides, to attract downloads, the adversary could develop malicious, general-purpose apps that control multiple devices or squatting apps that mimic popular devices. Additionally, the adversary is assumed to know the installed apps to construct cross-app chains. This knowledge could be obtained if multiple malicious apps are installed by the same victim or from multiple vulnerable devices manipulated by the attacker [16, 21, 24, 25, 39]. Moreover, a strong adversary could infer smart home configurations by intercepting and analyzing encrypted network traffic. The interaction of trigger or action events between IoT apps and devices can be inferred from traffic behavior (e.g., packet flow, heartbeat) [1, 34, 55, 69, 81]. By analyzing identified IoT devices and sequences of events, we could deduce the IoT apps used in the smart home environment [35, 50, 81]. Finally, we assume the attackers cannot circumvent the security measures of IoT platforms, exploit side channels, or observe information through the sinks, similar to previous studies [8, 14, 19, 20, 39].

3 Privacy Threat Categorization

Existing studies on IoT app privacy often adopt coarse-grained privacy labels. They either consider all types of device information sensitive [8, 14] or use over-general privacy labels such as private, public, or other [22, 39, 66], which are not informative about the privacy leakages. In practice, informative privacy implications are desired, which help users understand the privacy risks and implement precise privacy controls tailored to the unique needs of IoT

Cluster	Devices
Activity	activity sensor,bathtub,bed,shower,sleep sensor
Appliance	blender,blind,boiler,cleaner,coffee-maker,A/C,cooker,cooktop,cooler,curtain, dishwasher,dryer,fan,faucet,fireplace,freezer,fryer,heater,kettle,light,microwave, mop,mower,oven,printer,projector,refrigerator,stove,tv,vacuum,washer
Car	car,vehicle
Door	door,garage door,gate,window
Fitness	step sensor,watch,wristband
Health	body mass index sensor,body weight sensor,health,medicine
Location	geolocation
Lock	lock
Monitoring	camera,audio,image,speech,video
Music	player,soundbar,speaker
Presence	location mode,occupancy sensor,presence sensor

Table 1: Clusters of privacy-sensitive devices.

devices. In this work, we design fine-grained privacy labels and assign them to different IoT devices (or device clusters).

Device Clusters. IoT devices gather and process various types of data, some of which are privacy-sensitive. To assess potential privacy risks, we need to identify devices handling sensitive data, which is challenging due to device diversity and data heterogeneity.

We first collected a list of 88 types of supported devices from popular IoT platforms, i.e., Google Smart Home, Apple HomeKit, and SmartThings [5, 33, 60]. Using the spaCy NLP library [65], we computed the word embeddings of device names and applied the Agglomerative clustering [64] to group them based on the distances. Subsequently, we cross-referenced the clusters with the official service categories in IFTTT [42] and merged smaller clusters. For instance, cleaners and vacuums were in the same cluster. They were merged with the boiler cluster to form the appliance cluster. Finally, we adjusted the remaining categories according to their privacy implications. For example, as the sensory data related to the bed devices often indicates sleep patterns or occupancy, we move it from the appliance cluster to a new activity cluster. We also added new clusters for the lock, door, and presence devices that are frequently used in IoT apps. Table 1 lists the final clusters of privacy-sensitive IoT device types.

The Activity, Health, and Fitness clusters may exhibit certain similarities. Within the Activity cluster, the devices and sensors are designed to monitor an individual's daily routines and habits. They can track activities such as sleep patterns, shower usage, time spent in bed, and physical activity. These activities can offer insights into an individual's overall lifestyle and behavioral patterns. Devices and sensors categorized under the Health cluster are directly related to an individual's physical health status. Medicine data includes information about medications or treatments, while parameters like body mass index and body weight serve as crucial health indicators. Monitoring these aspects is vital for evaluating and preserving an individual's health. Devices within the Fitness cluster are typically employed in fitness-related contexts. For instance, wristbands and watches are commonly used to monitor activity data, including metrics like step count, distance traveled, and other fitness parameters. Step sensors are instrumental in tracking movement and exercise, making them relevant to fitness monitoring. The Health cluster includes personal identifiers and sensitive medical conditions, making it the most sensitive among these three clusters. On the other hand, the Activity cluster involves insights into an individual's daily routines and habits, which are also more sensitive compared to the Fitness cluster. Hence, it is important to maintain a clear distinction between these three clusters due to the sensitivity and the unique nature of the data.

Privacy Label	Device Clusters
(A) Identification	Monitoring
(B) Localization & Tracking	Activity,Appliance,Car,Door,Location,Lock, Monitoring,Music,Presence
(C1) Activity Profiling	Activity,Appliance,Car,Monitoring,Music
(C2) Health Profiling	Fitness,Health
(D) Lifecycle Transitions	Health,Location,Monitoring

Table 2: Privacy labels associated with different device clusters.

Privacy Labels. IoT privacy can be defined as a threefold guarantee for awareness of privacy risks imposed by smart things and services, individual control over the collection and processing of personal information, and awareness and control of subsequent use and dissemination of personal information [85]. Following this definition, Ziegeldorf et al. identified seven threat categories in the context of IoT. Four of them are relevant to home automation systems: (A) *identification* threat of associating an identifier of any kind from ID to camera data. (B) *localization and tracking* threat of determining a person's location, for example, from GPS, geolocation, or presence data. (C) *profiling* threat of correlating data and other profiles to infer individuals' interests. A broad spectrum of data can be used for profiling, therefore, we further classify it into (C1) *activity profiling* that uncovers physical activities, daily life, living habits, and routines, etc., and (C2) *health profiling* that reveals health conditions or treatment. And (D) *lifecycle transitions* threat that involves private information associated with changes in control during an item's lifecycle, such as photos left on a used camera or location data remaining on a used phone.

With a small number of device clusters, we manually assign privacy labels to each cluster according to its privacy implications. A device cluster may receive one or multiple privacy labels. For instance, the *Health* cluster deals with health and biometrics data and thus has a risk of exposing *Lifecycle Transitions* and *Health Profiling* data. We identified all device clusters with at least one privacy threat, as shown in Table 2, and explained the rationale of manual label assignment in Appendix A.

4 Cross-App Privacy Leakage

4.1 Cross-App Rule Chaining

The cross-app privacy leakage occurs only when the rules of multiple installed apps can be chained together.

Rule Chaining and App Chaining. Rules can be chained in two ways. First, two rules are chained if one's action activates the other's trigger, referred to as an ***activate*** relation. Secondly, they are chained if one's action satisfies the condition of the other, referred to as an ***enable*** relation.

As shown in Figure 1(a), the action $a_{\mathbf{R}_1}$ directly fires the trigger $t_{\mathbf{R}_2}$, so rules R_1 and R_2 have an *activate* relation represented by the dashed arrow. As shown in Figure 1(b), the action $a_{\mathbf{R}_3}$ enables the condition $c_{\mathbf{R}_4}$. Hence, R_3 and R_4 form an *enable* relation. Two apps are chained if their rules form an *activate* or an *enable* relation. In Figure 1(c), multiple apps $\mathbf{A}_1, ..., \mathbf{A}_n$ can be linked together if their rules $\mathbf{R}_1, ..., \mathbf{R}_n$ are chained. The first app and the last app are called the entry and exit apps, respectively.

Relations between Events and Rules. Rules can be chained either *explicitly* or *implicitly*[70]. Explicit rule chaining occurs when the action event of one rule *activates* or *enables* another rule on the same device. Similarly, when the action event of one rule influences the physical channel shared with the trigger event or condition event

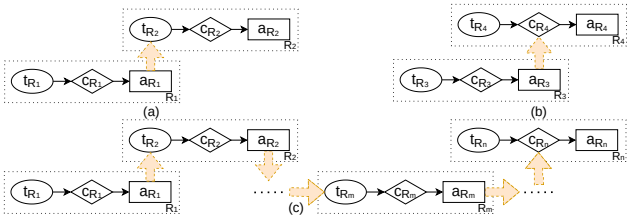

Figure 1: (a) *Activate* Relation; (b) *Enable* Relation; (c) Multiple-App Chain.

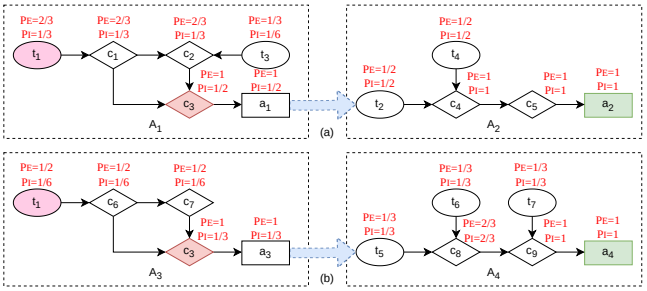

Figure 2: Examples of the probability inference for cross-app chains. P_E and P_I denote the execution and inference probabilities, respectively.

of another rule on different devices, this action *implicitly activates* or *enables* another rule. Therefore, rule chaining is associated with trigger, condition, and action events, as well as the devices and physical channels involved.

Device and physical medium. Consider a device s with a set of *commands* denoted as $\mathcal{C}(s)$ and *attributes* with *values* denoted as $\mathcal{A}(s)$ and $\mathcal{V}(s)$, respectively. A device may associate with an optional *physical channel*, denoted as $s.channel$, which either influences the device to change (some of) its attribute value(s) or gets influenced by some device commands.

Event. An event e on a device may execute a subset of commands, denoted as $\mathcal{C}(e)$, and change the values of a subset of attributes, denoted as $\mathcal{A}(e)$ and $\mathcal{V}(e)$, respectively. The execution of the event occurs under a condition (called a *constraint*), which involves a subset of device attributes, i.e., $\mathcal{A}(e.constraint)$. The constraint is satisfied if these attributes hold values in $\mathcal{V}(e.constraint)$. Next, we formally define the match and influence relations.

*Definition 1 (*match*).* Events p and q have a *match* relation if they occur on the same device and by calling commands in $\mathcal{C}(p)$, p changes the device state so that the constraint of q is satisfied.

$$\begin{aligned} match(p,q) \equiv\ & (s(p) = s(q)) \\ & \wedge\ ((\forall\ attr \in (\ \mathcal{A}(p) \cap \mathcal{A}(q.constraint))) \\ & \Rightarrow\ (attr.value \in \mathcal{V}(q.constraint))) \end{aligned}$$

*Definition 2 (*influence*).* Events p and q have an *influence* relation if they occur on different devices but share the same physical channel.

$$\begin{aligned} influence(p,q) \equiv\ & (s(p) \neq s(q)) \\ & \wedge\ (s(p).channel = s(q).channel) \end{aligned}$$

Based on these definitions, we further define the triggered and enabled relations between two rules, i.e., R_i with an action event a and R_j with a trigger event t. R_j has a condition, which is a set of constraints extracted from all the execution paths from the entry point to sinks excluding the constraint of the trigger event t. We denote the condition events of a TCA rule as $\mathcal{B} = \{b_1, ..., b_k\}$.

*Definition 3 (*activate*).* R_i activates R_j if a and t hold either a *match* or an *influence* relation.

$$\begin{aligned} activate(R_i,R_j) \equiv\ & (\forall\ a \in R_i, t \in R_j, \\ & (match(a,t)\ \vee\ influence(a,t))) \end{aligned}$$

*Definition 4 (*enable*).* R_i enables R_j if a and at least one of the condition events of R_j hold either a *match* or an *influence* relation.

$$\begin{aligned} enable(R_i,R_j) \equiv\ & (\forall\ a \in R_i, (\exists\ b \in \mathcal{B}, \\ & (match(a,b)\ \vee\ influence(a,b)))) \end{aligned}$$

4.2 Privacy Inferences from Cross-App Chains

Next, we explain privacy inferences drawn from cross-app chains.

Privacy Inferences. When multiple apps form a cross-app chain and the execution of the action event of the exit app is observed, one could infer that: (1) at least one trigger of the entry app is activated, and if so (2) along the path from this trigger to the executed action of the exit app, *all* the trigger events are activated, *all* the conditions are satisfied, and *all* the actions are executed. We proceed by identifying the devices and device states associated with these triggers, conditions, and actions, retrieving their privacy labels, and inferring potential privacy leakage of the cross-app chain.

For example, two apps in Figure 2(a) are chained together, since the action a_1 can activate the trigger t_2. App A_1 has two triggers, t_1 and t_3, and three TCA rules leading to action a_1. Meanwhile, app A_2 with triggers t_2 and t_4 has two TCA rules leading to action a_2. From the execution of a_2, we can infer that conditions c_4 and c_5 are deterministically satisfied. If c_4 or c_5 is associated with privacy-related data or devices (e.g., a lock), this inference results in privacy leakage. We could also infer that either the trigger t_2 or t_4 is activated, however, we cannot determine which trigger is executed since they have an equal activation probability. Similarly, if t_2 is activated by a_1, we cannot determine whether t_1 or t_3 was activated, or which condition was satisfied, because three TCA rules in A_1 share the same action. This example illustrates a probability-based inference method for cross-app privacy analysis. An app's triggers or conditions can be activated or enabled by several apps, and each app can have multiple TCA rules that share the same action. As a result, most probability inferences are non-deterministic.

The inferences made based on cross-app interactions reveal information about *device states* of all the devices associated with the triggers, conditions, and actions in the app chain, which may be privacy-sensitive. As discussed in Section 3, these device clusters with one or multiple privacy labels are considered sensitive. In certain scenarios, we can precisely identify the devices in use. For instance, in SmartThings rules, the device ID can be obtained from the rule's JSON files. This allows us to determine the exact device used in the rule by accessing the device IDs of each device. Similarly, in some IFTTT applets, the exact device can be identified through the service name defined in the IFTTT documentation.

However, there are scenarios where the devices associated with the rules are unknown. In such cases, we can deduce the device

from its textual descriptions or system-defined interfaces. SmartThings employs a capability-based permission model to regulate a SmartApp's access to a SmartDevice. This model maintains a list of capabilities [61], outlining each capability's purpose, use cases, commands, attributes, etc. Most capabilities can be directly mapped to specific devices in a deterministic manner. For instance, the *presence* capability is exclusively associated with presence sensors. Nonetheless, certain capabilities can be linked to different devices, and in such instances, we can only infer the devices from the textual descriptions. For example, capabilities like *switch*, *button*, *outlet*, and *switchLevel* can grant access to all devices with on/off functionality. We need to extract information about the device, called *device profile*, from the app's textual data, and privacy labels need to be assigned to each device based on this information.

Privacy Leakage Risks. A privacy risk occurs when data with privacy indication is disclosed, not all cross-app chains have a privacy leakage risk. For example, SmartThings provides two APIs, i.e., messaging and Internet, for apps to interact with external services. Only if an app uses these APIs to send *sensitive* data out without informing the users [8] or send data to unknown endpoints, it poses a privacy risk. Therefore, only the cross-app chains whose exit app has *taint sinks* are considered to have potential privacy leakage risks. Furthermore, the simultaneous occurrence of two or more sinks may indicate a strong correlation between the events that triggered them, which can be used to infer the likelihood of a common trigger, condition, or action being executed.

5 PrivacyGuard Design

In this section, we introduce PrivacyGuard, a privacy analysis tool designed to identify potential privacy leakage risks in a multi-app multi-platform environment. As depicted in Figure 3, PrivacyGuard extracts text data from apps, infers device types and states (i.e., *device profile generator*), analyzes the app and constructs TCA rules (i.e., *rule constructor*), identifies cross-app chains (i.e., *chain builder*), and conducts privacy inference analysis (i.e., *privacy threat detector*).

5.1 Device Profile Generator

Multi-linking sensors or actuators such as switches, buttons, or acceleration sensors can be connected to privacy-sensitive or non-sensitive devices. They do not inherently exhibit sensitivity when used alone. Instead, their sensitivity depends on the context and usage, for example, when controlling privacy-sensitive devices (e.g., a switch sensor linked to a lock, or an acceleration sensor linked to a door). To accurately infer privacy indications, we create a *device profile* for each device, including device type and state.

Text Data Extraction. Each IoT app has a human-written *app description* about its main functionalities and behavior. For example, it guides the users to configure the devices to control during SmartApp installation. Besides, the *input prompts*, *section prompts*, and *variable names* in the preferences block statements provide useful information to assist the user in this process. For example, an input prompt *which door*, a section prompt *select a door*, or a variable named *garageDoor* indicate a door-type device being operated. For OpenHAB rules, the device usage and context are often located in the rule description and *item* definition, such as "room light". In IFTTT, the functionalities of triggers and actions can be deduced

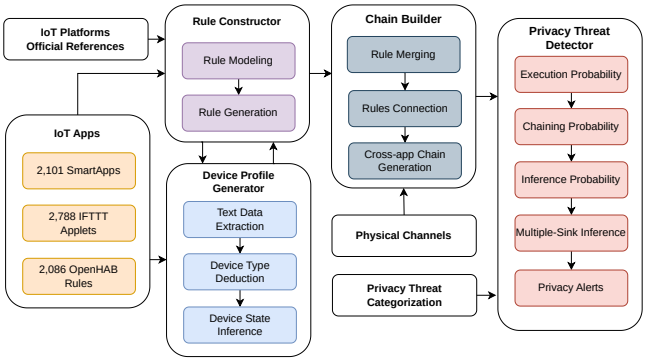

Figure 3: The architecture of PrivacyGuard.

from their identifiers, names, and descriptions, e.g., the action *turn on lights* controls a light device with the attribute *light on*.

Device Type Deduction. We infer the devices and their states involved in each rule component from the text data. The app descriptions often contain fuzzy information about the device. For example, a SmartApp *co2-vent* has a description *Turn on a switch when CO_2 levels are too high*, but its section prompt specifies that the switch is connected to a *ventilation fan*. So, we extract the capability from each input statement, the input prompt, section prompt, variable names from the input statement, and the app description from the definition block. In OpenHAB rules and IFTTT applets, clues about devices and attribute values can be found in the description, label text, ID name, and variable name.

We take a Natural Language Processing (NLP) approach to analyze the text data. To deduce the device, we conduct a part-of-speech (POS) tagging using the spaCy NLP library [65] and extract the nouns and proper nouns as candidates, which are then matched against the device type names used across different IoT platforms. This comparison is guided by a requirement that their similarity score must surpass a pre-set threshold of 0.6 [43]. However, the texts may contain multiple distinct nouns, and the frequency of their occurrences plays a significant role in contributing to the overall outcome. To capture this, we assign a weight to each text such that a noun's score is calculated by multiplying its occurrence count by the text's weight. Since the app description is considered less significant compared to other texts, we assign it a proportionally lower weight for its nouns. This approach allows us to consider all nouns from all texts, with each noun having a unique score that reflects both its frequency and contextual importance.

Next, we compare each noun with every known device type name to form pairs and measure their similarity. If the similarity is below a predefined threshold, the pair is discarded. For each valid pair, a score is calculated by multiplying the noun's score by its similarity to the device type name. These pairs are then sorted in descending order based on their scores. The device type name from the pair with the highest score is selected as the associated device type for that noun. This process is repeated for each candidate noun to determine the device type that appears most frequently, which is then selected as the resulting device type.

Device State Inference. Since devices in SmartApps are linked to specific capabilities, and OpenHAB rules use similar actions to SmartApps, such as *on/off*, *open/close*, there's no need to infer device

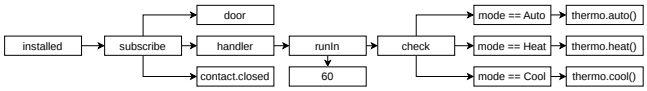

Figure 4: The CFG Diagram of Listing 1.

states in those cases. However, IFTTT applets present a different challenge: some devices support more actions on the IFTTT platform. For example, *Philips Hue* can perform additional actions such as *blinking*, *color looping*, and *toggling*. To accommodate these, we use the device attributes and values in SmartThings as a reference to map a device to its possible attributes and values. For devices with additional actions, we add extra attributes and values to represent these actions. We then infer the device state by extracting verbs and their corresponding adpositions, matching them with potential actions through word similarity score comparisons. For instance, in the phrase *turn off the light*, the adposition *off* is associated with the verb *turn*, indicating that the light is being turned off.

5.2 Rule Constructor

IoT apps have common structures that make it easier to extract TCA rules. We refer to the trigger, condition, or action as a *component* of a TCA rule. Our process involves identifying each component, applying the corresponding device profile, and constructing the rules for each IoT app on different platforms. We illustrate this approach with a code snippet in Listings 1, 2, and 3 as examples.

Rule Modeling. Listing 2 is an IFTTT applet whose trigger and action can be directly identified from the JSON file. We excluded the filter code about when an applet should be run or skipped. Ignoring the filter code does not affect our model, as we aim to detect the execution of the rule when it is triggered. Each IFTTT applet offers at least one trigger-action pair. The ID of the trigger and action follows the format: /endpoint type/API endpoint slug. The endpoint type can be a trigger or an action, and the API endpoint slug contains the service slug and service command. A service slug corresponds to an IoT device or a service, while a service command provides information about the device attribute value. For example, *temperature_above_threshold* indicates that the temperature is above the threshold.

For SmartApps and OpenHAB rules, we first construct the Abstract Syntax Trees (ASTs). For SmartApps, we directly use the *ScriptToTreeNodeAdapter* function provided by Groovy. As the script block of OpenHAB rules is syntax-compatible with Groovy, except for type casts. We use regular expressions to deal with type casting and extract the ASTs. From the AST of each app, we build a control flow graph (CFG) for static analysis, similar to prior app analysis approaches [2, 21, 44]. For example, Figure 4 shows the CFG diagram of the app in Listing 1.

Action Nodes Extraction. With the AST, we search for method call nodes. In SmartApps, the *subscribe* or *schedule* method call nodes, such as Line 4 in Listing 1, serve as the entry node of an app. In OpenHAB, the *when* statement such as Line 4 in Listing 3 serves as the entry node. Method call nodes consist of a receiver, a method name, and an argument list, such as *thermo.auto()* and *light1.sendCommand(ON)* in Line 7 of Listings 1 and 3.

Action nodes are method call nodes that can be categorized into *actuator nodes* and *sink nodes*. Actuator nodes represent commands that interact with physical devices to change their states, while sink nodes communicate with external channels. We filter out method call nodes where the receivers are associated with devices, and the method names are defined as actions in the official reference, categorizing them as actuator nodes(e.g., *thermo.auto()* and *light1.sendCommand(ON)*). Additionally, we consider method call nodes involving taint sinks as sink nodes(e.g., *sendNotification*).

Permission and Constraint Identification. In SmartApps, we extract devices' identifiers and capabilities from *input* method call nodes (e.g., Line 1 in Listing 1). Similarly, in OpenHAB, we identify all properties of home automation from *items* definitions (e.g., Line 1 in Listing 3). We refer to these as permission nodes. Method nodes represent functions that can be invoked within other functions, such as *check()* in Line 6 of Listing 1. The method call nodes and their argument lists are recorded to build a CFG. In particular, we extract method call paths from each method call node to its nearest method node, logging all the conditions encountered along the way. For example, we can extract a method call path between nodes *thermo.auto()* (Line 7 in Listing 1) and *check()* (Line 6 in Listing 1), which has a condition *mode==Auto*. With all the method call paths, we construct the paths from the entry node to the action nodes.

We identify condition nodes with an *if* or a *switch* statement by traversing backward along the path. Predicate constraints are constructed from the boolean expression in an *if* statement or by combining associated *case* statement(s). Then, we examine the conditions on the paths and identify all the variables associated with permission nodes, which are potential connection points for chaining TCA rules. We check if a variable is in the parameter list of the current method node and then recursively retrieve the parent node and/or its siblings along the path to locate the associated permission nodes. For example, in Listing 1, the variable *evt* in the parameter list of the function *handler* can be associated with the capability *contactSensor* based on the subscription (Line 4), thereby linking to the device *door*.

In SmartApps, potential triggers involve event subscription methods that subscribe event handlers to device attribute values, locations, or app touches, e.g., *subscribe(door, contact.closed, handler)* in Line 4 of Listing 1, or schedule call functions such as *schedule* or *run** that trigger event handler methods at a specific time. For OpenHAB rules, they follow the format "when *trigger*, if *condition*, then *action*", the *when* statements serve as subscription nodes, e.g., Line 4 of Listing 3. These subscription nodes contain trigger node information, allowing us to construct the TCA rule starting from action nodes and traversing backward to subscription nodes using a graph traversal algorithm. Additionally, we identify the corresponding constraints of each trigger from the subscription method and its event handler [21, 24]. Specifically, if the trigger subscribes to a device event (e.g., *contact.closed*), its constraint can be directly derived from that event. Otherwise, if it subscribes to all device events (e.g., *contact*), the event handler needs to compare the event's value, which becomes the constraint of the trigger. The other conditions on the path are parsed as the constraints of conditions.

Rule Generation. After modeling the rule and combining it with the *device profile* of each component, we generate the TCA rules and eliminate duplicate rules to avoid redundant cross-app chains.

Trigger	Condition	Action	Type
device: door attribute: contact operator: = value: closed	device: user-defined attribute: mode operator: = value: auto	device: thermostat attribute: mode operator: = value: auto	SmartApp Actuator
device: presence attribute: presence operator: = value: present		device: light attribute: switch operator: = value: on	OpenHAB Actuator
device: thermostat attribute: temperature operator: > value: threshold		device: sink attribute: notification operator: value:	IFTTT Sink

Table 3: Example rules extracted from Listings 1, 2, and 3.

Channel	Actuator Devices	Sensors
Humidity	dehumidifier,fan,humidifier,vent	humidity
Leakage	faucet,sprayer,sprinkler,valve	water
Location	location	presence
Luminance	light	illuminance
Motion	blind,curtain,cleaner,door,fan,garage door, gate,mop,vacuum,window	motion
Power	A/C,cooler,fireplace,heater,kettle,stove,oven	gas,power
Smoke	fireplace,heater,purifier,stove,oven	carbon dioxide, smoke, carbon monoxide
Sound	alarm,player,soundbar,speaker,tv	sound
Temperature	A/C,cooler,fan,fireplace,heater,stove,oven, thermostat	temperature

Table 4: Physical channels and the associated devices.

Rule Assembly. The trigger and condition constraints follow a specific structure: name.attribute-operator-value. Each component of a TCA rule is linked to a device type, an attribute, and their corresponding values. Therefore, we break down each component into 4 parts: device type, attribute, operator, and value. For instance, the action *light.on* associated with the capability *switch* is decomposed into: device(light), attribute(switch), operator(==), and value(on). Table 3 shows three TCA rules extracted from Listing 1, 2, and 3.

Duplicate Rule Removal. Duplicate rules can lead to redundant cross-app chains, resulting in decreased efficiency and potentially incorrect outcomes. When two rules share identical triggers and conditions, and the associated action involves the same device type and state (for actuator rules) or utilizes any taint sinks (for sink rules), we classify these rules as identical. In the case of actuator rules, the action activates the triggers or enables the conditions of the other rule. If any two rules have the same action with the same device type and state, the actions can be connected to the same chains, indicating they are identical. For sink rules, if two rules share identical triggers and conditions but use different taint sinks, they serve the same purpose of sending messages externally and are therefore considered identical. After recognizing duplicate rules, we eliminate them to prevent redundant processing.

5.3 Chain Builder

To create cross-app chains, we need to connect two separate TCA rules through an action-trigger/action-condition pair. The TCA rules can be categorized into *actuator rules* and *sink rules*. Actuator rules involve actions carried out by actuator nodes, whereas the actions within sink rules function as sink nodes, which implies that sink rules do not affect or connect with other triggers or conditions. Therefore, only the actuator rules can *activate* or *enable* other rules.

Rule Merging. When multiple rules share the same action, they can potentially *activate* or *enable* the same set of rules or chains. To reduce redundancy when building cross-app chains and simplify the calculation of rule execution likelihood (Section 5.4), we combine TCA rules with the same action into what we call a *merged rule*. The merged rules create several branches. Each branch represents a different TCA rule sharing the same action. Within a merged rule, each branch has an equal likelihood of leading to the execution of that action. The number of merged rules depends on the number of unique actions of the app.

Rule Connection. Rules can be explicitly or implicitly chained based on the actions match or influence on other triggers or conditions. Two rules are explicitly chained if they operate on the same device with the same device state. For implicit chaining, we followed the approach in [24] to discover shared physical channels. We identified 9 physical channels with corresponding actuator devices and sensors, as shown in Table 4. Compared to [24], our results include two new physical channels, *sound* and *power*. Note that our focus is not on discovering physical channels but on identifying physical channels enabling cross-app chains with the *influence* relation. Furthermore, the *influence* relation is one-way, flowing from the actuator device to the sensor.

Two rules connected through an action-trigger pair or an action-condition pair form an *activate* or *enable* relation, respectively. We can connect two rules by examining the relationship between the action node and the trigger/condition node in each rule, resulting in four distinct types of connections, i.e., *match-activate*, *influence-activate*, *match-enable*, and *influence-enable*. To connect two merged rules, we examine the action of one merged rule against all the triggers and conditions of the other merged rule.

Cross-app Chain Generation. Given a set of apps, we extract all the *merged actuator rules* and *merged sink rules* from each app to generate cross-app chains. We start with a sink rule and connect it with all other rules via available paths until the chains cannot be extended further. Then, we extract the chains reaching their maximum lengths along all paths. More details are presented in Appendix B. We used two stacks, *nodes* and *paths*, to track the current rule and the chains that have not yet reached the maximum length. Besides, we record the rules that have been visited in the *visited* set to avoid loops. Next, we find all adjacent rules that can *activate* or *enable* the current rule. No adjacent rules indicate that the current chain has reached its maximum length along the path.

5.4 Privacy Threat Detector

Given a set of IoT apps, PrivacyGuard generates a list of cross-app chains that consist of multiple TCA rules, which can potentially leak user privacy. To measure the risk of leaks, we design a probability-based method. The probability serves as a measure of confidence to compare the risks. A higher probability indicates greater confidence about the derived privacy leakage threat.

Execution Probability. We calculate the likelihood of the triggers being activated, the conditions being satisfied and the actions being executed, which is defined as the *execution probability* (P_E). With a single TCA rule, the execution of its action indicates that its triggers are activated and its conditions are met. So, the execution probability is 100%. However, when n TCA rules share the same action, the execution probability of each component (a trigger, a condition, or an action) of each rule is $1/n$. The overall execution probability of a component is the sum of its execution probabilities

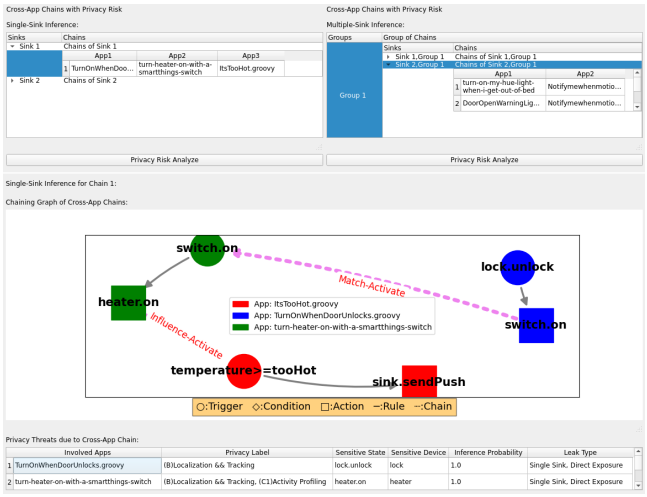

Figure 5: The interface and privacy alerts generated by PrivacyGuard.

across the rules. As shown in Figure 2(a), the action a_1 in the app A_1 is associated with three rules: $R_1 = t_1 \rightarrow c_1 \rightarrow c_3 \rightarrow a_1$, $R_2 = t_1 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow a_1$, and $R_3 = t_3 \rightarrow c_2 \rightarrow c_3 \rightarrow a_1$. Therefore, the execution probabilities of the triggers, conditions and actions in each rule are $1/3$, from which we compute the overall execution probability of each component: $P_{E_{t_3}} = 1/3$, $P_{E_{t_1}} = P_{E_{c_1}} = P_{E_{c_2}} = 2/3$, and $P_{E_{c_3}} = P_{E_{a_1}} = 1$.

Chaining Probability. We denote the probability that two rules can be chained together as the *chaining probability* (P_C). It is determined by two factors: (1) the *connection probability* about how likely a chain can be constructed with *match* or *influence* relation, and (2) the number of rules connected to the second rule. For chains established via the *match* relation, the connection probability is 100%. For example, the rules *turn on the light* and *activate by the light* are chained deterministically. However, in chains established through the *influence* relation, where two rules operate on different devices sharing the same physical channel, the connection probability may be lower. An adversary could exploit specific conditions to increase the connection probability. For instance, since illuminance is affected by factors such as weather, time of day, and light sources, the connection probability based on the light-illuminance relation could reach 100% at night. Similar to previous work [21, 24, 70], we assume that the attacker is aware of scenarios where the connection probability is deterministic, using a 100% connection probability for analysis. The chaining probability between two rules is calculated by dividing the connection probability by the number of rules linked to the second rule, i.e., $1/n$. For example, in Figure 2(a), the chaining probability between a_1 and t_2 is $P_{C_{a_1,t_2}} = 1$, because only one app A_1 can be linked to the trigger t_2 of app A_2.

Inference Probability. In a cross-app chain, by observing the execution of the exit app's action, an attacker could infer the status of each trigger, condition or action in the apps along the chain with a certain likelihood, referred to as *inference probability* (P_I). A privacy leakage is defined as a *direct exposure* if its inference probability is 100% or an *implicit inference* if the probability is larger than 50%. When two apps are connected, the inference probability of a component in the first app is the product of these three probabilities: its execution probability in the first app, the chaining probability between the two apps, and the inference probability of the trigger or condition in the second app. If the app is the exit app in the chain, the execution probability is the inference probability. For example, the inference probability of the trigger t_1 in Figure 2(a) can be calculated by $P_{I_{t_1}} = P_{E_{t_1}} \times P_{C_{a_1,t_2}} \times P_{I_{t_2}} = 1/3$.

Multiple-Sink Inference. Each cross-app chain has only one exit point, known as a sink. The inference probabilities discussed above are based on observing the execution of this single sink, called *single-sink inference*. In multiple cross-app chains with multiple sinks, we can observe the execution of all the sinks and combine the inference probabilities for the same component across different chains. This results in an increase in the overall inference probability called *multiple-sink inference*. If n cross-app chains share a common component whose inference probability in the k-th chain is P_{I_k}, the overall inference probability that this component is executed in at least one chain is $P_I = 1 - \prod_{k=1}^{n} \left(1 - P_{I_k}\right)$. For example, consider two cross-app chains with sinks a_2 and a_4 as shown in Figure 2. If both sinks are activated, the overall inference probability $P_{I_{t_1}}$ increases to $4/9$, and $P_{I_{c_3}}$ increases to $2/3$.

Privacy Alerts. PrivacyGuard detects potential cross-app chains and calculates the associated inference probabilities. Figure 5 shows the interface of PrivacyGuard and the privacy alerts for the user. Two examples of single-sink and multiple-sink inferences are shown at the top. Each provides a list of privacy-sensitive cross-app chains and the corresponding privacy threats. At the bottom, it visualizes the cross-app chains with privacy risks, where different IoT apps are represented by different colors, and triggers, conditions, or actions are represented by different shapes. The apps on the cross-app chains are connected by colored arrows with labels of the connection types.

6 Evaluations and Analysis

We implemented PrivacyGuard in both Groovy and Python, and evaluated its performance through experiments and case studies. We ran PrivacyGuard on a desktop computer with a 3.60GHz 12-core Intel I7-12700K processor and 16GB of RAM. To test with real-world data, we built a comprehensive dataset, excluding apps that did not include identifiable IoT devices, subscriptions, or that were malicious apps created for testing purposes [8]. The resulting dataset included 6,975 apps: 2,101 SmartApps from the SmartAppZoo dataset [71], 2,788 IFTTT applets [76], and 2,086 OpenHAB rules sourced from third-party apps on GitHub. Among these, 105 were classified as *fat* apps (those connecting to 10 or more devices). During our static analysis, we identified 17 typos and 6 previously unrecognized capabilities in the SmartApps, which we manually corrected and added to our dataset.

6.1 Correctness of PrivacyGuard

Device Profile Inference. We evaluated PrivacyGuard's capability to accurately identify device profiles (i.e., device types and states). First, we manually analyzed the text data and source code of 184 SmartThings official apps and 100 OpenHAB rules to obtain the ground truth about device types and states. We adopted the results from Safechain [39] for IFTTT applets and selected 935 IFTTT applets associated with these devices. Next, we compared the ground

Platforms	# Truth	Names	Accuracy
SmartThings	184	Device Type	96.6%
		Device States	100%
IFTTT	935	Device Type	93.0%
		Device States	96.2%
OpenHAB	100	Device Type	91.0%
		Device States	100%

Table 5: The accuracy of device profile inference.

Privacy Label	Direct Exposure	Implicit Inference
(A) Identification	audio,camera,video	audio,camera,video
(B) Localization & Tracking	location,door,lock,presence, fan,cooler,heater	location,door,lock,presence, cooler,fan
(C1) Activity Profiling	fan,cooler,heater,player, A/C,light	cooler,fan,heater,player,light
(C2) Health Profiling	watch,health,wristband	watch,health,wristband
(D) Lifecycle Transitions	audio,camera,health	audio,health

Table 6: Sensitive devices associated with different privacy threats.

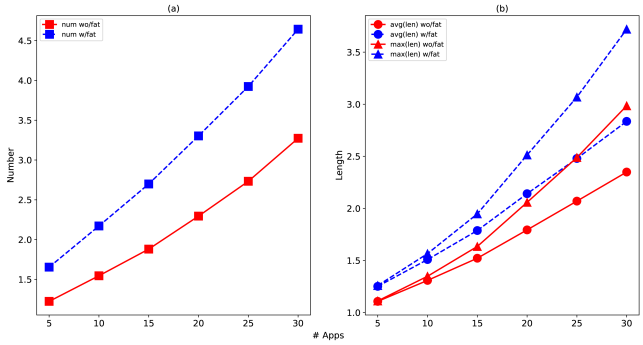

Figure 6: The average (a) number and (b) length of cross-app chains without and with a fat app.

truth with the results from the automated analysis to evaluate PrivacyGuard's performance in device profile inference. Table 5 shows the accuracy on each platform.

We also manually reviewed several incorrect cases. For example, we cannot infer *activate scene* as a LIFX light due to the lack of contextually relevant terms to associate it with *light*. Similarly, PrivacyGuard associated the app *it moved* to a *motion* device based on its description *when movement is detected*, although it is associated with an *acceleration* device.

Rule and Chain Extraction. We randomly selected 300 TCA rules from 184 official SmartApps and 200 TCA rules from randomly chosen IFTTT applets and OpenHAB rules. The results showed that PrivacyGuard accurately extracted these TCA rules. Next, we randomly chose 100 IoT apps to construct cross-app chains and manually verified the chains identified by PrivacyGuard. We confirmed that all of these chains were correct.

6.2 Cross-App Chains and their Patterns

Two-App Chains. Using a dataset of 6,870 IoT apps (excluding 105 *fat* apps), we extracted TCA rules from the apps and constructed chains involving two apps each. We explored how likely a cross-app chain could be formed. A total of 7,458,979 two-app chains were identified. The *fat* apps were excluded because they would create a huge number of connections with other apps, potentially introducing bias. Among these chains, 15.40% involve sink apps, and 7.67% has the potential to leak at least one type of privacy data.

We also observed interesting interaction patterns that may assist attack design. For example, among all the two-app chains, 95.43% and 4.57% are formed through the *activate* and *enable* relations, respectively. Additionally, 38.19% are established through the *influence* relation via shared physical channels, with the most popular physical channel being *thermostat-temperature* and *light-illuminance*. The chains created with the *match* relation frequently involve device pairs *light-light* and *switch-switch*.

Multiple-App Chains. We simulated real-world usage scenarios of IoT apps to evaluate the likelihood of forming multiple-app chains. Specifically, we randomly selected 5 to 30 apps from two datasets: one without and one with the *fat* apps, and constructed all possible cross-app chains. Each experiment was repeated 5,000 times, and calculated the average values for the metrics used.

Average Number and Length of Chains. In the multiple-app experiments, we considered all possible chains with varying lengths. As shown in Figure 6, the average number of chains increases linearly with the number of installed apps. Similarly, both the average and maximum lengths of the chains show a linear trend. With a small number of installed apps (e.g., 5), it is less likely to form a cross-app or even two-app chain. When the number of apps increases (e.g., 25), on average one two-app or three-app chain could exist. If we include one *fat* app, the change to form the cross-app chain increases significantly. This means installing many apps or a *fat* app would increase the cross-app privacy risk.

Probability of Risky Chains. Among all cross-app chains, we identified the ones with privacy leakage risks and measured the proportion of these risky chains. The average probability of constructing at least one risky chain increases with the number of installed apps, e.g., from 3.28% with 5 apps to 20.97% with 30 apps. With *fat* apps, the probability rose to 5.60% with 5 apps and 30.06% with 30 apps. Therefore, users should be cautious about the number of IoT apps they install and limit the use of fat apps.

Typical Sensitive Devices. We further extracted the typical sensitive devices involved in cross-app chains and calculated their occurrence frequency related to each type of privacy leakage through *direct exposure* and *implicit inference*. Table 6 reported the top-80% devices in each type ordered by frequency. We avoided the fat apps in this experiment to obtain a clear picture of the common devices involved. We found that most devices are associated with two privacy threats, *Localization & Tracking* and *Activity Profiling*. In contrast, the *Health Profiling* threat is associated with relatively fewer devices. Certain devices such as cameras are associated with multiple privacy labels, they should be used cautiously or possibly restricted from integration with specific apps or services by users. In future research, information about typical sensitive devices can be used to develop and implement appropriate privacy controls and measures aimed at safeguarding privacy information.

Average Number of Risky Chains with Different Privacy Threats. We measured the average number of risky chains related to each type of privacy threat, under *direct exposure* and *implicit inference*, respectively. We analyzed three scenarios and reported the results in Figure 7: (1) a single sink with direct exposure, (2) a single sink with implicit inference, and (3) multiple sinks with implicit inference. The results of the three scenarios are reported in three groups (in the dashed squares) from left to right, respectively. In the first two groups, the right columns include privacy leaks within a single app (i.e., chains with length 1). Besides, the experiments

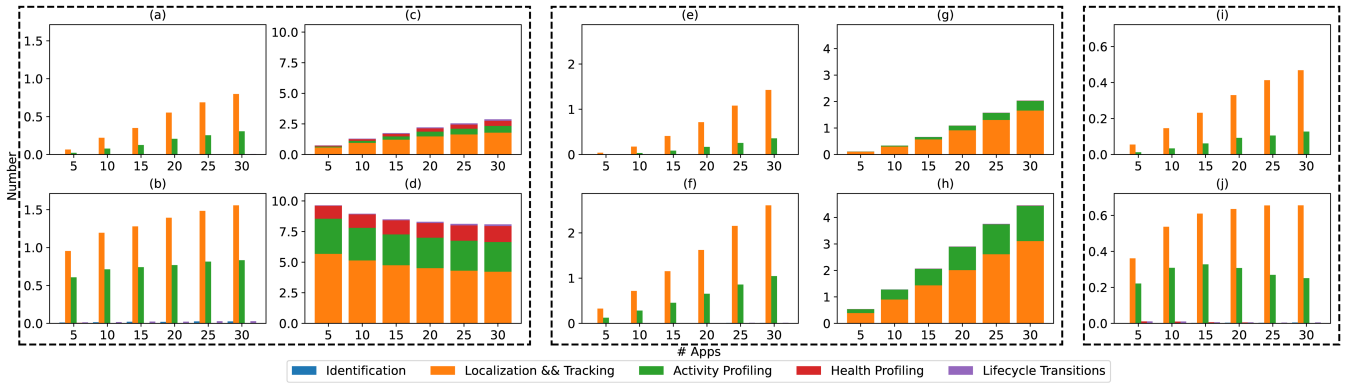

Figure 7: Average number of risky chains with different privacy threats.

ID	App Name	Device t	Device c	Device a
P_1	strobe when I am home and someone arrives	presence		alarm
P_2	turn on when door unlocks	lock		switch
P_3	its too hot	temperature		sink
P_4	ready for rain	door		sink
P_5	door open warning light	door		light
P_6	notify me when motion stops for more than 2 minutes	motion	illuminance	sink
I_1	turn on my hue light when I get out of bed	bed		light
P_7	**initial state event streamer**	**alarm switch**		**sink$_1$ sink$_2$**
I_2	**turn heater on by switch**	**switch**		**heater**
O_1	**turn off A/C after leaving home**	**presence**		**A/C**

Table 7: The IoT apps and devices used in the case study.

Case	Cross-App Chain	Connection[1]	Device(P_I) & Label	Leak[2]
L_1	$I_1 \rightarrow P_6$ $P_5 \rightarrow P_6$	IE IE	light(1):$\mathbb{B}$, $\mathbb{C}$1	SD
L_2	P_4 $P_5 \rightarrow P_6$	 IE	door($\frac{3}{4}$):$\mathbb{B}$	TF
L_3	$P_1 \rightarrow \mathbf{P_7}$ $P_2 \rightarrow \mathbf{P_7}$	MA MA	presence(1):$\mathbb{B}$ lock(1):$\mathbb{B}$	SD SD
L_4	$\mathbf{O_1} \rightarrow P_3$	IA	presence(1):$\mathbb{B}$ A/C(1):$\mathbb{B}$, $\mathbb{C}$1	SD
L_5	$P_2 \rightarrow \mathbf{I_2} \rightarrow P_3$	MA $\rightarrow$ IE	lock(1):$\mathbb{B}$ heater:$\mathbb{B}$, $\mathbb{C}$1	SD

[1] Match(M), Influence(I), Activate(A), Enable(E)
[2] Single-Sink(S), Multiple-Sink(T), Direct-Exposure(D), Implicit-Inference(F)

Table 8: The privacy leakages identified in the case study.

were taken under two settings, i.e., without and with *fat* apps. The top and bottom rows of Figure 7 show the results of the two settings, respectively.

The fat app significantly increases the number of risky chains in all five privacy threat categories. Among the identified threats, *Localization & Tracking* and *Activity Profiling* are prevalent, while detecting *Identification* remains challenging. This suggests that users should be aware of their location and activity patterns, including daily routines and habits. Without *fat* apps, the leakage due to *direct exposure* is relatively low. The privacy leakage due to *implicit inference* (Figures 7(e)-(f)) are notably higher than that from *direct exposure* (Figures 7(a)-(b)). Additionally, Figures 7(i)-(j) show that multiple-sink inference can also contribute to privacy leaks.

Similar to existing detection approaches, PrivacyGuard can detect privacy leaks in single apps. In the single-sink setting, counting single-app leakage, i.e., Figures 7(c), (d), (g), and (h), increases the number of chains with privacy leakage. Moreover, Figures 7(d) and (h) show that as the number of apps increases, a *fat* app is more likely to connect to other apps. This decreases the likelihood of a direct chain between two apps. It makes *direct exposure* more challenging but simplifies the *implicit inference*.

6.3 Privacy Leakage Analysis

To provide a detailed examination and demonstrate the effectiveness of PrivacyGuard, we present case studies using real-world IoT apps with typical devices such as alarms, doors, lights, locks, thermostats, motion detectors, and temperature sensors[9]. The apps listed in Table 7, along with their names and associated devices for each component, include P_1 to P_7 as SmartApps, I_1 and I_2 as IFTTT applets, and O_1 as an OpenHAB rule. P_1 to P_6 and I_1 are deployed in the original scenario, while P_7, I_2, and O_1 are intentionally used to induce leakage from the original scenario. We extract the TCA rules of each app and construct a chaining graph, shown in Figure 8. Figure 8(a) illustrates the chaining graph of the original scenario, while Figures 8(b) to (d) depict the chaining graphs of scenarios with intentional leakage. By constructing the TCA rules for each app, we found that P_4 has two branches that share the same action, which means that each trigger in both branches has an execution probability of 1/2. In contrast, the TCA components in the remaining apps have an execution probability of 1.

Table 8 shows the details of the identified privacy leaks in each case, including the case number, cross-app chain, chain connection type, privacy-sensitive devices with their inference probabilities, the privacy label if leakage exists, and the type of leak. First, we examine whether the original scenario in Figure 8(a) could potentially leak sensitive information. App P_4 is an individual app with a sink. When the sink is activated, the trigger *door.open* in P_4 has an inference probability of $\frac{1}{2}$, and no leakage is detected. Both apps I_1 and P_5 can form a chain through the physical channel *light-illuminance*, via an *influence-enable* relation with P_6. When P_6 sends a message, the trigger *bed.out* in I_1 and *door.open* in P_5 both have an inference probability of 1/2, without causing a privacy leak. However, the action *light.on* is executed deterministically, leading to a privacy leak categorized under *Localization & Tracking* and *Activity Profiling*, as shown in case L_1. In case L_2, since P_4 and P_5 share the same trigger, a multiple-sink implicit inference can increase the inference probability of *door.open* to 3/4, resulting in a privacy leak classified as *Localization & Tracking*.

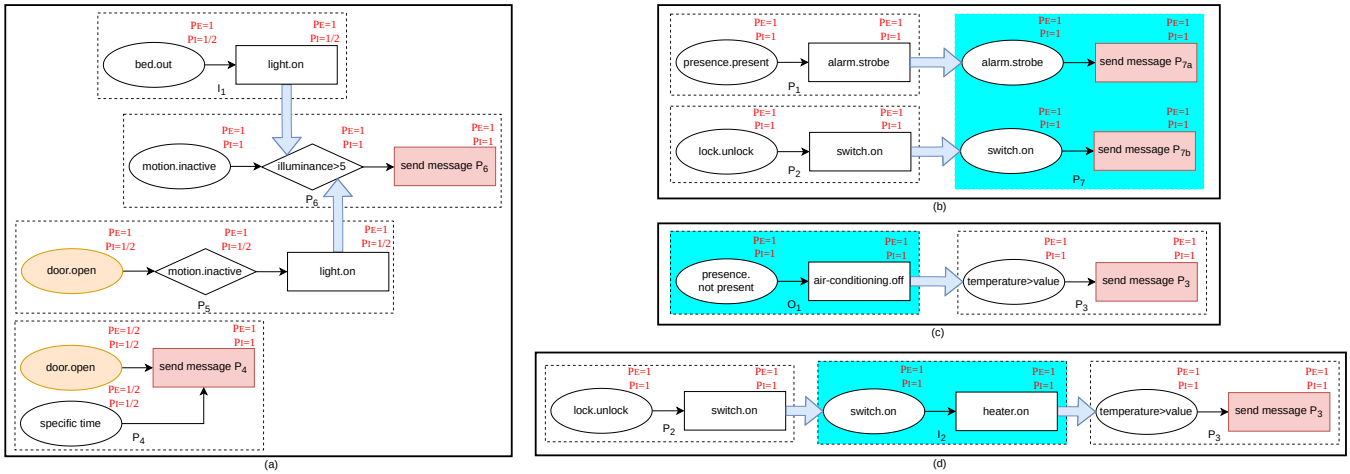

Figure 8: The chaining graphs of apps in the case study: (a) the original scenario; (b) malicious exit app; (c) malicious entry app; (d) malicious middle app.

We added malicious apps that intentionally leak privacy-sensitive information. Each malicious app P_7, I_2, and O_1 appears to be benign. In the original scenario P_1, P_2, and P_3 do not leak any information. However, when these apps are chained together, privacy leakage occurs. In case L_3 (i.e., Figure 8(b)), we deliberately include a *fat* app P_7, whose sinks are connected to P_1 and P_2 via a *match-activate* relation. In case L_4 (i.e., Figure 8(c)), turning off the air conditioning causes the temperature to exceed a threshold, which connects O_1 and P_3 through an *influence-activate* relation. In case L_5(i.e., Figure 8(d)), I_2 connects P_2 and P_3, acting as a bridge to leak information from the entry app P_2. In each case, the inference probabilities are 1 due to the direct exposure from a single sink, leading to the triggers *presence.present*, *presence.not present*, and *lock.unlock*, which leak information about *Localization & Tracking*. Additionally, the actions *A/C.off* and *heater.on* also disclose information related to *Localization & Tracking* and *Activity Profiling*.

6.4 System Performance

We conducted two performance evaluations: (1) processing time, which measures the one-time effort needed to extract TCA rules, and (2) calculation time, which evaluates the time required to build cross-app chains and generate privacy threat results for a set of apps. Processing time depends on the number and complexity of IoT apps. On average, processing takes 3,835.85 milliseconds per SmartApp, 0.23 milliseconds per IFTTT applet, and 225.54 milliseconds per OpenHAB rule. Calculation time averages about 7.00 milliseconds for 5 apps and 772.69 milliseconds for 30 apps. This overhead is reasonable for a smart home environment.

7 Related work

IoT Security and Privacy. Various aspects of IoT security have been studied in the literature, e.g., firmware security [17, 83], authentication [41, 75, 78], device identification [12, 52, 57], privileges [18, 31, 62], and access control [4, 37, 45, 63, 79]. [24, 25] discovered the attacker's ability to exploit physical channels, while [36, 58, 81] explore side-channel attacks. [32, 54] proposed mechanisms for information flow control, [44] introduce context-based permission systems, and [16, 67] studied policy enforcement. [8, 14] employ code analysis to trace sensitive information, [1, 50] explore privacy issues through traffic analysis, [6, 7] investigate compromised devices that leak sensitive data, [11, 39, 51, 66, 74] study the security and privacy risks associated with IFTTT rules.

Cross-App Risk Analysis. Research efforts have been dedicated to detecting cross-app risks in IoT environments. [15, 16, 21, 24] investigate cross-app interference through static code analysis of Smartapps, [40, 70] utilize NLP tools to perform conflict analysis in IFTTT recipes,[39] analyzes the privacy leakage of IFTTT chain, [2, 16, 77] identify conflicts in both Smartapps and IFTTT applets, and [68, 77] use model checker to discover conflicts between apps. In addition, [53] explores violations in cross-vendor interactions, [24, 25] investigates physical channel interference between apps.

Cross-app risks exist in Android apps, as they enable cross-app communication through Inter-Component Communication (ICC), which provides APIs for components to exchange data and reuse functions between different apps, introduces risks such as privacy leakage [46, 48, 59, 73, 80], privilege escalation [10, 23, 30, 38, 84], and collusion attacks [13, 26, 27]. Due to differences in platforms and ecosystems, the risk of cross-app exploitation in Android apps arises from the abuse of ICC APIs, existing studies have not been able to establish cross-app chains, making it difficult for attackers to glean information leakage from such chains. Additionally, Android apps primarily interact with sensors on smartphones, which limits the potential scope of attacks. [30] introduces the risk of an app with permissions performing a privileged task for a malicious app, [47] combines different apps into a single APK to perform cross-app analysis, [46, 49, 72] detect inter-component information leaks by static taint analysis, [23] hijacks a defective privileged app to forward attacking intents, [59] models ICC links and increases the privacy leak detection rates, [84] detects privilege escalation by control flow graph, [26, 27] build ICC maps and flow analysis to detect collusion attack, [73] explores the privacy risk in cross-app content sharing activities, [3] detects malicious inter-app communication activities in dynamically loaded code.

Tool Name	Platforms	Static Analysis	Dynamic Analysis	Multiple Apps	Privacy Categorization	Device Profile	Privacy Inference	Inference Probability
Soteria[15]	SmartThings	●	○	●	○	○	○	○
IoTGuard[16]	SmartThings,IFTTT	●	●	●	○	○	○	○
HomeGuard[21]	SmartThings	●	○	●	○	○	○	○
SAINT[14]	SmartThings	●	○	○	●	○	○	○
IoTWatch[8]	SmartThings	○	●	○	●	○	○	○
Surbatovich[66]	IFTTT	●	○	●	●	○	○	○
SafeChain[39]	IFTTT	○	●	●	●	○	●	○
PrivacyGuard	SmartThings,IFTTT,OpenHAB	●	○	●	●	●	●	●

Table 9: Comparison between PrivacyGuard and existing approaches for IoT privacy leakage detection. (●: include, ○: not include)

Comparison to Existing Works. Table 9 compares PrivacyGuard with the most relevant works. Surbatovich et al. [66] and Cobb et al. [22] build an information flow model to analyze integrity/secrecy violations. However, their work fails to consider actual attribute values and neglects the privacy threat that arises from a chain of safe rules. SafeChain [39] utilizes dynamic analysis and model checking to detect privilege escalation and privacy leakage between IFTTT rules. However, their approach works only online, doesn't cover all trigger-action leaks, ignores taint sinks, and limits chain length to 2 without handling multiple chains or measuring threat levels. All the studies mentioned use coarse-grained privacy labels.

SAINT [14] provides a static taint analysis tool to identify possible data-leak paths in individual SmartApps, considering five types of taint sources, including *device states*. IoTWatch [8] utilized NLP to classify text data into privacy labels, including device information and states. While some device states may lead to compromising inferences, not *all* of them are sensitive, and they only focus on individual apps, neglecting cross-app privacy leaks where an app is considered leakage-free if no sensitive data is transmitted out of its sinks. As a result, data-leak paths identified by SAINT and IoTWatch may contain several false positives.

Soteria [15], IoTGuard [16], and HomeGuard [21] investigated property violations caused by cross-app interactions in SmartApps. Similarly, we utilize static analysis on associated rules/chains, but our focus is on a different privacy leakage problem. In our scenario, two apps may not be chained together even if their *trigger* and *action* are associated with the same capability, such as a switch, because the switch can control devices at different sensitivity levels. Therefore, we extract text data such as descriptions, prompts, and variable names in static analysis to construct device context/usage, a factor not considered in their approaches.

None of the existing works provide a systematic categorization of device privacy, and they overlook privacy inference threats within app conditions. PrivacyGuard stands out as the only privacy analysis tool that comprehensively addresses privacy inference threats across different platforms and provides additional processing for probability-based inference in static analysis.

8 Discussion and Future Work

PrivacyGuard proposes finer-grained privacy labels, infers device profiles, provides inference probabilities to users, and identifies privacy leakage risks in cross-app chains. PrivacyGuard enhances user awareness of how data can be indirectly leaked through interactions between multiple apps, even if each app is trusted individually.

Finer-grained privacy labels offer detailed information, allowing users to understand specific privacy risks and make decisions about which apps or devices to use based on their privacy preferences and concerns. This ensures that users only share data they are comfortable with, reducing unnecessary exposure. Device profiles are essential for *multi-linking sensors/actuators* that may link to multiple devices, which might or might not access sensitive data. Users can use the *device profile generator* to select apps that align with their desired device usage, based on the textual data within the apps. Inference probabilities offer a confidence metric for assessing privacy risks. When included in privacy alerts, this metric helps users understand the potential risks in the interactions among a group of apps. Conversely, attackers can exploit inference probabilities to identify the most vulnerable cross-app chains and target those with the highest likelihood of leaking privacy-sensitive information.

Our current design uses text data from apps to identify device types and states, and we gather supported devices from different IoT platforms. However, some devices may not be included, leading to errors or unrecognized identification. Our method relies heavily on app text, making it easy for attackers to intentionally provide misleading information to deceive users. We plan to explore additional resources to address these limitations in the future. Additionally, we assume that devices are connected via a physical channel mapping, but factors such as location and environmental conditions affect their ability to interact. To accurately model cross-app chains in future work, we plan to explore the real physical interactions between devices. Finally, user participation is essential for accurately reflecting users' privacy preferences, we plan to integrate a user preferences module into our privacy label classification process, which will better address users' privacy needs in future work.

9 Conclusion

In this paper, we present PrivacyGuard, a static cross-app privacy leakage analysis tool, to detect privacy-sensitive information leakage caused by cross-app chains. We provide a systematic categorization of privacy-sensitive devices. We formalize the concept of cross-app privacy inferences and explain how such inferences can be drawn from cross-app chains, giving the inference probability. We implement the prototype of PrivacyGuard and evaluate its correctness and effectiveness using a large-scale dataset of real-world IoT Apps. Our results demonstrate that PrivacyGuard can effectively detect privacy leakage enabled by cross-app chains.

Acknowledgments

This work was supported in part by NSF IIS-2014552, DGE-1565570, and the Ripple University Blockchain Research Initiative. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. 2020. Peek-a-boo: I see your smart home activities, even encrypted!. In *ACM WiSec*.

[2] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis of interaction threats in iot systems. In *ISSTA*.

[3] Mohannad Alhanahnah, Qiben Yan, Hamid Bagheri, Hao Zhou, Yutaka Tsutano, Witawas Srisa-An, and Xiapu Luo. 2020. Dina: Detecting hidden android inter-app communication in dynamic loaded code. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2782–2797.

[4] Bayu Anggorojati, Parikshit Narendra Mahalle, Neeli Rashmi Prasad, and Ramjee Prasad. 2012. Capability-based access control delegation model on the federated IoT network. In *IEEE WPMC*. 604–608.

[5] Apple. 2023. Home Kit. https://www.apple.com/ios/home. [Online; accessed 31-August-2023].

[6] Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. 2017. Identifying counterfeit smart grid devices: A lightweight system level framework. In *IEEE ICC*. 1–6.

[7] Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. 2019. A system-level behavioral detection framework for compromised CPS devices: Smart-grid case. *ACM TCPS* (2019), 1–28.

[8] Leonardo Babun, Z Berkay Celik, Patrick McDaniel, and A Selcuk Uluagac. 2021. Real-time analysis of privacy-(un) aware IoT applications. *PETS* (2021), 145–166.

[9] Leonardo Babun, Amit Kumar Sikder, Abbas Acar, and A Selcuk Uluagac. 2022. The truth shall set thee free: Enabling practical forensic capabilities in smart environments. In *Proceedings of the 29th Network and Distributed System Security (NDSS) Symposium*.

[10] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering* 41, 9 (2015), 866–886.

[11] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If this then what? Controlling flows in IoT apps. In *ACM CCS*.

[12] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. 2018. Behavioral fingerprinting of iot devices. In *Proceedings of the 2018 workshop on attacks and solutions in hardware security*. 41–50.

[13] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 71–85.

[14] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *USENIX Security*.

[15] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *USENIX ATC*. 147–158.

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

[17] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[18] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2022. Practical Data Access Minimization in Trigger-Action Platforms. In *USENIX Security*.

[19] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. 2022. Delay wreaks havoc on your smart home: Delay-based automation interference attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 285–302.

[20] Haotian Chi, Qiang Zeng, and Xiaojiang Du. 2023. Detecting and Handling IoT Interaction Threats in Multi-Platform Multi-Control-Channel Smart Homes. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1559–1576.

[21] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2020. Cross-app interference threats in smart homes: Categorization, detection and handling. In *DSN*.

[22] Camille Cobb, Milijana Surbatovich, Anna Kawakami, Mahmood Sharif, Lujo Bauer, Anupam Das, and Limin Jia. 2020. How Risky Are Real Users'{IFTTT$}$ Applets?. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. 505–529.

[23] Biniam Fisseha Demissie and Mariano Ceccato. 2020. Security testing of second order permission re-delegation vulnerabilities in android apps. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 1–11.

[24] Wenbo Ding and Hongxin Hu. 2018. On the safety of IoT device physical interaction control. In *ACM CCS*. 832–846.

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

[26] Karim O Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G Ryder. 2018. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing* 19, 1 (2018), 90–102.

[27] Karim O Elish, Danfeng Yao, and Barbara G Ryder. 2015. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*. Citeseer.

[28] Mahmoud Elkhodr, Seyed Shahrestani, and Hon Cheung. 2012. A review of mobile location privacy in the Internet of Things. In *ICT and Knowledge Engineering*. IEEE.

[29] Sarah Eskens. 2016. Profiling the European Citizen in the Internet of Things: How Will the General Data Protection Regulation Apply to This Form of Personal Data Processing, and How Should It? *Available at SSRN 2752010* (2016).

[30] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission re-delegation: Attacks and defenses.. In *USENIX security symposium*, Vol. 30. 88.

[31] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *IEEE S&P*. IEEE, 636–654.

[32] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security*.

[33] Google. 2023. IoT Solutions. https://developers.google.com/iot. [Online; accessed 31-August-2023].

[34] Tianbo Gu, Zheng Fang, Allaukik Abhishek, Hao Fu, Pengfei Hu, and Prasant Mohapatra. 2020. Iotgaze: Iot security enforcement via wireless context analysis. In *IEEE INFOCOM*.

[35] Tianbo Gu, Zheng Fang, Allaukik Abhishek, and Prasant Mohapatra. 2020. Iotspy: Uncovering human privacy leakage in iot networks via mining wireless context. In *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE, 1–7.

[36] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. 2018. Do you feel what I hear? Enabling autonomous IoT device pairing using different sensor types. In *IEEE S&P*.

[37] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things (IoT).. In *USENIX Security*.

[38] Yi He and Qi Li. 2016. Detecting and defending against inter-app permission leaks in android apps. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–7.

[39] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. Safechain: Securing trigger-action programming from attack chains. *IEEE TIFS* 14, 10 (2019).

[40] Bing Huang, Hai Dong, and Athman Bouguettaya. 2021. Conflict detection in iot-based smart homes. In *IEEE ICWS*.

[41] Ben Hutchins, Anudeep Reddy, Wenqiang Jin, Michael Zhou, Ming Li, and Lei Yang. 2018. Beat-pin: A user authentication mechanism for wearable devices through secret beats. In *AsiaCCS*. 101–115.

[42] ifttt. 2023. IFTTT Website. https://ifttt.com. [Online; accessed 31-August-2023].

[43] Aminul Islam and Diana Inkpen. 2008. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 2, 2 (2008), 1–25.

[44] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. 2017. ContexIoT: Towards providing contextual integrity to appified IoT platforms.. In *NDSS*.

[45] Tam Le and Matt W Mutka. 2019. Access control with delegation for smart home applications. In *IoTDI*.

[46] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[47] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings 30*. Springer, 513–527.

[48] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2014. Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 388–397.

[49] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 229–240.

[50] Yuan Luo, Long Cheng, Hongxin Hu, Guojun Peng, and Danfeng Yao. 2020. Context-Rich Privacy Leakage Analysis Through Inferring Apps in Smart Home IoT. *IEEE Internet of Things Journal* (2020), 2736–2750.

[51] Kulani Mahadewa, Yanjun Zhang, Guangdong Bai, Lei Bu, Zhiqiang Zuo, Dileepa Fernando, Zhenkai Liang, and Jin Song Dong. 2021. Identifying privacy weaknesses from multi-party trigger-action integration platforms. In *ISSTA*.
[52] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. 2017. Iot sentinel: Automated device-type identification for security enforcement in iot. In *ICDCS*.
[53] Vasudevan Nagendra, Arani Bhattacharya, Vinod Yegneswaran, Amir Rahmati, and Samir Das. 2020. An intent-based automation framework for securing dynamic consumer iot infrastructures. In *The Web Conference*. 1625–1636.
[54] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. 2018. IoTSan: Fortifying the safety of IoT systems. In *ACM CoNEXT*. 191–203.
[55] TJ OConnor, Reham Mohamed, Markus Miettinen, William Enck, Bradley Reaves, and Ahmad-Reza Sadeghi. 2019. HomeSnitch: Behavior transparency and control for smart home IoT devices. In *WiSec*.
[56] openHAB. 2023. Open Source Automation Software for Home. https://www.openhab.org. [Online; accessed 31-August-2023].
[57] Jorge Ortiz, Catherine Crawford, and Franck Le. 2019. DeviceMien: network device behavior modeling for identifying unknown IoT devices. In *IoTDI*. 106–117.
[58] Eyal Ronen and Adi Shamir. 2016. Extended functionality attacks on IoT devices: The case of smart lights. In *IEEE EuroS&P*. IEEE, 3–12.
[59] Jordan Samhi, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2021. Raicc: Revealing atypical inter-component communication in android apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1398–1409.
[60] Samsung. 2023. SmartThings. https://www.smartthings.com. [Online; accessed 31-August-2023].
[61] Samsung. 2023. SmartThings Developers. https://developer.smartthings.com/docs/devices/capabilities/capabilities. [Online; accessed 31-August-2023].
[62] Faysal Hossain Shezan, Kaiming Cheng, Zhen Zhang, Yinzhi Cao, and Yuan Tian. 2020. TKPERM: cross-platform permission knowledge transfer to detect overprivileged third-party applications. In *NDSS*.
[63] Amit Kumar Sikder, Leonardo Babun, Z Berkay Celik, Abbas Acar, Hidayet Aksu, Patrick McDaniel, Engin Kirda, and A Selcuk Uluagac. 2020. Kratos: Multi-user multi-device-aware access control system for the smart home. In *ACM WiSec*. 1–12.
[64] sklearn. 2023. Agglomerative Clustering. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html. [Online; accessed 31-August-2023].
[65] spaCy. 2023. Trained Models & Pipelines. https://spacy.io/models. [Online; accessed 31-August-2023].
[66] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*.
[67] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *USENIX Security*.
[68] Rahmadi Trimananda, Seyed Amir Hossein Aqajari, Jason Chuang, Brian Demsky, Guoqing Harry Xu, and Shan Lu. 2020. Understanding and automatically detecting conflicting interactions between smart home IoT applications. In *ESEC/FSE*.
[69] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-level signatures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 2020.
[70] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action IoT platforms. In *ACM CCS*. 1439–1453.
[71] Zhaohui Wang, Bo Luo, and Fengjun Li. 2023. SmartAppZoo: a Repository of SmartThings Apps for IoT Benchmarking. In *IoTDI*. 448–449.
[72] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.
[73] Jiangrong Wu, Yuhong Nan, Luyi Xing, Jiatao Cheng, Zimin Lin, Zibin Zheng, and Min Yang. 2024. Leaking the Privacy of Groups and More: Understanding Privacy Risks of Cross-App Content Sharing in Mobile Ecosystem. In *NDSS*.
[74] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. 2019. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access* 7 (2019), 63457–63471.
[75] Zhenyu Yan, Qun Song, Rui Tan, Yang Li, and Adams Wai Kin Kong. 2019. Towards touch-to-access device authentication using induced body electric potentials. In *MobiCom*.
[76] Haoxiang Yu, Jie Hua, and Christine Julien. 2021. Analysis of ifttt recipes to study how humans use internet-of-things (iot) devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 537–541.
[77] Yinbo Yu and Jiajia Liu. 2022. Tapinspector: Safety and liveness verification of concurrent trigger-action iot systems. *IEEE TIFS* 17 (2022), 3773–3788.
[78] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, Deqing Zou, Hai Jin, and Yuqing Zhang. 2020. Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation.. In *USENIX Security*.
[79] Eric Zeng and Franziska Roesner. 2019. Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study.. In *USENIX Security*.
[80] Daojuan Zhang, Yuanfang Guo, Dianjie Guo, Rui Wang, and Guangming Yu. 2017. Contextual approach for identifying malicious Inter-Component privacy leaks in Android apps. In *2017 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 228–235.
[81] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1074–1088.
[82] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. 2018. User perceptions of smart home IoT privacy. *Proceedings of the ACM on HCI* 2 (2018), 1–20.
[83] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL:High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security*. 1099–1114.
[84] Xingqiu Zhong, Fanping Zeng, Zhichao Cheng, Niannian Xie, Xiaoxia Qin, and Shuli Guo. 2017. Privilege escalation detecting in android applications. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 39–44.
[85] Jan Henrik Ziegeldorf, Oscar Garcia Morchon, and Klaus Wehrle. 2014. Privacy in the Internet of Things: threats and challenges. *Security and Communication Networks* (2014), 2728–2742.

A Privacy Label Assignment

Here, we explain the rationale behind the privacy label assignment.

Identification. Identification denotes the potential risk of exposing the real identities of individuals. For instance, a camera in the *Monitoring* cluster can be used for security, monitoring, or photography. When cross-referenced with facial databases, the camera data poses a risk of face recognition. Similarly, access to audio clips or speech data of a person, such as *audioCapture* and *speechRecognition*, can be used to recognize an individual.

Localization and Tracking. When individuals are localized or tracked, the potential risks include stalking, harassment, and even physical harm. The *geolocation* feature, with attributes like latitude, longitude, and speed, has the potential to leak precise location information. Additionally, devices associated with *presence* can reveal information about home occupancy and user presence. Presence information can also be inferred from certain activities, for example, turning on the lights, taking a shower, or sleeping in bed strongly suggests that the user is at home, while driving a car indicates the user is away from home.

Activity and Health Profiling. Profiling data reveals the patterns and habits, which may potentially leak personal and private aspects of an individual's life. Analyzing activity data can lead to insights into an individual's behaviors and preferences. Everyday actions such as showering or going to bed might uncover one's daily routine. Similarly, the status of a car while driving could reveal a driver's habits, while cameras could broadcast a user's day-to-day activities. Devices associated with music could expose an individual's musical tastes, while fitness and health data could outline sensitive health conditions. These insights may be used for targeted advertising, influencing decisions, or even predicting future actions.

Lifecycle Transitions. When ownership of smart devices changes, there is a potential risk of privacy-sensitive information associated with the device being leaked. For instance, health data such as Body mass index (BMI) and sleep patterns collected by medical devices could reveal the health condition of the previous user. Similarly, location data stored in devices related to location services might

expose the address of the previous user. In addition, audio, video, and photos of the previous user could be unintentionally disclosed. Devices storing such data pose a risk of privacy violations due to the threat of lifecycle transition. As ownership shifts from one individual to another, the mishandling of data during this process can result in the compromise of sensitive information.

B Chain Generation Algorithm

The algorithm starts by choosing a sink rule and then constructs all possible chains that reach their maximum length, ending at this sink rule. It uses two stacks to keep track of the current rule and the chains being formed, and maintains a set of visited rules to avoid cycles. The process involves identifying adjacent rules that can either *activate* or *enable* the current rule, and continues until no more adjacent rules are available. This approach effectively builds chains that achieve the maximum length.

Algorithm 1: Cross-App Chain Generation

Input: a sink rule: *sink*
Output: list of chains: *chains*

```
Function getCrossAppChainsFromSink(sink):
    chains, visited ← ∅, ∅
    nodes, paths ← stack(), stack()
    nodes.push(sink)
    paths.push([sink])
    while not nodes.empty() do
        node = nodes.pop()
        path = paths.pop()
        if node ∈ visited then
            continue
        end
        visited += node
        adjacents ← getAdjacentRules(node)
        if adjacents.empty() then
            chains += path
            continue
        end
        foreach rule ∈ adjacents do
            nodes.push(rule)
            paths.push(path + rule)
        end
    end
    return chains
```