Detecting Smart Home Automation Application Interferences with Domain Knowledge

Tao Wang†‡, Wei Chen†‡*¶, Liwei Liu†‡, Guoquan Wu †‡*§, Jun Wei†‡*, Tao Huang†‡¶,
†*State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences, Beijing, China*
‡*University of Chinese Academy of Sciences, Beijing, China*
**Nanjing Institute of Software Technology, Nanjing, China*
§*Joint Laboratory on Cyberspace Security, China Southern Power Grid, Guangzhou, China*
†{wangtao19, wchen, liuliwei19, gqwu, wj, tao}@otcaix.iscas.ac.cn,

***Abstract*—Trigger-action programming (TAP) is a widely used development paradigm that simplifies the Internet of Things (IoT) automation. However, the exceptional interactions between automation applications may result in interferences, such as conflicts and infinite loops, which cause undesirable consequences and even security and safety risks. While several techniques have been proposed to address this problem, they are often restricted in handling explicit and simple conflicts without considering contextual influences. In addition, they suffer from performance issues when applying to large-scale applications.**

To address these challenges, we design an effective and practical tool KnowDetector with comprehensive domain knowledge to detect application interferences. To detect application interferences, KnowDetector constructs an automation graph with 1) events, conditions, and actions from automation applications, 2) vertices representing physical environment channels, and 3) edges derived from potential semantic relations between the vertices. In order to make the graph extensively capture the interactions between automation applications, we propose a knowledge model named KnowIoT that accurately characterizes IoT devices with command-level IoT services and the intricate relations between these services and the contextual environment. We abstract the interference detection into a graph pattern-matching problem and summarize ten application interference patterns of four types. Finally, KnowDetector can efficiently detect application interferences by searching for sub-graphs matching the patterns within the automation graph. We evaluated KnowDetector on three real-world datasets. The results demonstrated that it outperformed the other state-of-the-art tools with the highest precision, recall, and F-measure. In addition, KnowDetector is scalable to detect application interferences within a large number of applications with a minimal time overhead.

***Index Terms*—smart home platform, TAP, automation application interference, Internet of Things**

I. Introduction

The Internet of Things (IoT) is becoming increasingly widespread, particularly in smart homes. Smart home platforms connect IoT devices and provide users with programming frameworks to build home automation. Currently, trigger action programming (TAP) is the most popular paradigm supported by IFTTT [1], Samsung SmartThings [2], Home Assistant [3], OpenHab [4], Zapier [5], and others.

With TAP, smart home users can create event-driven automation applications of various forms, e.g., TAP rules and script-based applications, to facilitate their lives. A TAP rule basically contains a **trigger** and an **action** in the form of "IF a trigger occurs, THEN perform an action". For example, "IF the temperature rises to 26°C, THEN turn on the cooling mode of the air conditioner." A TAP application is similar despite its implementation in script code. Automation applications enable smart home platforms to react to their triggers and conduct the declared action automatically without users' manual operations. However, TAP's limited expressiveness makes it difficult for users to build large and complex automation systems [6], [7]. To fulfill the requirements of a scenario, the user needs to design multiple applications and coordinate them to accomplish complex tasks. Nevertheless, the IoT devices have some intrinsic features, e.g., implicit dependencies between device services and implicit interactions through physical channels. These can lead to undesired interactions between automation applications, resulting in exceptions and even security and safety risks [8], [9]. In this paper, we focus on the interferences caused by automation application interactions and regard them as **automation application interferences** (application interference for short).

Recent years have seen many studies on detecting and resolving application interferences in smart home systems. However, some problems still need to be addressed further.

First, some works are policy-based, such as RemedIoT [10] and IoTGuard [11], but they are restricted in handling some explicit and simple conflicts between automation applications without considering contextual influences. Specifically, these approaches are insensitive to time, physical environments (also known as physical channels), and IoT service relations.

Second, the approaches based on formal methods usually suffer performance and scalability issues. These works formally specify constraints and properties and exploit model-checking techniques, e.g., SMT [8], [9], [12], LTL [13], [14], and others [15], [16], [17], to detect violations. However, such approaches must handle large state spaces, significantly affecting their performance and scalability. For instance, the model checking-based approaches take at least 30 minutes to check one application pair [15], which is unbearable in the production platforms. These approaches also lack of modeling IoT services impact on physical channels and relations between IoT services.

¶Wei Chen and Tao Huang are the corresponding authors.

Third, implicit dependencies among device services and physical channels can cause many interferences. Detecting such interferences requires comprehensive domain knowledge. Several studies have proposed graph- and knowledge-based techniques [18], [17]. However, these existing works only concern a proportion of domain knowledge, making them unable to adequately model various entities and their potential relations involved in complex scenarios, such as IoT device services and their influence on environmental contexts and implicit dependencies between device services. For example, an air conditioner (AC) does not always decrease the room temperature unless it works under the cooling mode; besides, the cooling mode of an AC also implies that it is switched on. As a result, these works cannot detect application interferences comprehensively and accurately.

Consequently, a practical approach is required to detect application interferences at scale and to be extensible to handle various interferences. The scalability enables the approach to be adequate to detect interferences among a large number of automation applications within a short time, and the extensibility enables the approach to detect newly emerging interferences through simple configurations.

To this end, we design *KnowDetector*, a tool for detecting smart home application interferences with comprehensive domain knowledge. Overall, KnowDetector abstracts interference detection into a problem of graph pattern matching. It parses a set of automation applications into an ECA graph that contains events, conditions, and actions of the applications. Afterward, KnowDetector transforms the graph into an automation graph by adding vertices representing physical environment channels and edges derived from potential semantic relations between the vertices. In order to make the graph extensively capture the interactions between automation applications, we propose a knowledge model named *KnowIoT* that accurately characterizes IoT devices at a fine granularity. KnowIoT includes details such as the command-level IoT services and the intricate relations between these services and the contextual environment. Besides, we summarize four types of application interferences and propose 10 interference patterns in the form of sub-graphs. Subsequently, KnowDetector traverses the automation graph and searches for sub-graphs matching the patterns. In addition, we improve the detection accuracy by analyzing whether two potentially interfered applications can execute simultaneously based on their conditions.

To evaluate the accuracy, effectiveness, and performance of KnowDetector, we evaluated it on three real-world datasets. The results demonstrated its ability to detect application interferences accurately. It detected 312 interferences from 5000 IFTTT applications. Furthermore, we compared the accuracy of KnowDetector with the other state-of-the-art tools using a publicly available benchmark. Our tool outperformed the others with the highest precision, recall, and F-measure. Additionally, KnowDetector has exceptional detection efficiency with a minimal overhead. It can accurately identify interference from one thousand applications in less than a second.

We summarize our contributions as follows.

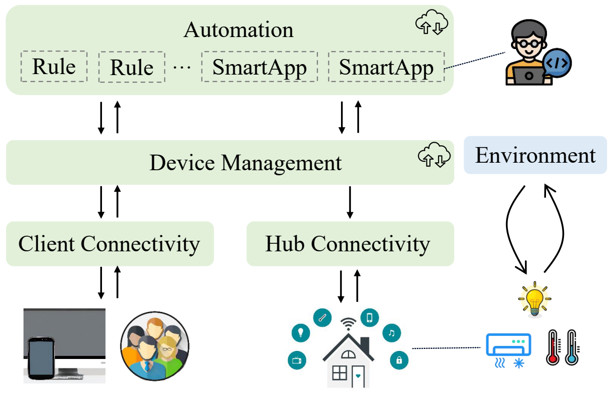


Fig. 1. The cloud-centric architecture of smart home platforms.

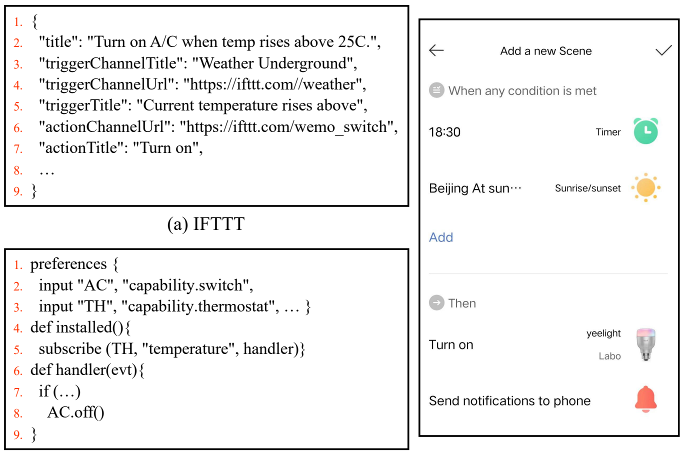
- We propose a fine-grained knowledge graph model, KnowIoT. KnowIoT captures implicit dependencies, such as implications and exclusions, between IoT services, as well as service impacts on physical channels.
- Based on KnowIoT, we propose KnowDetector, a tool for detecting application interferences. KnowDetector can detect four types and ten patterns of interferences, including conflict, loop, redundancy and blocked action.
- We evaluate KnowDetector on real-world datasets, including IoTBench [19], IFTTT applets, and automation applications from a commercial smart home platform. KnowDetector outperforms other state-of-the-art techniques and can successfully detect interferences in real-world automation applications.
- We provide the tool and data at https://github.com/tcse-iscas/KnowDetector [20] for evaluating application interferences, which contains more complex cases occurred in practical scenarios.

The rest of this paper is organized as follows. Section II gives an overview of smart home platforms and automation. Section III presents our knowledge graph model and its construction process. Section IV elaborates on our detection methodology. Section V evaluates our tool on its effectiveness, accuracy, and performance. Section VI discusses the limitations. Section VII presents the related work, and finally, Section VIII concludes this paper.

II. Background

A. *Smart Home Platforms*

Many home platforms, especially commercial ones, are usually based on cloud-centric architecture. Figure 1 presents a brief view. A platform contains three main components, i.e., connectivity, device management, and automation. The devices are connected to the platform through hub connectivity, and users can control the devices through client connectivity provided by the platform. The device management component controls devices, and the automation component provides home automation applications for users. Homeowners can create custom rules and install third-party automation applications to make their homes smarter.



```
{
  "title": "Turn on A/C when temp rises above 25C.",
  "triggerChannelTitle": "Weather Underground",
  "triggerChannelUrl": "https://ifttt.com//weather",
  "triggerTitle": "Current temperature rises above",
  "actionChannelUrl": "https://ifttt.com/wemo_switch",
  "actionTitle": "Turn on",
  …
}
```

(a) IFTTT

```
preferences {
  input "AC", "capability.switch",
  input "TH", "capability.thermostat", … }
def installed(){
  subscribe (TH, "temperature", handler)}
def handler(evt){
  if (…)
    AC.off()
}
```

(b) SmartApp

Add a new Scene

When any condition is met

18:30 Timer

Beijing At sun··· Sunrise/sunset

Add

Then

Turn on yeelight Labo

Send notifications to phone

(c) Configuration interface

Fig. 2. Automation applications of various forms.

Smart home platforms usually use an event-driven execution model, allowing multiple automation applications to run concurrently. For instance, consider the automation application named *Lock-It-When-I-Leave*, which attempts to lock doors once it detects someone leaving. In this scenario, the application subscribes to events produced by motion sensors within the device management component. The smart home hub, located in the connectivity layer, relays messages from local devices to the cloud. The cloud then determines whether particular events have occurred and executes corresponding automation applications accordingly.

B. Automation

Smart Home automation interconnects devices and systems in a household, automating the control of home appliances, home scenarios, and home security. Automation applications follow the TAP model.

Trigger action programming. TAP provides an intuitive abstraction for non-technical users to control IoT devices. The TAP model consists of events, conditioners, and actions. *Events* can be device service executions or state changes that occur in a moment, e.g., motion is detected. *Conditions* refer to the constraints on physical channels (e.g., temperature $>26°C$, time range, location). *Actions* consist of updates or invocations on the state of one or more devices, often resulting in an actuation like activating a light switch. Note that, an IoT service can be either regard as an event or an action.

There are several ways to achieve automation in smart homes. The three popular methods are IFTTT (If This Then That) [1], the Samsung SmartThings SmartApp [21], and the configuration interface provided by smart home platforms [2], [22], [23]. Figure 2 shows the automation applications format. For convenience, we will collectively refer to them as home automation applications.

IFTTT. IFTTT is a web-based service that allows users to create applets or automated workflows between different services, applications, and devices. For example, a user can set up an applet that triggers the lights to turn on automatically when the user arrives home, using location data from their smartphone as the trigger. IFTTT supports hundreds of services and devices, including smart home devices, social media platforms, and cloud storage services.

SmartApp. Samsung SmartThings SmartApp is a popular solution for smart home automation. SmartApp enables users to create advanced rules and scenarios by combining multiple devices to execute tasks. Compared to IFTTT, SmartApp has more expressive and descriptive power. It supports multiple triggers, multiple actions, and scheduling functions like setting the specific time for executing tasks. For instance, a user could set up an automation that turns off all the lights in three minutes when they leave the house.

Configuration interface. Some smart home systems provide users with a configuration interface to create custom automation by specifying triggers, conditions, and actions. This approach demands more technical expertise than services like IFTTT, but it offers greater control over the automation applications.

In summary, smart Home automation empowers users to remotely monitor and control their homes, delivering enhanced convenience, comfort, and security.

C. Automation Application Interaction

Different automation applications can interact through IoT devices and physical channels. There are two main ways of application interaction based on IoT devices, i.e., direct and indirect. Two applications, denoted a_1 and a_2, may interact in the following manners.

Direct interaction. a_1 and a_2 interact through device services. The interaction can occur via the same or different device services. For example, when the door is opened, a_1 may activate a light, and while the light is on, a_2 may start a video. In this case, the execution of a_1 can trigger a_2. This is an example of an interaction based on the same device services.

Another example is an air conditioner, which has two services: *turn-on* and *mode-heat*. When the air conditioner is in heating mode, it is also switched on. We refer to these as ***implied dependencies***: dependencies that exist between services. An ***exclusive dependency*** between device services is also possible, as seen in a phone's camera and flash. It is impossible to enable the both simultaneously. These two cases are interactions based on different device services.

Indirect interaction. The interaction between a_1 and a_2 is facilitated through physical channels. Indirect interaction occurs when the execution of a_1 physically satisfies the trigger of a_2, implying a physical relationship between their actions. For instance, if a_1 executes the device service *mode-cool*, which leads to a drop in temperature, and another application, a_2, has a trigger like *"when the temperature drops below 20°C"*, then an indirect interaction between a_1 and a_2 exists. This happens because the execution of a_1 affects the physical environment and triggers a_2. The common physical channels for such interactions are temperature, humidity, motion, leakage, location, smoke, and luminance.

Application interference. The interactions between applications can involve more than two applications. Multiple applications can be chained through the direct and indirect ways discussed earlier. These complex relationships can lead to interferences, where the enforced cooperation of applications yields unexpected results. Detection of application interference requires consideration of several factors, such as the relationships between device capabilities and their impact on the physical environment. Comprehensive domain knowledge is necessary for detecting diverse instances of interference.

III. Knowledge Graph

We propose a knowledge model to describe (1) IoT device capabilities and implicit dependencies between IoT services and (2) IoT services' impact on physical channels. After that, we present how to construct a knowledge graph based on the model.

A. Knowledge Model

As Figure 3 shows, the model has six elements, i.e., device, capability, attribute, command, service, and physical channel. We mainly focus on the relationships between different device services and the impact of corresponding device services on physical channels.

Currently, most smart home platforms manage devices in similar ways. For simplicity, we detail the device management with SmartThings [2]. In the context of device management, SmartThings employs the concepts of capabilities, attributes, and commands. Specifically, each device is assigned one or more capabilities, with each capability being associated with one or multiple attributes and commands. For instance, a smart light bulb has two distinct capabilities: switch and color. The switch capability enables external applications to manipulate bulb status through the commands *on()* and *off()*. On the other hand, the color capability comprises three attributes - color, hue, and saturation - each of which can be fine-tuned via the commands *setColor*, *setHue*, and *setSaturation*, respectively. We have built our knowledge graph based on the model mentioned above, with some extensions such as "service" and "physical channel". Additionally, we have also incorporated relationships between services and the impact of services on physical channels. Table I provides several examples. It is noteworthy that the division and definition of these capabilities are device-independent, thus ensuring their generalities across various IoT devices.

Device, capability, attribute, and command. Each device has one or more capabilities, and each capability has one or more associated attributes and commands. This follows the common device model applied by most smart home platforms.

IoT service. An IoT service is specified by a device capability and the corresponding command, which captures a specific device functionality. Specifically, an IoT service S_i is represented as a tuple $S_i = <Did, Cap, Cmd>$, where Did denotes the IoT device, Cap denotes the IoT capability, and Cmd denotes the invoked device command of the capability.

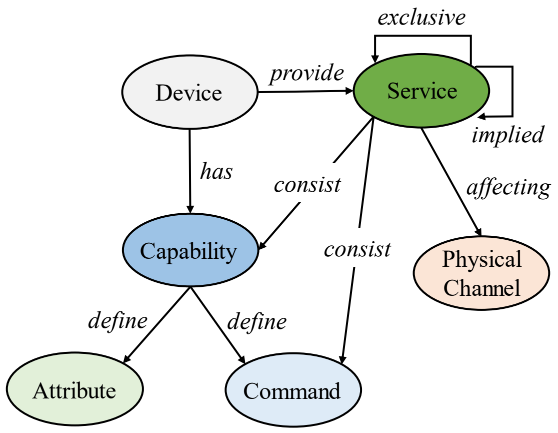


Fig. 3. Knowledge Model

TABLE I
EXAMPLES OF SOME PRE-DEFINED CAPABILITIES, COMMANDS, ATTRIBUTES AND SERVICES

Capability	Command	Attribute	Service

switch	on()
	off()
		state.on
		state.off
			switch.on
			switch.off

mode	heat()
	cool()
	auto()
		state.heat
		state.cool
		state.auto
			mode.heat
			mode.cool
			mode.auto

alarm	off()
	strobe()
	siren()
	both()
		state.off
		state.strobe
		state.siren
		state.both
			alarm.off
			alarm.strobe
			alarm.siren
			alarm.both

For example, the heating service of an air conditioner can be expressed as $<AC, mode, heat>$.

Physical channel. We focus on seven physical channels in our knowledge graph, i.e., temperature, humidity, motion, leakage, location, smoke, and luminance.

Relation. Our knowledge model defines a set of relations, with a specific focus on three of them: *exclusive*, *implied*, and *affecting*. We use *exclusive* and *implied* relations to capture the implicit dependencies between IoT services mentioned in Section II-C. The *affecting* relation captures the impact of an IoT service on physical channels. This relation is further subdivided into two categories: *increasing* and *decreasing*, depending on whether the corresponding physical channels are positively or negatively impacted.

B. Knowledge Graph Construction

Based on the defined model, we acquire knowledge and construct our knowledge graph using the large language model technique.

Acquiring knowledge. Large language models (LLMs) [24], [25], [26], such as ChatGPT, have garnered significant attention from both academia and industry. These models rely on large-scale text corpora, combined with reinforcement learning from human feedback, to achieve superior capabilities in language understanding, generation, interaction, and reason-

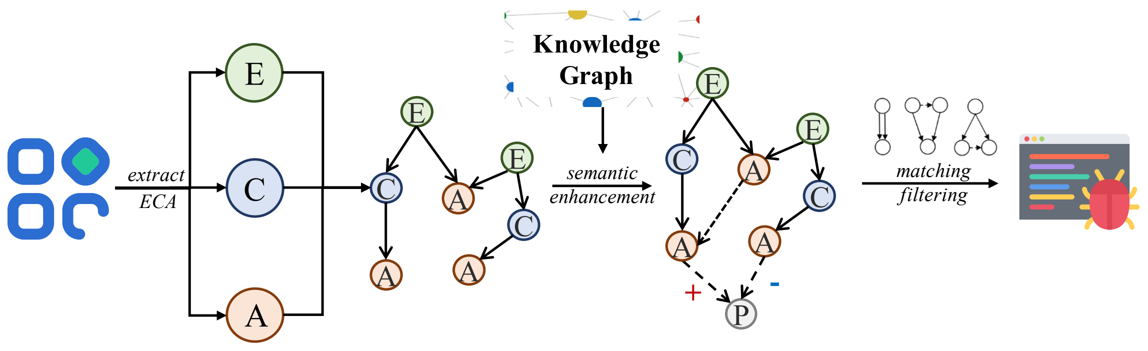


Fig. 4. The overall process

ing. We acquire IoT device functionality descriptions through *gpt-3.5-turbo* of the GPT models, which is publicly accessible through the OpenAI API [27].

We first use the *gpt-3.5-turbo* with a step-by-step input prompt to produce the initial corpus. We manually fine-tune the prompt to prompt the LLM to provide a description of the device's capabilities. Here is the input we used for debugging: "Now we assume that each IoT device has its capability, command, attribute, and service division. You can refer to SmartThings' management of device capability. And service is a combination of capability and command. Now I will give you the name of an IoT device, and you need to give me the division functions of this device and the impact of different services and physical channels." Next, we extract content from the functional description of the same device and construct the knowledge graph. We employ natural language processing (NLP) techniques to extract the relevant information. Our preprocessing steps involve *stop word removal, lemmatization,* and *entity linking*. We use an industrial-strength NLP tool called *spaCy* [28] to filter out stop words. Lemmatization is used to reduce the inflectional forms of a word to a common base form. For example, "cooling, cools, and cooled" are converted to the base word "cool" which helps us identify IoT devices and their services more accurately. Finally, we perform entity linking on these processed texts by matching them with pre-defined model entities (capability, attribute, command, and service). The impact of the service on the environment is mainly obtained by describing the mentioned physical channel and our keyword measurement of the impact of positive and negative feedback.

With the knowledge acquired automatically, we finally had three authors cross-validate it to confirm its correctness. We have also supplemented and revised its content based on common sense.

IV. Methodology

A. Overall Process

We present an overview of our methodology in Figure 4, which involves three main steps: (a) application parsing,

```
{
 "trigger":{
   "conditions": [{
     "conditionType": "time",
     "params": {
       "year": "",
       ...
       "day": "everyday",
       "timeRange": {
         "startTime": "10H0",
         "endTime": "12H0"
       },
   ...
```

```
"events": [{
     "devType": "01E",
     "prodId": "148M",
     "capabilityId": "switch",
     "params": {
       "deviceId": "light1",
       "Command": "on",
     }}],}, // trigger-end
  "actions":[{
    "devType": "00A",
    "prodId": "001T",
    "capabilityId": "audioPlayer1",
    "input": {
      "deviceId": "Audio",
      "Command": "playMusic#001",
    ...
```

Fig. 5. Key elements of commercial dataset in JSON format

(b) automation graph construction, and (c) pattern matching. Firstly, we parse the set of automation applications and extract their corresponding elements, including events, conditions, and actions. Based on the control relationships among these elements, we construct the ECA graph. Subsequently, we employ knowledge graphs to enhance the semantics of the ECA graph by incorporating information regarding the relationships between device services and the impact of services on the physical channels. This step outputs an automation graph. Next, we identify patterns of interference that arise from interactions between automation applications and categorize them accordingly. Finally, we perform a filtering analysis to remove false-positive cases.

B. Application Parsing

This step extracts three key types of elements, i.e., *events, conditions,* and *actions,* from automation applications. We illustrate the process with two subjects: IFTTT recipes and real-world data from a commercial smart home platform.

Parse IFTTT applications. We extract the following information for IFTTT applications and map them into the ECA graph: (1) Application title and description, which provides a general description of the application; and (2) trigger and action fields, which identify the events that trigger the application and the actions performed by the corresponding devices. Since IFTTT applications are defined through natural language, we use NLP techniques to extract the relevant information, including *stop word removal, lemmatization,* and *entity linking*. We perform entity linking on these processed texts by matching device names and IoT services. We also use the trigger and action fields to help us further identify the extracted triggers and actions. For complex conditional judgments, such as "when the temperature rises to 26°C", we use regular expressions to extract the necessary information.

Parse commercial data. We processed and decrypted the commercial dataset and finally formed the automation applications in JSON format. For instance, Figure 5 illustrates the core elements of the application "Every morning from 10:00 to 12:00, if I turn on the light1, play music 001". The dataset provides a clear definition of the conditions, events, and actions involved in each application. Therefore, by conducting a simple analysis, one can easily obtain information regarding the device type, product ID, device ID, capability, command, and parameters associated with each element.

C. Automation Graph

The automation graph is denoted as $AG = (V_s, V_p, V_c, E_c, E_e, E_i, E_x)$, where

- V_s: a set of *IoT service vertices*. An IoT service vertex is uniquely denoted by the concatenation of an IoT service S_i, where $S_i = <Did, Cap_i, Cmd_i>$ as we defined before. For example, an IoT service vertex, "$AC_1.switch.on$", represents the switch-on service of the device AC_1. Notably, a V_s can semantically represent either a trigger or an action.
- V_p: a set of *physical channel vertices*. A physical channel vertex represents a type of physical context existing in smart home platforms. We consider seven physical channels as discussed before. The vertex ID is set as its channel name. These vertices can be used to capture the impact of IoT services on physical channels.
- V_c: a set of *condition vertices*. A condition vertex is a node that represents the constraints on physical factors (seven physical channels and time) of an application. The condition vertex is defined as <env, op, args>. For example, the condition "when temperature rises up to 25°C" can be denoted as <temp, >, 25 >.
- E_c: a set of *control edges*. A control edge is a directed edge that flows between vertices in $V = V_s \cup V_p \cup V_c$, representing the control relationship. For simple applications without conditions, the sources of the control edges are triggers. The targets of the control edges are the actions. Supposing the trigger and the action are IoT services, the application can be expressed as $V_{s_t} \rightarrow V_{s_a}$. The application with condition can be expressed as $V_{s_t} \rightarrow V_c \rightarrow V_{s_a}$
- E_e: a set of *affecting edges*. Each effect edge is a directed edge that flows from IoT service vertices to the physical channel vertices, which can be expressed as $V_s \xrightarrow{eff} V_p, eff \in \{hasIncreased, hasDecreased\}$. The effect edge represents the impact of an IoT service on the corresponding physical channels. The value *hasIncreased* indicates the IoT service has a positive or incremental impact on a specific physical channel, and the other value *hasDecreased* is the opposite.
- E_i: a set of *implied dependency edges*. Each implied dependency edge is a directed edge between IoT services. An implied dependency edge $V_{si} \xrightarrow{imp.} V_{sj}$ represents that the execution of service V_{si} also implies the execution of V_{sj}. For example, <AC, mode, cool>$\xrightarrow{imp.}$ <AC, switch, on>represents that the AC in cool mode indicates the AC is also switched on.
- E_x: a set of *exclusive dependency edges*. Each exclusive dependency edge is a directed or bidirectional edge connecting two IoT services. This kind of edges represent two IoT services that cannot be executed simultaneously. For instance, we found that the camera and flash of the phone could not be turned on at the same time, which can be expressed as <Phone, flashlight, on>$\xrightarrow{ex.}$ <Phone, camera, on>.

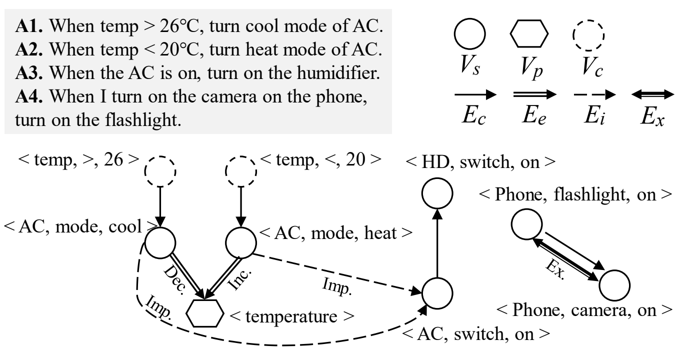

A1. When temp > 26°C, turn cool mode of AC.
A2. When temp < 20°C, turn heat mode of AC.
A3. When the AC is on, turn on the humidifier.
A4. When I turn on the camera on the phone, turn on the flashlight.

Fig. 6. A simple automation graph

Example. Figure 6 shows a simple automation graph of four applications. The four applications are within the gray box in the upper left corner of the figure. The circled nodes represent V_s, the IoT services. The dotted circle nodes represent V_c, the conditions. Hexagonal nodes represent V_p, the physical channels. For A1, there is a V_c pointing to V_s with the content <temp, >, 26>→ <AC, mode, cool>. The service <AC, mode, cool>can decrease the temperature. Therefore, there exists a double-lined edge E_e from <AC, mode, cool>to the vertex V_p $temperature$ i.e., <AC, mode, cool>$\Longrightarrow$ <temperature>. When A1 or A2 is executed, it will lead to the execution of A3. The dotted edge E_i $\dashrightarrow$ characterizes the implicit dependency between them.

D. Semantic Enhancement

After extracting event, condition, and action information from automation applications, we can construct a basic ECA control flow graph that includes fundamental control relationships. For example, in Figure 6, application *A1* includes two nodes: *<temperature, >, 26>*→ *<AC, mode, cool>*. Next, we enrich the graph's semantics based on our knowledge

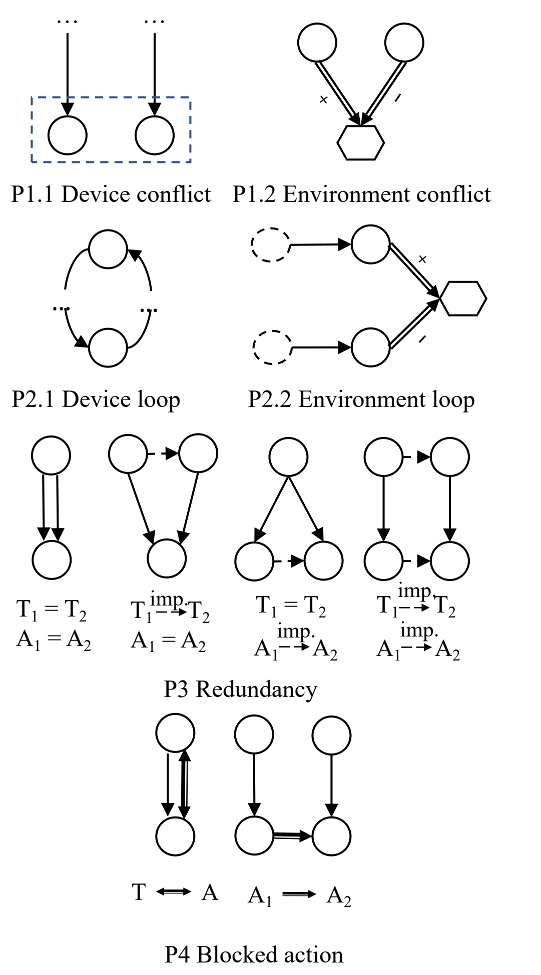

P1.1 Device conflict
P1.2 Environment conflict
P2.1 Device loop
P2.2 Environment loop
$T_1 = T_2$ $A_1 = A_2$
$T_1 \xrightarrow{imp.} T_2$ $A_1 = A_2$
$T_1 = T_2$ $A_1 \xrightarrow{imp.} A_2$
$T_1 \xrightarrow{imp.} T_2$ $A_1 \xrightarrow{imp.} A_2$
P3 Redundancy
$T \longleftrightarrow A$
$A_1 \longrightarrow A_2$
P4 Blocked action

Fig. 7. Automation application interaction interference pattern

graph. This step primarily enhances two types of information: the relationships between device services and the impact of device services on the physical channels. To begin, we create all physical channel nodes and then traverse the IoT service nodes in the graph. We construct relationship edges between the IoT service nodes and their corresponding physical channel nodes, as well as the edges with implicit dependencies between services. For example, the <AC, mode, heat>$\xrightarrow{imp.}$ <AC, switch, on>in the Figure 6 is added through the semantic enhancement process.

E. Pattern Matching

We present a taxonomy of automation application interaction interferences that we have summarized from previous works and encountered in real-world scenarios. For each type of interference, we provide a clear definition and identify its corresponding graph pattern. We summarize ten interferences patterns of four types in Figure 7. We detect interferences by matching these sub-graphs from constructed automation graph.

P1. Conflicting action. Conflicting actions can arise when different applications prescribe opposite actions of the same device or when device actions have opposite effects on the same physical channel.

P1.1 Device conflict. Conflicting actions occur when multiple applications dictate different actions of the same device. The pattern reveals that services with different commands under the same capability can conflict. For example, the following two applications pertain to a device conflict:

<door, switch, on>→ <light, switch, on>
<TV, switch, on>→ <light, switch, off>

P1.2 Environment conflict. Two actions may have opposite effects on the same physical channel. We refer to this pattern as *environment conflict*. Specifically, the values on *effect edges* E_e that point to the same physical channel are mutually exclusive. For example,

<temperature, >, 26>→ <AC, mode, cool>$\xrightarrow{-}$ *<tmp.>*
<window, switch, off>→ <heater, switch, on>$\xrightarrow{+}$ *<tmp.>*

P2. Infinite loop. Actions from multiple applications can cyclically trigger each other infinitely. We categorize this as the *infinite loop* and subdivide it into two sub-categories.

P2.1 Device loop. The device loop is formed directly based on IoT services. For example,

<light, switch, on>→ <light, switch, off>
<light, switch, off>→ <light, switch, on>

As such, our focus is on detecting whether a loop exists in the graph.

P2.2 Environment loop. The environment loop is caused by the impact of IoT services on physical channels. Consider the following two applications:

<temperature, >, 26>→ <AC, mode, cool>
<temperature, <, 20>→ <AC, mode, heat>

It is evident that the air conditioner (AC) will alternate between executing cooling and heating commands to maintain the temperature within the range of 20-26 degrees. To address this interference, we analyze whether behaviors that affect the same physical channel can satisfy each other's conditions. Specifically, when the AC is in cooling mode, it lowers the temperature below 20 degrees, which triggers the heating mode. The heating mode then raises the temperature up to 26 degrees, triggering the AC to enter cooling mode again.

P3. Redundancy. Applications can be redundant because they are the same, or there is an *implied dependency* relation between their triggers (or actions). Redundant applications can trigger certain device behaviors multiple times. As shown in Figure 7, there are four redundancy patterns. Detecting this type of interference involves analyzing the *implied dependency* relations between nodes.

P4. Blocked action. Nodes may have a mutual exclusion relationship, resulting in applications being unable to execute. This type of interference is known as *blocked action* interference, which can occur even within a single application. An example is the case where the flashlight cannot be turned on after turning on the camera on the phone. Moreover, blocking issues can also arise between multiple applications. For instance, one application r_i that cuts down the water supply may block another application r_j that relies on the water supply.

Filter analysis. To minimize false positives introduced by static analysis, we employ filter analysis techniques. We examine the temporal conditions between applications where interaction interferences are detected and remove applications that cannot co-occur. For instance, if one application occurs exclusively at night while another occurs during the day, it is impossible for both to run at the same time, and one of them can be safely removed.

V. Evaluation

In this section, we comprehensively evaluate our tool KnowDetector from multiple aspects. We tackle several important questions to provide a thorough assessment, including:

- **RQ1:** How well does KnowDetector perform in practice? Can it find common interaction interferences in real-world applications?
- **RQ2:** Can KnowDetector be extended to detect specific interference compared to other state-of-the-art techniques? What is the overall accuracy of KnowDetector?
- **RQ3:** What is the performance of KnowDetector?

Dataset. Our evaluation uses three distinctive sets of data:

(1) IoTBench [19] is a well-known public benchmark that provides a collection of automation applications. This resource has been adopted as the ground truth for interference detection by several previous studies [17], [29], [16], [30].

(2) A public dataset of IFTTT applets [31], from which we selected 5000 applets to assess the effectiveness and efficiency of KnowDetector.

(3) Through collaboration with an anonymous company, we obtained some real automation applications that cover a broader range of device usage scenarios. These applications were parsed, desensitized, and made available as benchmarks for future work on detection and verification.

To conduct our experiments, we used a laptop with an 8-core 3.2GHz AMD Ryzen 7 processor and 16 GB of RAM.

A. RQ1. Real-world Applications

We evaluate the capability of KnowDetector to identify interferences in real-world applications. We evaluate KnowDetector on 5,000 IFTTT applets and 120 automation applications collected from a commercial smart home platform.

IFTTT dataset. We have built a collection of devices and sensors that are commonly utilized in automation applications. The devices include air conditioners, heaters, thermostats, lights, switches, locks, door controls, alarms, and humidifiers. The collection also includes a variety of sensors that can perceive temperature, humidity, smoke, motions, and other physical contexts. Finally, we randomly selected 5,000 IFTTT applets that use these devices and sensors for further evaluation.

Results on IFTTT. Table II illustrates how the detected interferences were distributed among the four interference patterns presented in Figure 7. Interferences, particularly redundancy (P3) and device conflict (P1.1), are the most common issues. The prevalence of redundant cases can be attributed to the fact that different users contribute to the creation of applications, resulting in an increased number of redundant applications compared to those generated by a single user. The redundant applications often can result in confusion and inefficiencies during smart home system operations. Device and environment loops, which make up 16% of interferences, can pose significant challenges. These loops occur when devices are affected by physical channels such as temperature or humidity, resulting in frequent switching on and off. These infinite loops can not only affect the lifespan of the device but also increase energy consumption and even can lead to potential safety hazards. For instance, if a device experiences excessive heat due to its location or surroundings, it may switch off frequently or malfunction altogether, resulting in critical system failures that compromise overall performance and safety. Therefore, it is essential for system developers to consider these factors during the design and testing phases to ensure optimal performance even under varying environmental conditions. Additionally, users must be aware of these interferences. They can take necessary precautions to prevent potential hazards while using their devices.

Results on commercial data. There are 45 interferences (ground truth) in 120 commercial applications, and KnowDetector successfully detected all interferences. Among them, 13 are conflicts, 27 are loops, 2 are redundancy, and 3 are blocked actions. The related cases of blocked action all happened on the mobile phone, for example, the "Do Not Disturb" mode and the ringer cannot be turned on at the same time, and the volume cannot be adjusted during a call. Unlike the results of IFTTT, infinite loops occupy a larger proportion of interferences in commercial data.

B. RQ2. Accuracy

To evaluate the accuracy of KnowDetector in identifying application interferences, we used the test suite of IoTBench [19]. This dataset includes a set of customized SmartThings classic applications that specify some specific interferences instead of common interferences. We focus on this dataset due to the lack of a common interference benchmark. Thus, we extended KnowDetector to detect some specific interferences. These interferences are defined by violating pre-defined properties. For example, one property can be defined as "The light must be turned off when sleeping". A specific interference occurs if one application violates the pre-defined properties.

Challenge. During our evaluation of KnowDetector against state-of-the-art methods, we faced a challenge regarding their reproducibility. Unfortunately, most related works [9], [32], [11], [18], [33], [34], [35] are not available in open source form. This lack of accessibility and reproducibility hinders the comparison between KnowDetector and existing solutions. However, we were able to find Soteria [16], IoTSan [29], and IoTCom [17], which have published comprehensive detection results on IoTBench. Therefore, we can conduct a fair comparison between KnowDetector and these techniques. The interferences that occur, either individually or in groups, have been pre-identified, serving as the ground truth. However, SmartApps in IoTBench do not involve interferences related to

TABLE II
DISTRIBUTION OF THE DETECTED INTERFERENCES ACROSS PATTERNS ON IFTTT APPLETS

Pattern	P1.1 Device conflict	P1.2 Env. conflict	P2.1 Device loop	P2.2 Env. loop	P3 Redundancy	P4 Blocked action	Total
Number	104	20	38	12	123	15	312

TABLE III
DESCRIPTIONS OF NEWLY DESIGNED APPLICATIONS

Benchmark	Violation	Description
N1	P4	Turn on the flashlight after turning on the camera on your phone.
Bundle 7	P1.1	When temperature rises above 26°C, turn AC in cooling mode and turn off the light. When AC is turned on, turn on the light.
Bundle 8	P2.2	When temperature rises above 26°C, turn AC in cooling mode. When temperature drops below 24°C, turn AC in heating mode.

TABLE IV
VIOLATION DETECTION PERFORMANCE COMPARISON BETWEEN SOTERIA, IOTSAN, IOTCOM AND KNOWDETECTOR. TRUE POSITIVE (TP) AND FALSE NEGATIVE (FN) ARE DENOTED BY SYMBOLS ✓ AND ◯. (X#) REPRESENTS THE NUMBER # OF DETECTED INSTANCES.

Test Case	Soteria	IoTSan	IoTCom	KnowDetector
Individual Apps				
ID1.BrightenMyPath	✓	✓	✓	✓
ID2.SecuritySystem	✓	◯	✓	✓
ID3.TurnItOnOff	✓	◯	✓	✓
ID4.PowerAllowance	✓◯	(◯2)	(✓2)	(✓2)
ID6.TurnOnSwitchNotHome	✓	✓	✓	✓
ID7.ConflictTimeAndPresence	✓	◯	✓	✓
N1.TurnOnCameraAndFlash	◯	◯	◯	✓
Bundles of App				
Application Bundle 1	✓	✓	✓	✓
Application Bundle 2	✓	◯	✓	✓
Application Bundle 3	✓	◯	✓	✓
Application Bundle 4	◯	◯	✓	✓
Application Bundle 5	◯	◯	✓	✓
Application Bundle 6	◯	◯	✓	✓
Application Bundle 7	◯	◯	◯	✓
Application Bundle 8	◯	◯	◯	✓
Precision	88.9%	100%	100%	100%
Recall	53.3%	20%	80%	100%
F-measure	66.6%	33.3%	88.9%	100%

physical channels and implicit dependency between device services. Therefore, we include three SmartApp bundles (Bundles 4-6) designed by IoTCom, which involve interferences related to physical channels. Besides, we provide one individual app (N1) and two bundles (Bundles 7-8) with designed interferences of implicit dependencies. Table III gives the descriptions of these applications.

Detection extension. We extend KnowDetector to detect specific interferences. As mentioned above, the specific interference is detected by checking whether the applications violate the pre-defined properties. We refer to the property descriptions listed in Soteria [16] and state how we integrate them into KnowDetector for specific interference detection.

Soteria lists various property descriptions, and we convert these descriptions into respective graph patterns. These patterns are then matched based on the automation graph. For instance, one of the properties listed in Soteria states that "The lights must be turned off when the sleep sensor detects a user is sleeping." To identify such an interference, we negate the prohibited (must not) properties. Therefore, the above description can be translated into the following graph pattern: <sensor, sleep, detected>→ <light, switch, on>. Note that, the translation and analysis of property description and Samsung SmartApp are performed manually.

Results. Table IV presents the results of our experiments, which aimed to evaluate the accuracy of KnowDetector in detecting specific interferences. We compared it with other state-of-the-art techniques, Soteria, IoTSan, and IoTCom.

The individual applications consist of nine test cases (ID1-9). However, we have excluded three of these test cases: ID5.FakeAlarm, ID8.Location-SubscribeFailure and ID9.DisableVacationMode. These three test cases had code logic errors that prevented us from directly converting them into corresponding elements. For instance, ID5.FakeAlarm called a virtual alarm API that did not actually operate the alarm, making it cannot be translated into an action like <alarm, switch, on>. ID8.LocationSubscribeFailure failed to map the location through the subscribe() function, while ID9.DisableVocationMode did not enable vacation mode in the code logic. Finally, we conducted comparative experiments on the remaining 7 (6+N1) individual applications.

KnowDetector identifies all interferences in 7 individual applications and 8 bundles of Apps. However, IoTCom is unable to detect the interference caused by N1.TurnOnCameraAndFlash and bundle 7-8. This is because IoTCom did not capture the relationships between device capabilities. Additionally, bundles 4-6 define interferences that occur due to physical channels between applications, which Soteria and IoTSan failed to detect due to their lack of modeling of the physical channels. Overall, KnowDetector gets the highest overall score with 100% precision, recall, and F-measure in each category.

C. RQ3. Performance

This section evaluates the analysis time required by different phases of KnowDetector. It describes the overhead incurred by the automation graph construction and pattern matching drawn from IFTTT applets and commercial automation applications.

As illustrated in Figure 8, KnowDetector is highly practical and scalable when tested across a range of applications. The line with circles represents the time required for graph construction, and the line with rectangles represents the time required for interference detection. Note that the graph is with a logarithmic axis. More specifically, as the number of applications increased from 1 to 5,000, the time required for execution remained consistently low, demonstrating that the tool can effectively handle large datasets in real-world contexts. Notably, when analyzing 1000 applications, the tool exhibited a minimal overhead of only 931+32 milliseconds, indicating that it is well-suited to large-scale applications without significantly impacting system performance.

In practice. The ability of our tool to maintain its efficiency across different scales is a key indicator of its versatility and effectiveness. The results suggest that our tool can be easily used for diverse scenarios, making it a valuable asset for researchers and practitioners alike. The scalability and practicality of our tool have made it a valuable asset in real-world contexts, as evidenced by its successful integration into an anonymous company's smart home system. Our tool has been utilized within the system to detect interference with minimal overhead, providing valuable insights for improving the system's overall performance. The results obtained through our tool's analysis have helped enhance the user experience and improve the system's efficiency.

Furthermore, the scalability of KnowDetector indicates its potential for future extension and development. As technologies evolve and datasets continue to grow in size and complexity, the ability of our tool to adapt and scale accordingly will be invaluable for maintaining its relevance and usefulness.

Comparison with other work. Most methods based on model checking can only identify whether there is interference between two applications. However, detecting such application pairs often requires a significant amount of time overhead. For instance, Soteria [16] takes an average of 4±2.1 seconds to extract the state model and 16 seconds to detect interferences between 1-2 applications. On the other hand, IOTCom [17] and IoTSan [29] take an average analysis time per bundle of 11.9 minutes (ranging from 0.05 to 104.78 minutes) and 216.9 minutes (ranging from 0.33 to 580.91 minutes), respectively. It is evident that detecting interferences between applications using model checking methods can be computationally expensive and time-consuming.

In summary, the result highlights our tool's practicality, scalability, and potential within a variety of real-world contexts. Its low overhead, adaptability, and versatility make it valuable for researchers and practitioners in fields where efficient TAP application processing and analysis are critical.

VI. Discussion and Limitations

We examine the limitations of our work and discuss potential solutions for addressing those limitations.

Device assumption. Our approach assumes that all devices within the smart home system function properly and accurately reflect their environment. By leveraging our application interference detection technique in conjunction with device reliability research, we can proactively ensure the dependability of smart home systems. Early identification and resolution of potential issues allow for enhanced correctness and efficiency of these systems. Ultimately, our work aims to provide robust and reliable smart home solutions.

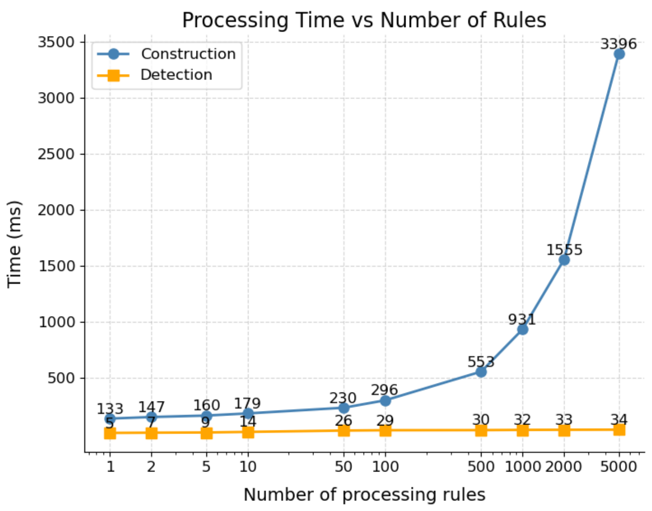


Fig. 8. Construction and detection time overhead

Interference resolution. Our approach is centered around identifying interferences within the measured target. However, it is important to note that users may not necessarily trigger these detected interferences. Instead, our system serves as an early warning mechanism, alerting users to potential conflicts that could arise. This proactive approach allows for preemptive measures to be taken to mitigate any potential issues and ensure the smooth functioning of the target system.

Knowledge model. Our current efforts in modeling primarily focus on capturing relevant information regarding device services. However, by incorporating additional data, such as device location, we can significantly enhance our effectiveness in modeling IoT environments. The introduction of location information allows for more precise detection and resolution of related interferences. For instance, consider a scenario where one rule influences the temperature in a specific area of a house, while another rule utilizes the temperature as a trigger for different actions. Although these two rules reference the same physical entity (i.e., temperature), they are irrelevant for their different locations.

VII. Related Work

The IoT ecosystem has garnered significant attention from various perspectives [36]. In this context, we primarily focus on discussing research efforts related to our work, i.e., violation and interference detection on automation applications.

For ensuring the security of IoT applications, most approaches apply the model checking techniques, such as Soteria [16], IoTSan [29], HomeGuard [9], IOTCom [17], and TAPInspector [37]. These approaches extract behavior models from IoT applications using AST analysis on Groovy codes and construct control flow graphs (CFGs) for model construction. IoTSan translates the Groovy code of applications to Java for using Bandera [38], a model checking toolset for Java programs. Unlike source code analysis, IoTMon and iRuler

[8] use NLP techniques to identify behavior models for model checking or risk analysis where source code is unavailable. Safechain [39] is a model checking based system that manually constructs FSMs of IoT systems and optimizes the FSM's state space by grouping functionally equivalent attributes and pruning redundant attributes. SIFT [40], AutoTAP [13], ContexIoT [41], and IoTGuard [11] mainly focus on action-related violation detection and resolution in either static or dynamic manners. The techniques based on model checking generally have some problems, such as excessive overhead and low scalability. For new devices, they also need to re-model and extend their method.

There is also a minor aspect of work related to knowledge-based interference detection. These techniques employ Natural Language Processing (NLP) and knowledge graphs to enhance semantics and aid in detecting interference. A3ID [35] uses the common sense knowledge graph ConceptNet [42] and WordNet, a vocabulary database, to enhance the comprehension of IoT devices. This method identifies opposing or conflicting device actions by grouping device behaviors. Bing Huang et al. [18] use knowledge to detect conflicts between smart home applications associated with environmental attributes. Such conflicts occur when various IoT events influence the environment in opposing ways, such as simultaneously turning on the air conditioner and opening windows. Their approach focused on the relationship between device services and the physical environment in an IoT setting. However, they did not consider the connection between the device's capabilities themselves and the impact of the environment. This could lead to inconsistencies in device functionality. For instance, their study suggests that the effect of air conditioning on temperature is only reduced, which is an unreasonable assumption.

VIII. Conclusion

This paper presents a novel fine-grained knowledge graph model and a lightweight tool KnowDetector to detect automation application interferences. Our model captures the intricate relationships between IoT services and their impact on the physical channels. The model enhances semantics while effectively detecting application interferences. Specifically, KnowDetector can detect ten patterns of interferences of four types: conflict, loop, redundancy, and blocked action. In our evaluations, we demonstrate the high accuracy and effectiveness of KnowDetector in detecting application interferences, as well as its low detection overhead, making it ideal for real production environments. Overall, our work contributes to the growing body of research on automated systems, and we believe that it has a significant potential impact on software engineering practice.

Acknowledgements

This work is supported by National Natural Science Foundation of China U20A6003, China Southern Power Grid Company Limited (Grant No.037800KK58190001).

References

[1] (2022) Ifttt. [Online]. Available: https://ifttt.com/
[2] (2022) Smartthings. [Online]. Available: https://www.smartthings.com/
[3] (2022) Home assistant. [Online]. Available: https://www.home-assistant.io/
[4] (2022) Openhab. [Online]. Available: https://www.openhab.org/
[5] (2022) Zapier. [Online]. Available: https://zapier.com/
[6] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proceedings of International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2015, p. 215–225.
[7] F. Corno, L. De Russis, and A. Monge Roffarello, "Empowering end users in debugging trigger-action rules," in *Proceedings of CHI Conference on Human Factors in Computing Systems (CHI)*, 2019, p. 1–13.
[8] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, p. 1439–1453.
[9] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 411–423.
[10] R. Liu, Z. Wang, L. Garcia, and M. Srivastava, "Remediot: Remedial actions for internet-of-things conflicts," in *Proceedings of International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys)*, 2019, p. 101–110.
[11] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.
[12] C. Rocha, J. Meseguer, and C. Muñoz, "Rewriting modulo smt and open system analysis," in *Proceedings of International Workshop on Rewriting Logic and its Applications (WRLA)*, 2014, pp. 247–262.
[13] L. Zhang, W. He, J. Martinez, N. Brackenbury, S. Lu, and B. Ur, "Autotap: Synthesizing and repairing trigger-action programs using ltl properties," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2019, pp. 281–291.
[14] Z. Fang, H. Fu, T. Gu, Z. Qian, T. Jaeger, and P. Mohapatra, "Foresee: A cross-layer vulnerability detection framework for the internet of things," in *Proceedings of International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2019, pp. 236–244.
[15] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, "Understanding and automatically detecting conflicting interactions between smart home iot applications," in *Proceedings of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, p. 1215–1227.
[16] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 147–158.
[17] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, p. 272–285.
[18] B. Huang, H. Dong, and A. Bouguettaya, "Conflict detection in iot-based smart homes," in *Proceedings of International Conference on Web Services (ICWS)*, 2021, pp. 303–313.
[19] (2018) Iotbench test suite. [Online]. Available: https://github.com/IoTBench/IoTBench-test-suite
[20] (2022) Knowdetecor. [Online]. Available: https://github.com/tcse-iscas/KnowDetector
[21] (2022) Smartapp. [Online]. Available: https://developer.smartthings.com/docs/connected-services/smartapp-basics/
[22] (2022) Mi home. [Online]. Available: https://www.mi.com/global/smart-home
[23] (2022) Huawei ai life. [Online]. Available: https://consumer.huawei.com/en/mobileservices/ai-life/
[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, 2020.

[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[26] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv:2205.01068*, 2022.

[27] (2022) Openai api. [Online]. Available: https://platform.openai.com/

[28] (2022) spacy: Industrial-strength natural language processing. [Online]. Available: https://spacy.io/

[29] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel, "Iotsan: Fortifying the safety of iot systems," in *Proceedings of International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018, p. 191–203.

[30] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *Proceedings of USENIX Conference on Security Symposium (SEC)*, 2018, p. 1687–1704.

[31] X. Mi, F. Qian, Y. Zhang, and X. Wang, "An empirical characterization of ifttt: Ecosystem, usage, and performance," in *Proceedings of Internet Measurement Conference (IMC)*, 2017, p. 398–404.

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

[33] X. Li, L. Zhang, and X. Shen, "Diac: An inter-app conflicts detector for open iot systems," *ACM Transactions Embedded Computing System*, vol. 19, no. 6, 2020.

[34] A. Kashaf, V. Sekar, and Y. Agarwal, "Protecting smart homes from unintended application actions," in *Proceedings of International Conference on Cyber-Physical Systems (ICCPS)*, 2022, pp. 270–281.

[35] D. Xiao, Q. Wang, M. Cai, Z. Zhu, and W. Zhao, "A3id: An automatic and interpretable implicit interference detection method for smart home via knowledge graph," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 2197–2211, 2020.

[36] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1362–1380.

[37] Y. Yu and J. Liu, "Tapinspector: Safety and liveness verification of concurrent trigger-action iot systems," *IEEE Transactions on Information Forensics and Security*, vol. 17, p. 3773–3788, 2022.

[38] J. Hatcliff and M. Dwyer, "Using the bandera tool set to model-check properties of concurrent java software," in *Proceedings of International Conference on Berlin Heidelberg*, pp. 39–58.

[39] K.-H. Hsu, Y.-H. Chiang, and H.-C. Hsiao, "Safechain: Securing trigger-action programming from attack chains," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 10, pp. 2607–2622, 2019.

[40] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, "Sift: Building an internet of safe things," in *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN)*, 2015, p. 298–309.

[41] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. Unviersity, "Contexlot: Towards providing contextual integrity to appified iot platforms." in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 2–2.

[42] (2022) Conceptnet. [Online]. Available: https://conceptnet.io/