Detecting and Handling IoT Interaction Threats in Multi-Platform Multi-Control-Channel Smart Homes

Haotian Chi
Shanxi University, Temple University
htchi@sxu.edu.cn

Qiang Zeng
George Mason University
zeng@gmu.edu

Xiaojiang Du
Stevens Institute of Technology
xdu16@stevens.edu

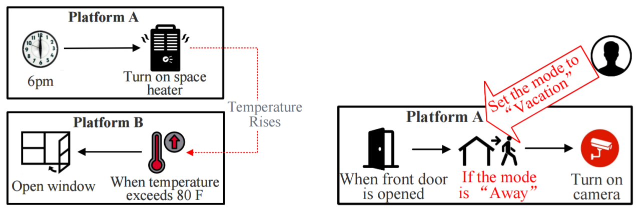
Abstract

A smart home involves a variety of entities, such as IoT devices, automation applications, humans, voice assistants, and companion apps. These entities interact in the same physical environment, which can yield undesirable and even hazardous results, called *IoT interaction threats*. Existing work on interaction threats is limited to considering automation apps, ignoring other IoT control channels, such as voice commands, companion apps, and physical operations. Second, it becomes increasingly common that a smart home utilizes multiple IoT platforms, each of which has a partial view of device states and may issue conflicting commands. Third, compared to detecting interaction threats, their handling is much less studied. Prior work uses generic handling policies, which are unlikely to fit all homes. We present IoTMediator, which provides accurate threat detection and threat-tailored handling in multi-platform multi-control-channel homes. Our evaluation in two real-world homes demonstrates that IoTMediator significantly outperforms prior state-of-the-art work.

1 Introduction

Rapid development of IoT has led to the flourishing deployment of smart homes. A smart home is a complex system involving a variety of entities, such as IoT devices, automation applications, humans, voice assistants, and companion apps. These entities interact in the same physical environment, which can cause undesirable and even hazardous interaction results, called *interaction threats*. For example, as shown in Figure 1(a), an app, which turns on a space heater at 6pm, can trigger the execution of another app, which opens the window when the temperature exceeds a threshold; this gives burglars a chance to break in. This problem draws great attention [21, 23, 26, 29, 30, 34, 50, 54]. However, as illustrated in Table 1, existing work has three major limitations.

First, prior work is limited to studying Cross-App Interaction (CAI) threats [26], which are caused by the interaction of automation apps (i.e., interaction threats via the automation channel), **ignoring other IoT control channels**. Actually, in addition to the automation channel, an IoT device can be controlled through various *manual control* channels, such as companion apps, voice commands, and physical operations. For example, as illustrated in Figure 1(b), a user sets the `mode` to "*Vacation*" using a companion app, which disables the automation that turns on security cameras when the front door is opened and the home is in "*Away*" mode. Unlike automation apps, manual controls are less predictable. We call the interaction between a manual control and automation apps *Cross Manual-control and Automation Interaction* (CMAI), which has not been studied yet.



(a) Cross-App Interaction (CAI) threat

(b) Cross Manual-control and Automation Interaction (CMAI) threat

Figure 1: Examples of interaction threats. (a) The interaction of the two apps may cause the window to be opened. (b) If the user sets "*Vacation*" mode using a companion app, the automation cannot turn on the camera when a break-in happens; such an interaction threat involves manual control and cannot be detected by prior work.

Table 1: Comparing prior work with IoTMediator (our work), regarding whether the three limitations are addressed. ✓: Yes; ✗: No..

	Detect Cross Manual-control and Automation Interaction?	Support Multi-Platform Homes?	Threat-Tailored Handling?
Soteria [21]	✗	✗	✗
IoTSan [50]	✗	✗	✗
SafeChain [34]	✗	✗	✗
IoTIE [23]	✗	✗[1]	✗
iRuler [54]	✗	✗	✗
HomeGuard [26]	✗	✗	✗
IoTCom [15]	✗	✗	✗
IoTSafe [30]	✗	✗	✗
IoTGuard [22]	✗	✗[1]	✗
IoTMediator	✓	✓	✓

[1] Some work [22, 23] recognizes this limitation but do not address it (Section 3.1).

Second, it becomes increasingly common that a smart home utilizes **multiple automation platforms** [1,4,24], while existing IoT interaction threat detection systems typically assume one single platform. At first glance, it seems a trivial deployment issue, since the user can deploy a threat detector at each platform to adapt to a multi-platform home. However, on one hand, it is not uncommon that some IoT devices are connected to one platform, while others to another. As a result, *each* platform has a partial view of the devices and hence it is difficult for the multiple detectors that *scatter* in these platforms to predict whether/when/how one device interacts with another. On the other hand, one device may be controlled through multiple platforms (e.g., both Google Home [9] and Amazon Alexa [8]), which may issue conflicting control over devices. These issues are an elephant in the room: they cause significant challenges in detecting interaction threats, while existing work does not discuss them.

Third, despite the much work on detecting interaction threats, their **handling** is much less studied. Static approaches [21, 23, 26, 29, 34, 50, 54] need users to discard, rewrite or reconfigure automation apps that may cause interaction threats, even though many reported threats are actually false positives. A few works [22,30] assume that generic predefined policies can fit different homes, which is impractical given the diversity of homes, scenarios and user preferences. A security expert can define custom policies for a home [46], but this approach does not scale well for many homes and incurs extra costs. Threat-tailored handling that factors in the context and consequences of a threat instance is much desired but not available.

We present IoTMediator, which addresses all the three limitations. It provides accurate detection of interaction threats in multi-platform multi-control-channel homes and generates threat-tailored handling. Inspired by [25], we leverage a hub-based architecture, where a local mediator mediates the original communication between IoT devices and their platforms. By unifying the device identifiers across multiple platforms, the mediator acquires a global view of device events and commands. On top of this, a two-phase detection method is devised: it first identifies interaction threat candidates via static analysis, and then detects real threats based on dynamic information. An automation app can be modeled as one or more automation rules, each in the form of $\langle$*trigger, condition, action*$\rangle$. By considering the impact of an automation rule on the trigger, condition and action of another rule or a manual control, we systematically categorize interaction threats, including the new CMAI threats. When a threat is detected, according to the threat type and other dynamic information, threat-tailored handling is generated.

We evaluate IoTMediator in two real-world smart homes. The first is a one-resident apartment installed with 21 automation apps and 22 IoT devices, which are connected to four platforms: SmartThings, Alexa, IFTTT and Philips Hue. The second is a two-resident two-floor house, where 22 devices are connected to three platforms (SmartThings, openHAB, Philips Hue) and 14 automation apps installed. The evaluation demonstrates that IoTMediator can effectively detect all the types of interaction threats and generate threat-tailored handling. We make the following contributions:

- **Detecting Cross Manual-control and Automation Interaction (CMAI) Threats.** We identify a new family of interaction threats due to the interactions between manual controls (via companion apps, voice commands or physical operations) and automation apps. While prior work on interaction threats is limited to considering the automation channel, our work extends the scope to various control channels and provides a comprehensive categorization.

- **Supporting Multi-Platform Homes.** A multi-platform home raises intriguing challenges in detecting and handling interaction threats. We leverage a hub-based architecture[1] to mediate IoT messages between IoT devices and servers and translate device IDs across platforms, which makes cross-platform interaction detection and handling viable.

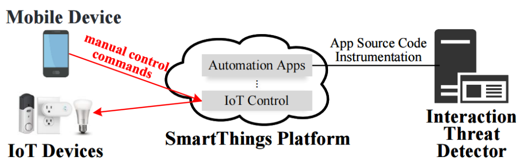
- **Threat-Tailored Handling.** Given diverse homes and user preferences, a one-size-fits-all solution using generic policies does not work well. IoTMediator is the first that generates threat-tailored handling, which comprises user-friendly options tailored to the threat instance.

2 Threat Model

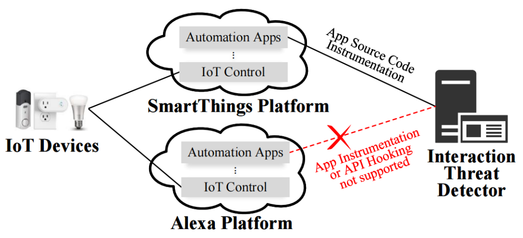
Interaction threats can be created or exploited by attackers in multiple ways. (1) Given those widely installed apps, an attacker can develop and promote apps that cause interaction threats with the popular apps. The apps can individually pass the malicious-app checking but cause threats together. (2) By sniffing encrypted WiFi traffic of a home [24,63], an attacker can infer information about devices and apps in a victim home and use it to inject and predict interaction threats. For example, a robot vacuum cleaner that starts working at 10am triggers another app, which sets the home to "*home*" mode when motion is detected [30]. An attacker can infer when the interaction arises and break in a home, without triggering an alarm since it is the "*home*" mode. (3) Consider the example in Figure 1(a): via the interaction threat, an attacker can manipulate a well secured device (e.g., the window) by compromising a vulnerable device (e.g., turning on the space heater). Prior work [22, 26, 29, 30, 34, 54] assumes a similar threat model.

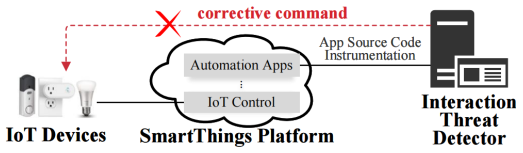
We clarify that not all interactions are hazardous, but when a new interaction arises, the user should be aware of it to avoid confusions and undesired interaction results. Our work reports interactions to users and provides handling options.

[1] We clarify that the architecture was first proposed by a work that protects user privacy [25]; we employ it for a very different purpose (Section 3.2).



(a) Cannot capture manual controls


Automation Apps
IoT Control
SmartThings Platform
App Source Code Instrumentation
IoT Devices
Automation Apps
IoT Control
Alexa Platform
App Instrumentation or API Hooking not supported
Interaction Threat Detector
(b) Cannot generalize to other platforms



(c) Cannot send corrective commands

Figure 2: Limitations of prior systems.

3 Design Overview

3.1 Challenges

As illustrated in Table 1, existing approaches cannot detect or handle interaction threats well in multi-platform multi-control-channel homes. We discuss the challenges in designing cross-platform interaction detection and handling.

Monitoring manual controls in real time is required for detecting CMAI threats. Manual device control via companion apps or voice commands generates cyberspace commands and results in new events. Manual physical operations do not generate cyberspace commands but cause new events as well after changing device states. Neither static approaches [21, 26, 34, 50, 54] nor the instrumentation-based dynamic approaches [22, 30] can capture manual controls. As shown in Figure 2(a), automation app instrumentation cannot monitor manual control behaviors since the commands do not go through the automation apps. Therefore, to detect CMAI threats, the detector must be able to monitor manual controls as well as automation apps.

Threat detection across multiple platforms needs a global view. Most existing works [21, 26, 30, 34, 50, 54] only consider a single-platform system (e.g., SmartThings or IFTTT) and cannot accurately detect threats in multi-platform homes. To detect (and handle) interaction threats among automation apps running on different platforms, a solution must have a global view (and control) over the different platforms. To achieve a global view, one can employ a generablizable accessing method for multiple platforms, or adopt an accessing method supported by each individual platform. However, due to the heterogeneity and closed-source nature of most platforms, neither can a generablizable technique be found from existing work (e.g., code instrumentation [22, 30] and API-based webhooking [30] are not applicable to most proprietary platforms), nor does it hold that each platform supports at least one technique for third-party accessing (see Figure 2(b)).

To *bypass* the limitation, a few works [22, 23] rewrite apps from different platforms and migrate them to a single platform, so that a single-platform approach can be applied. This suffers from the following issues. (1) It needs significant efforts to rewrite apps and mitigate apps from one platform to another. (2) Different platforms have different strengths (e.g., in app expressiveness, device connections, security and storage), and the user may want to utilize the multiple platform to benefit from the strengths. Therefore, we need to find a viable path to deal with multiple platforms.

Threat-tailored handling is needed. Handling approaches proposed by existing works are in two categories: (1) reconfiguring, re-writing or discarding automation apps, and (2) enforcing policies in runtime. Static approaches [21, 26, 34, 50, 54] usually propose the first category since they cannot intervene in the system runtime behaviors. The process is time-consuming and error-prone. Plus, due to false positives, users may be asked to modify apps that do not cause interaction threats. Dynamic approaches [22, 30] can employ the second category. However, it is unclear how to define generic policies that fit all the diverse homes, user needs, and scenarios. As an example, Celik et al. [22] use generic interaction handling policies and context-aware security policies. Some generic policies are used to prevent interaction threats, which is overly restrictive and disrupts desired automation. On the other hand, context-aware security policies, e.g., "*the door must be locked when a user is not present at home or sleeping*", define specific scenarios where devices must or mustn't be in certain states. Nevertheless, it is difficult, if not impossible, for experts to provide policies that fit all homes, or for end users to define a complete set of such policies to prevent all threats while ensuring good usability. Worse, existing techniques, such as code instrumentation [22, 30], have other limitations. As shown in Figure 2(c), the detector can *only* intervene while instrumented automation apps are executed, so cannot actively send post-execution corrective commands.

3.2 IoTMediator Overview

We design a system IoTMediator to detect and handle interaction threats in multi-platform systems, overcoming the aforementioned challenges. IoTMediator can run on a local device, such as a desktop, Raspberry Pi, WiFi router, etc. Figure 3 shows its architecture, which has three modules: Messenger, Threat Detection and Threat Handling.

The *Messenger* module is built to acquire a global view and control over the multiple platforms. It has two components *device gateway* and *device virtualization*, which connect with

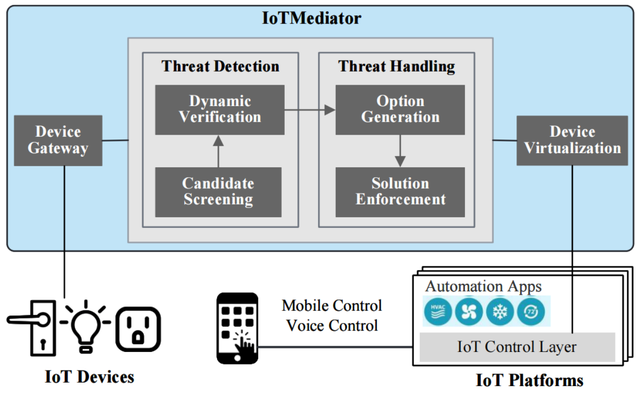


Figure 3: Architecture of IoTMediator.

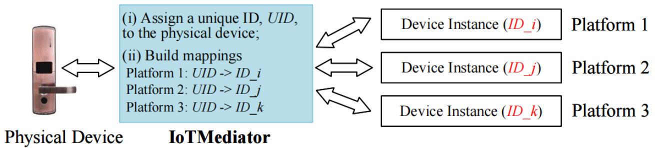


Figure 4: Device identifier unification across platforms.

the smart devices and platforms, respectively. As shown in Figure 3, Messenger segregates and mediates the original connections between IoT devices and platforms. Therefore, Messenger is capable of (1) intercepting all events from IoT devices and forwarding them to IoT platforms; (2) intercepting all cyberspace commands coming from automation apps, companion apps and voice commands and forwarding them to IoT devices; and (3) generating commands and sending them to IoT devices.

Comparison with PFirewall. The mediation architecture was first proposed in PFirewall [25]. Our work differs from PFirewall in the following aspects. (1) *Different purposes.* PFirewall, as well as Peekaboo [38], protects IoT user privacy, while our work studies interaction threats. (2) *One-way vs. two-way mediation.* PFirewall mediates outgoing data flows (i.e., IoT events) only, while IoTMediator mediates data flows in both directions (i.e., IoT events and commands). (3) *Cross-platform checking.* PFirewall does not conduct cross-platform checking, while IoTMediator does. For example, when two platforms issue lock-related commands to a home, IoTMediator needs to figure out whether the two commands refer to the same lock or not. To facilitate this capacity, as shown in Figure 4, our Messenger assigns a unique ID *UID* to each physical device, and maintains a mapping between the unique ID and an instance ID *per* platform that connects the device. The mappings are then used to translate between unique IDs and device instance IDs.

Modules *Threat Detection* and *Threat Handling* are built on top of Messenger. They are capable of viewing and controlling all the events/commands mediated by Messenger. In the Threat Detection module, a *candidate screening* component utilizes static information (e.g., automation apps, device types) to identify interaction threat candidates, which are monitored in the runtime by a *dynamic verification* component to detect real interaction threats. When a real interaction is detected, the *option generation* component in the Threat Handling module automatically generates handling options, according to the details of the detected threat instance. Once a handling option is selected as the solution by the user, the *solution enforcement* component enforces the solution to handling the current and future occurrences of the interaction threat. We present the details of interaction threat detection and handling in Section 4 and Section 5, respectively. We are open-sourcing the code of IoTMediator at https://github.com/HaotianChi/IoTMediator.

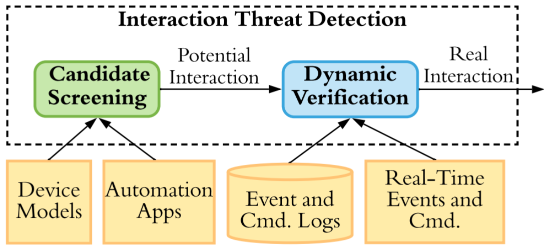


Figure 5: Cross-platform interaction detection framework.

4 Detecting Interaction Threats

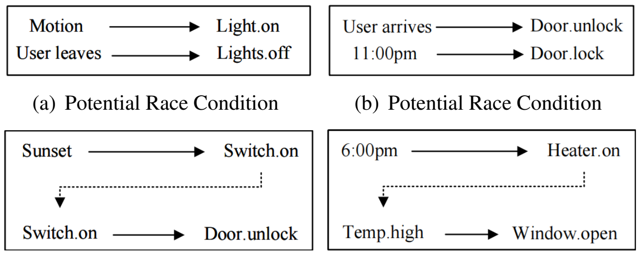
Figure 5 shows the workflow of interaction threat detection, which consists of two major phases: candidate screening and dynamic verification.

4.1 Candidate Screening

The candidate screening phase identifies all *potential* interactions between each pair of automation apps (i.e., CAI), or between each manual control and automation app (i.e., CMAI). Therefore, the candidate screening needs to take as inputs all the trigger-condition-action rules defined by automation apps and all the device controls (i.e., commands) supported by the devices. This paper employs the existing rule extraction techniques as a building block (see more details in Section 7). The *device gateway* can easily obtain the supported commands of each connected device from its device information during the join phase. With these inputs, this paper focuses on the interaction analysis based on automation rules and device control commands.

4.1.1 Identifying CAI Candidates

Consider two automation rules $R_i = \langle T_i, C_i, A_i \rangle$ $(i = 1,2)$, where T_i, C_i, A_i denote the trigger, condition and action, respectively. R_1 and R_2, if misconfigured, may cause interaction threats (e.g., conflicts, chained execution). Figure 6 shows examples of CAI. Note that the contribution of this paper is not to discover new CAI patterns, but to present a novel approach



(a) Potential Race Condition (b) Potential Race Condition

(c) Chained Execution (d) Chained Execution

Figure 6: Examples of CAI.

Table 2: Summary of different interaction threat patterns and the prerequisites for candidate screening. Notations: "," concatenates multiple constraints; $R_i = \langle T_i, C_i, A_i \rangle$ $(i = 1,2,3)$ denote three automation rules, where T_i, C_i, A_i denote the trigger, condition and action, respectively; c denotes a manual control; $T_1 = T_2$ denotes that two triggers are the same; $A_{(\cdot)} \Rightarrow C_{(\cdot)}$ and $A_{(\cdot)} \nRightarrow C_{(\cdot)}$ denotes that $A_{(\cdot)}$'s effect satisfies and dissatisfies $C_{(\cdot)}$, respectively; $c \Rightarrow C_3$ and $c \nRightarrow C_3$ denotes that c's effect satisfies and dissatisfies C_3, respectively; $A_{(\cdot)} \mapsto T_{(\cdot)}$ denotes that $A_{(\cdot)}$'s effect triggers $T_{(\cdot)}$; $c \mapsto T_3$ denotes that MC's effect triggers T_3; $A_1 = \neg A_2$ denotes that two actions have conflict; $c = \neg A_3$ denotes that a manual command and a rule action have conflict; $C_1 \wedge C_2$ denotes that both conditions could be satisfied.

Interaction Pattern	**Prerequisite**
CAI - Condition Enabling	$A_1 \Rightarrow C_2$
CAI - Condition Disabling	$A_1 \nRightarrow C_2$
CAI - Race Condition	$T_1 = T_2, A_1 = \neg A_2$
CAI - Potential **Race** Condition (RC)	$A_1 = \neg A_2$
CAI - Chained Execution	$A_1 \mapsto T_2$
CAI - Action Revert	$A_1 \mapsto T_2, A_2 = \neg A_1$
CAI - Condition Bypass	$T_1 = T_2, A_1 = A_2, C_1 \neq C_2$
CAI - Infinite Loop	$A_1 \mapsto T_2, A_2 \mapsto T_1$
CMAI - Chained Execution	$c \mapsto T_3$
CMAI - Potential **Race** Condition (RC)	$c = \neg A_3$
CMAI - Condition Enabling	$c \Rightarrow C_3$
CMAI - Condition Disabling	$c \nRightarrow C_3$

to detecting and handling CAI threats in multi-platform systems. For the ease of presentation, we collect the CAI threat patterns from state-of-the-art works [26, 50, 54], as shown in Table 2 and Figure 7.

We adopt the SMT-based approach in [26] for candidate screening. We list in Table 2 the prerequisite that a pair of rules, R_1 and R_2, must satisfy to be considered as a candidate of a certain CAI pattern. When the candidate screening component identifies a CAI candidate $\langle R_1, R_2, P \rangle$ where two automation rules $R_1 = (T_1, C_1, A_1)$ and $R_2 = (T_2, C_2, A_2)$ cause a potential threat pattern P, it reports the candidate to the dynamic verification component, which verifies if the candidate actually causes *real* interaction in runtime. Note that prerequisites of some CAI patterns, including race condition, potential race condition, condition bypass, infinite loop, are commutative, while the others are not and, in this case, the order of R_1 and R_2 in $\langle R_1, R_2, P \rangle$ matters.

4.1.2 Identifying CMAI Candidates

Manual control, like automation actions, can interact with the trigger, condition, or action of automation apps. We term this family of interactions as Cross Manual-Control and

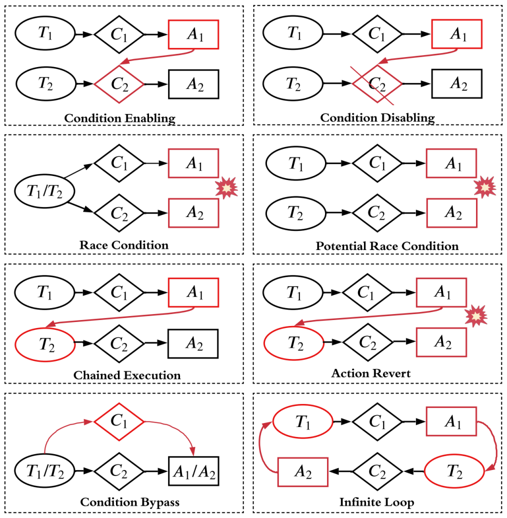
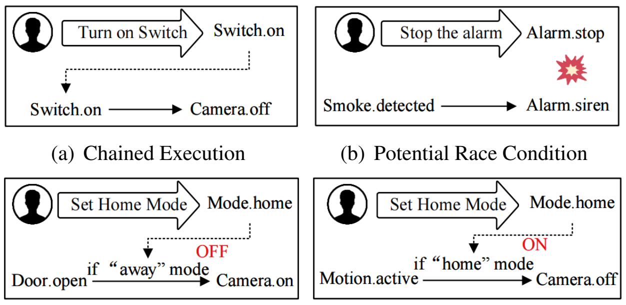


Figure 7: CAI patterns. In rules $R_m = \langle T_m, C_m, A_m \rangle, m \in \{1,2\}$, T_m, C_m and A_m are the trigger, condition and action, respectively.



(a) Chained Execution (b) Potential Race Condition

(c) Condition Disabling (d) Condition Enabling

Figure 8: Examples of CMAI.

Automation Interactions (CMAI) and categorize them into four patterns. Table 2 shows the prerequisite for candidate screening and Figure 8 illustrates some CMAI examples. To identify all potential CMAI, we collect a set of all supported controls by the devices (i.e., actuators) in a home, denotes as $\mathcal{C}$. By checking whether each control $c \in \mathcal{C}$ interacts with the trigger, condition and action of every automation app, and satisfies the prerequisite of one of the four CMAI patterns (see Table 2), all the CMAI candidates are identified. Suppose a manual control c and an automation rule R_3 satisfies a CMAI pattern P. We record this CMAI candidate as a tuple $\langle c, R_3, P \rangle$. All CMAI candidates are reported to the *dynamic verification* component. Note that manual controls are due to user actions and can only be captured at the runtime.

4.2 Dynamic Verification

Candidate screening, through static analysis, has the advantage of quickly identifying potential interaction cases but cannot precisely determine if a candidate actually occurs in a real environment. Figure 6(d) shows a chained execution

Table 3: Notations used in Section 4.2.2 (Detecting Real Interactions), Table 4, and Appendix B.

Symbol	Description
$(\cdot)$	A wildcard function argument that may be a rule's trigger, condition, action, or a manual control.
$\mathcal{E}(\cdot)$	A function that takes as input a rule's trigger or condition, and returns the corresponding event (e.g., `switch-on` event) that can activate the input trigger (e.g., *when switch is turned on*) or satisfy the input condition (e.g., *if the switch is currently on*).
$\mathcal{S}(\cdot)$	A function that takes as input a rule's trigger, condition or action, and returns the corresponding device state(s) (e.g., *switch is on*) which is subscribed to by the input trigger (e.g., *when switch is turned on*), satisfies the input condition (e.g., *if switch is currently on*), or set by the input action (e.g., *turn on switch*).
$\mathcal{C}(\cdot)$	A function that takes as input a rule's action or a manual control, and returns the command (e.g., `turn-on-switch` command) that is generated by the input rule action or manual control (e.g., *turn on switch*).
obs	An assertion that an event or command is observed.
$\mathtt{match}\mathcal{S}(\cdot)$	An assertion that the current device state (e.g., *the switch is on*) matches the anticipated state $\mathcal{S}(\cdot)$ (e.g., *switch is on*).
$\xrightarrow{\mathtt{match}\mathcal{S}(\cdot)}$	The assertion, $\mathtt{match}\mathcal{S}(\cdot)$, holds true during a specified time period.

candidate. Turning on the space heater at 6pm does not lead to opening the window *unless* the heater increases the temperature sensor's reading above the threshold (note the temperature sensor may have a distance from the heater). False alarms due to static analysis motivate us to further use dynamic verification to verify the candidates.

4.2.1 Recognizing Manual Control and Automation

To monitor manual controls and automation, the dynamic verification component listens to the incoming events and commands in real time and accesses the historical event and command logs maintained by IoTMediator. Physical operations (e.g., turning on an outlet by pressing the button on it) do not generate a cyberspace command but result in a new event (e.g., the outlet reports an *on* event). This pattern can be utilized to recognize physical operations. Manual controls (through mobile/web companion apps, voice commands) and automation apps always generate cyberspace commands. Upon the reception of a command, we need to determine its source. Note that the execution of an automation app issues certain command(s). We build a mapping for tracing from a command back to the automation app(s) that can generate this command. Then, we further check the precedent logs to see which specific automation app has been activated and issued the command. The source of the command is labeled as "`automation:AppName`" if an automation app is the source; otherwise, it is labeled as "`mobile/web/voice control`".

4.2.2 Detecting Real Interactions

Given an interaction candidate, $\langle R_1, R_2, P\rangle$ or $\langle c, R_3, P\rangle$, the dynamic verification component continuously monitors the events and commands to see if an instance of the interaction candidate occurs. For each CAI/CMAI type, we define a sequence of assertions that IoTMediator verifies in the runtime. See Table 3 for the notations. The verification is terminated if one assertion is not true. A CAI/CMAI candidate is verified to be a real threat instance if its corresponding assertions are verified to be true. We discuss the dynamic verification processes of several interaction patterns below and those of the other patterns in Appendix B.

CAI - Chained Execution/Action Revert. Suppose two rules R_1 and R_2 are a candidate of chained execution or action revert. IoTMediator checks the constraints below. If all the constraints are evaluated true, an instance of the interaction candidate is verified and vice versa. Action revert is a special case of chained execution, i.e., R_2, when triggered by R_1, performs a contradictory action against R_1. The contradiction of actions is confirmed in the candidate screening phase. Thus, IoTMediator takes the same steps to dynamically verify if R_1 and R_2 cause a chained execution or action revert.

$\mathtt{obs}\,\mathcal{E}(T_1), \mathtt{match}\mathcal{S}(C_1), \mathtt{obs}\,\mathcal{C}(A_1),$ /* R_1 is executed */
$\mathtt{match}\mathcal{S}(\neg T_2),$ /* The trigger of R_2 was false */
$\mathtt{obs}\,\mathcal{E}(T_2),$ /* The action of R_1 activates the trigger of $R_2 * /$
$\mathtt{match}\mathcal{S}(C_2)$ /* The condition of R_2 is true $* /$

In addition to action-trigger chaining via cyberspace interactions (Figure 6(c) shows an example, where the action *Switch.on* generates an event that triggers the execution of another automation rule), physical interactions also cause action-trigger chaining (Figure 6(d) shows an example). While detecting physical interactions is not our contribution, IoTMediator provides full-fledged capabilities for the purpose, including real-time event/command monitoring, historic events/commands logging, and controlling devices. Specifically, we employ static physical interaction relations [29] in the candidate screening phase and verify real ones in the dynamic verification phase. IoTMediator can be extended to incorporate techniques in IoTSafe [30] and IoTSeer [51] to handle special cases due to sophisticated physical effects, such as continuous effect, joint effect, etc.

CAI - Potential Race Condition. The execution order of two rules R_1 and R_2, which are vulnerable to potential race condition, is non-deterministic since they have different triggers. Therefore, we need to detect both cases: (1) R_1 is executed before R_2, and (2) R_2 before R_1. We present how to verify the former case below and the latter is similar.

$\mathtt{obs}\,\mathcal{E}(T_1), \mathtt{match}\mathcal{S}(C_1), \mathtt{obs}\,\mathcal{C}(A_1),$ /* R_1 is executed */
$\mathtt{obs}\,\mathcal{E}(A_1) \xrightarrow{\mathtt{match}\mathcal{S}(A_1)}$ /* The device state remains unchanged */
$\mathtt{obs}\,\mathcal{E}(T_2), \mathtt{match}\mathcal{S}(C_2)$ /* until R_2 executes with conflict actions */

CMAI - Chained Execution. To verify that a manual control c triggers the chained execution of an automation rule R_3, the following constraints are evaluated.

obs $\mathcal{C}(c)$, /* A manual control c is observed */
match$\mathcal{S}(\neg T_3)$, /* The trigger of R_3 was false */
obs $\mathcal{E}(T_3)$, /* The manual control c activates the trigger of R_3 */
match$\mathcal{S}(C_3)$ /* The condition of R_3 is true */

If a candidate is verified to yield a real interaction instance, it is handled by the threat handling module (Section 5).

5 Handling Interaction Threats

Different from existing app-instrumentation approaches that block or approve commands, our handling not only provides options tailored to a threat context and their explanations, but also generates corrective commands as needed.

5.1 Syntax of Handling Option and Solution

To handle an identified interaction threat, IoTMediator generates multiple *handling options* $\{Opt_j | j = 1, 2, \cdots\}$ for users to select from (see Section 5.2). The selected handling option is termed as the *solution* (i.e., $Sln \in \{Opt_j | j = 1, 2, \cdots\}$). A handling option Opt_j for an interaction threat consists of one or more *option rules*, each of which is is denoted as $OR = \langle \mathcal{V}, I, O, IT, V, A \rangle$. $\mathcal{V}$ denotes the set of all device or environment state values (e.g., *motion sensor is active*, *time is 8am*, and *the switch is on*) in a smart home. I denotes all types of events and commands (e.g., event: *motion-active*, event: *time-8am*, or command *turn-on-switch*). Note that the state values, event types and command types are device-sensitive. For example, the *motion-active* events from two motion sensors (labeled with different device IDs) are regarded as two different events. O is a set of meta operators $\{\Rightarrow, \overset{t}{\Rightarrow}, \rightarrow, \nrightarrow, \overset{t}{\rightarrow}\}$ that denote *enforce*, *enforce after t*, *pass the transmission*, *discard the transmission*, *discard the transmission within t and begin to pass after t*, respectively. V denotes a set of state values $V \subseteq \mathcal{V}$, all of which must be true for OR to take actions A. The option rule action A is a set of operations $A = \{a | a \in (\{\Rightarrow, \overset{t}{\Rightarrow}\} \times \mathcal{V}) \cup (\{\rightarrow, \nrightarrow, \overset{t}{\rightarrow}\} \times I) \cup \emptyset\}$, which include enforcing (without or with a delay t) device states ($\{\Rightarrow, \overset{t}{\Rightarrow}\} \times \mathcal{V}$), passing or blocking events/commands ($\{\rightarrow, \nrightarrow, \overset{t}{\rightarrow}\} \times I$) or doing nothing ($\emptyset$).

We use a shorthand $OR = \langle IT, V, A \rangle$ to denote an option rule since $\mathcal{V}$, I and O are shared by all option rules in the same home deployment. Thus, an option rule can be interpreted as "*when an instance of interaction threat IT is detected, if all state values specified by V are true,* IoTMediator *will take actions in A.*" For simplicity of presentation, the sets V and A are referred to using a single element v_1 and a_1, respectively, if they only have one element (i.e., $V = \{v_1\}$ and $A = \{a_1\}$).

5.2 Handling Option Generation

Comparison with Prior Approaches. Given the diversity of smart homes and users, an interaction could be a user-favored feature [34] or a security/safety threat, which depends on three factors: (a) the interaction pattern; (b) the involved automation apps and/or manual control; and (c) user intentions/preferences. For each interaction, IoTMediator extracts its corresponding factors (a) and (b) when detecting it, but cannot obtain factor (c), as it is difficult to figure out a user's intentions/preferences. Some existing works [26,50,54] ask users to rewrite/reconfigure/remove apps or specify security policies to handle interaction threats, which allow users to express their intentions/preferences (factor (c)); however, it require non-trivial expertise in IoT and is error-prone [27]. Other works [22] define generic policies to handle all interactions of the same patterns in the same manner (factor (a)), reducing user efforts but ignoring the actually involved apps and/or manual control (factor (b)), and of course ignoring user intentions (factor (c)); as a result, they are often too restrictive, which may violate user intentions and cause incorrect interventions. In this paper, IoTMediator adopts a threat-tailored strategy to handle interactions.

Generating Options. Given an identified interaction, IoTMediator considers handling choices that a user may make, including allowing, prohibiting, and/or remedying the interaction. IoTMediator then generates handling options representing these choices. The "Handling Options" in Table 4 show how IoTMediator generates handling options for an identified interaction based on its interaction pattern (factor (a)) and the involved automation app(s) and/or manual control (factor (b)). ("Explanation Templates" are described below and we present concrete examples in Section 5.3.)

User Decisions. IoTMediator presents useful information to help users make informed decisions (factor (c)), including text descriptions of the identified interaction, the involved automation apps and/or manual control, and the recommended handling options. A prior work [26] presented how to generate text descriptions of interaction threats and involved automation rules: populating pre-defined text templates with concrete information of interaction threats and automation rules. We extend the approach to additionally generate textual explanations for handling options by populating text templates, as shown in the "Explanation Templates" of Table 4. By reading the text, a user can pick a preferred handling option. IoTMediator also allows users to save the solution to handle future occurrences of that interaction.

5.3 Examples

We use examples of one interaction pattern, *potential race condition*, to show the followings: the rationale of the generated handling options, the advantage of our threat-specific handling compared to existing works, and the required user

Table 4: Handling options and explanation templates for some of the interaction threat patterns (see Table 11 for the other patterns). IoTMediator generates multiple handling options and an explanation for each option, which are provided to users for making informed decisions: understand whether an interaction is a threat or feature, and allow/prohibit/remedy the interaction by choosing a solution from the handling options.

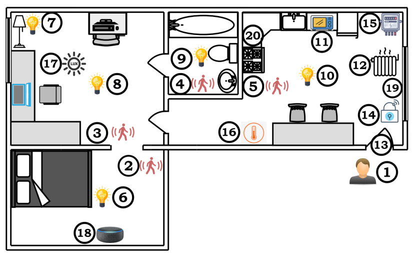
Interaction Pattern (P)	**Handling Options & Explanation Templates**
CAI - Potential RC	**Option 1**: $\langle\langle R_1,R_2,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_2)\rangle + \langle\langle R_2,R_1,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_1)\rangle$ **Explanation Template:** The execution order does not matter, so let the two rules execute without intervention. **Option 2:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_2)\rangle + \langle\langle R_2,R_1,P\rangle,\emptyset,[\rightarrow \mathcal{C}(A_1),\overset{\texttt{t}}{\Rightarrow} \mathcal{C}(A_2)]\rangle$ **Explanation Template:** The execution order matters and the final device state should be decided by R_2; thus, if action A_2 is executed first, execute it again `t` seconds after A_1 is executed (`t` is 30 by default but configurable). **Option 3:** $\langle\langle R_2,R_1,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_1)\rangle + \langle\langle R_1,R_2,P\rangle,\emptyset,[\rightarrow \mathcal{C}(A_2),\overset{\texttt{t}}{\Rightarrow} \mathcal{C}(A_1)]\rangle$ **Explanation Template:** The execution order matters and the final device state should be decided by R_1; thus, if action A_1 is executed first, execute it again `t` seconds after A_2 is executed (`t` is 30 by default but configurable). **Option 4:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\xrightarrow{\texttt{t}} \mathcal{C}(A_2)\rangle + \langle\langle R_2,R_1,P\rangle,\emptyset,\xrightarrow{\texttt{t}} \mathcal{C}(A_1)\rangle$ **Explanation Template:** A_1 and A_2 should not be issued too closely; instead, the time interval between the two actions should be at least `t` seconds (`t` is 30 by default but configurable).
CAI - Chained Execution	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\nrightarrow \mathcal{C}(A_2)\rangle$ **Explanation Template:** Action A_2 should not be executed. **Option 2:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_2)\rangle$ **Explanation Template:** Action A_2 should be executed. **Option 3:** $\langle\langle R_1,R_2,P\rangle,\texttt{cond}\subseteq \mathcal{V},\rightarrow \mathcal{C}(A_2)\rangle$ **Explanation Template:** Allow A_2 to be executed under a certain condition `cond` (`cond` is configurable and can be specific device states and/or time period).
CMAI - Chained Execution	**Option 1:** $\langle\langle c,R_3,P\rangle,\emptyset,\nrightarrow \mathcal{C}(A_3)\rangle$ **Explanation Template:** Action A_3 should not be executed. **Option 2:** $\langle\langle c,R_3,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_3)\rangle$ **Explanation Template:** Action A_3 should be executed. **Option 3:** $\langle\langle c,R_3,P\rangle,\texttt{cond}\subseteq \mathcal{V},\rightarrow \mathcal{C}(A_3)\rangle$ **Explanation Template:** Allow A_3 to be executed under a certain condition `cond` (`cond` is configurable and can be specific device states and/or time period).
CMAI - Potential RC	**Option 1:** $\langle\langle c,R_3,P\rangle,\emptyset,\nrightarrow \mathcal{C}(A_3)\rangle$ **Explanation Template:** Manual control c should always execute to override rule action A_3, but the rule action should not be executed to override the manual control. **Option 2:** $\langle\langle c,R_3,P\rangle,\emptyset,\rightarrow \mathcal{C}(A_3)\rangle$ **Explanation Template:** Manual control c and rule action A_3 can override each other. **Option 3:** $\langle\langle c,R_3,P\rangle,\emptyset,\xrightarrow{\texttt{t}} \mathcal{C}(A_3)\rangle$ **Explanation Template:** Manual control c should always execute to override rule action A_3, but the rule action should not be executed to override the manual control within `t` (`t` is 30 by default but configurable).
CMAI - Condition Disabling	**Option 1:** $\langle\langle c,R_3,P\rangle,\emptyset,\Rightarrow \mathcal{S}(A_3)\rangle$ **Explanation Template:** Action A_3 should be executed. **Option 2:** $\langle\langle c,R_3,P\rangle,\emptyset,\emptyset\rangle$ **Explanation Template:** Action A_3 should not be executed.

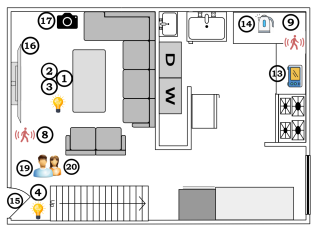
effort for making decisions. (See Section 6.3 for more examples and comparisons between IoTMediator and prior work with regard to handling other interaction threats. Note that IoTMediator has the limitation of increasing the user effort, which is discussed in Section 7.)

Example 1 of Potential Race Condition. It is non-deterministic whether a *potential race condition* case is desirable or not. Consider the two rules in Figure 6(a): the first rule for convenience *turns on the ceiling lamp when motion is detected in living room* and the second for energy saving *turns off all lights when user leaves home*. The two rules fit the pattern of potential race condition, i.e., the second rule overrides the command of the first rule when the user walks through the living room and then leaves home; however, the interaction actually does not cause a problem. Prior work [22] proposes a generic policy "*two events cannot respectively trigger two apps to perform conflict actions*" for handling potential race condition (Section 3.1). As a result, when the two rules are triggered in a row, the second rule will be disabled by the generic policy and thus lights are not turned off as expected. Even for the very simple interaction case, the generic policy fails to handle it properly. In contrast, with IoTMediator, a user who finds execution order and timing do not need to be intervened in can choose *Option 1* as the solution.

Example 2 of Potential Race Condition. Figure 6(b) illustrates an example where a reported potential race condition case is problematic. The first rule for convenience *unlocks door when the user approaches home* and the second for safety *locks the door at 11pm*. The two rules may or may not cause a real threat depending on the actual situations: (i) if the user arrives before 11pm, everything works fine, but (ii) if the user arrives after 11pm, the second rule runs before the first one; consequently, the door may be left unlocked overnight. Static approaches such as those in [26, 50, 54] handle this threat by presenting identified interaction threats as well as both rules to users and asking them to rewrite/reconfigure/remove automation rules. However, it harms the usability and functionality by simply modifying rules since both rules have their own functionalities. Some works [22, 30, 58] propose that users could specify security policies based on a set of expert-defined security policies; however, none of the policies listed in [22, 30, 58] can properly resolve the above Potential Race Condition. For example, the generic policy "*two events cannot respectively trigger two apps to perform conflict actions*" prevents the second rule from securing the home in situation (i) and disables the first rule to provide convenience in situation (ii), showing poor performance due to the lack of flexibility in considering user intentions.

As shown in Table 4, IoTMediator provides a comprehensive set of options for users to handle a potential race


(a) Testbed T_1


(b) 1st floor of Testbed T_2

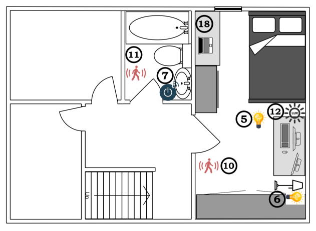

(c) 2nd floor of Testbed T_2

Figure 9: Floor plans and device placement. Devices denoted by the ID numbers are listed in Table 5. For the sake of brevity, IoT hubs/bridges are not illustrated.

condition. A user can choose to allow or prohibit the interaction, or fix it by taking into consideration the execution order and/or timing of the involved automation rules. In Example 2, IoTMEDIATOR presents the text descriptions of the rules and the interaction, and the explanation of options to the user. When perceiving that the final door state should be decided by the second rule, the user selects *Option 2* as the solution, which is interpreted as: *The execution order matters and the final device state should be decided by the second rule; thus, if the action "locks the door" is executed first, execute it again 30 seconds after "unlocks the door" is executed.* This way, the interaction threat can be handled properly.

6 Evaluation

In Section 6.1, we describe the deployment details of two real-world smart home testbeds used for evaluating IoTMEDIATOR. We present the performance of IoTMEDIATOR in terms of interaction threat detection and handling in Section 6.2 and Section 6.3, respectively. Latency introduced by IoTMEDIATOR is discussed in Section 6.4.

6.1 Smart Home Testbeds

There are no publicly available datasets of entire-home configurations that include devices, platforms, automation rules and logs (events, app commands and manual control). Similar to previous IoT security research [22,25,26,30,63], we build our own smart home testbeds, denoted as T_1 and T_2, to evaluate IoTMEDIATOR. T_1 is an apartment with one resident and T_2 is a two-floor house with two residents. The floor plans and device placements are shown in Figure 9. The details of devices

Table 5: IoT devices and their connections to platforms. **d-ID**: device ID. Acronyms: SmartThings (ST), Philips Hue (PH).

Testbed T_1		
d-ID	**Device Type**	**Connected Platforms**
–	SmartThings hub	SmartThings
–	Philips Hue bridge	Philips Hue
①	Presence sensor (smartphone)	SmartThings
②③⑤⑰	PH motion sensor	SmartThings, Alexa, IFTTT
④	PH motion sensor	Philips Hue
⑥⑦⑨⑩	Philips Hue bulb	PH, SmartThings, Alexa, IFTTT
⑧	LIFX bulb	SmartThings, Alexa, IFTTT
⑪⑫	Wemo smart plug	SmartThings, Alexa, IFTTT
⑬	ST multipurpose sensor	SmartThings, Alexa, IFTTT
⑭	Kwikset door lock	SmartThings, Alexa
⑮	ST power outlet	SmartThings
⑯	SmartThings motion sensor	SmartThings, Alexa, IFTTT
⑱	Alexa Echo Dot	Alexa
⑲⑳	ST WiFi plug	SmartThings, Alexa, IFTTT
Testbed T_2		
d-ID	**Device Type**	**Connected Platforms**
–	SmartThings hub	SmartThings
–	Philips Hue bridge	Philips Hue
①-⑥	Philips Hue bulb	PH, SmartThings, openHAB
⑦	ThirdReality switch	SmartThings
⑧-⑫	PH motion sensor	SmartThings, openHAB
⑬⑭	ST power outlet	SmartThings
⑮	ST multipurpose sensor	SmartThings, openHAB
⑯	ST WiFi plug	SmartThings
⑰⑱	Arlo Essential camera	SmartThings
⑲⑳	Presence sensor (smartphone)	SmartThings

and automation apps are listed in and Table 5 and Table 6, respectively. In total, 44 devices and 35 automation apps are installed on five different platforms (Alexa, IFTTT, SmartThings, openHAB, and Philips Hue). The apps are chosen from official app stores [7] and open-source datasets [2], or developed by the authors based on the examples from related literature [21,22,26,29]. In each testbed, IoTMEDIATOR runs on a Raspberry Pi 4 Model B. All the IoT devices, hubs and Raspberry Pis are provisioned by the researchers, except that the home wireless routers are offered by the testbed residents. We obtained an IRB approval for the research.

6.2 Interaction Threat Detection

To evaluate the performance of interaction detection, we first run the candidate screening component in both testbeds and we find 12 app groups (i.e., candidates) that have *potential* interaction threats for further testing. We compare the performance of IoTMEDIATOR with existing approaches through both microbench and one-week experiments.

6.2.1 Microbench

In this setting, we manually operate devices and trigger apps in each test group to check if our IoTMEDIATOR and prior systems [22,26,30,54][2] can detect the interactions accurately. We manually enumerate all possible combinations of initial device states in each group and then operate the devices to trigger the apps. The total number of combinations of initial

[2]We have the code of HomeGuard [26] and implemented the approaches in [22,30,54] for conducting the evaluation.

Table 6: Automation apps in the two testbeds. **RID**: app ID.

Testbed	RID	App Description and Device Binding	Platform
T_1	1	When motion ③ is detected in living room, if luminance ⑰ is below 15 lux, turn on floor lamp ⑦ and ceiling lamp ⑧.	SmartThings
	2	When front door ⑬ is opened, turn on ceiling lamp ⑧.	SmartThings
	3	When motion ⑤ is detected in kitchen, turn on the outlets for microwave ⑪ and heater ⑫.	IFTTT
	4	When motion ⑤ is detected in kitchen, if temperature ⑯ is above 72 °F, turn off the heater outlet ⑫.	SmartThings
	5	When the user ① arrives home, unlock the front door ⑭.	SmartThings
	6	When 11pm, turn off the kitchen lights ⑩.	Philips Hue
	7	When 11pm, lock the front door ⑭ and turn off outlets ⑪⑫.	SmartThings
	8	When the power ⑮ is higher than 1800W, turn off the heater outlet ⑫.	SmartThings
	9	When motion ② is detected in bedroom, if the time is between 9am and 11pm, turn on the light ⑥.	SmartThings
	10	When motion ② is detected in bedroom, turn on the light ⑥.	Alexa
	11	When 6pm, if the home is not in *saving* mode, turn on the heater ⑫.	Alexa
	12	When temperature ⑯ exceeds 75 °F, if the home is in *saving* mode, turn on the window opener switch ⑲.	IFTTT
	13	When oven outlet ⑳ turns on, set the location mode to *party* mode.	SmartThings
	14	When the location mode changes to *party* mode, unlock the door ⑭, and turn on lights ⑦⑧⑨⑩ and microwave outlet ⑪.	SmartThings
T_2	1	When motion ⑩ is detected, if luminance ⑫ is below 15 lux, turn on ceiling lamp ⑤ and floor lamp ⑥.	SmartThings
	2	When motion ⑩ is detected in bedroom, turn on ceiling lamp ⑤.	SmartThings
	3	When illuminance ⑧ falls below 10 lux in living room, if any user ⑱⑲ is at home, turn on ceiling lights ①②③.	SmartThings
	4	When illuminance ⑧ exceeds 30 lux in living room, if motion ⑧ is inactive, turn off ceiling lights ①②③.	openHAB
	5	When the door contact ⑮ is open, if the home is in *away* mode, turn on camera ⑰.	SmartThings
	6	When motion ⑩ is detected in bedroom, if the home is in *home* mode, turn off camera ⑱.	openHAB

Table 7: Microbench experiment for the comparison of interaction detection between the state-of-the-art approaches (with global view) and ours. **Test Group**: a candidate of a certain interaction pattern; it consists of a pair of automation rules, or a manual control and an automation rule. N_{all}: the number of all combinations of initial device states in an test group. Note that static approaches report a test group as problematic as long as one of the combinations causes a real interaction threat. N_{gt}: the number of initial device state combinations that cause a real interaction, based on our observations on the devices as *ground truth*. **N/A** denotes that a work does not consider and therefore cannot detect a specific interaction pattern. "**—**" means that the value cannot be computed due to "divided by zero". For instance, HomeGuard [26] never identifies the test group 11 & 12 as *CAI – Chained Execution* since the conditions of Rules 11 & 12 have no overlap; thus, the calculation of precision (i.e., the ratio of correctly reported cases to all reported cases) encounters "divided by zero".

Testbed	Test Group	Interaction Pattern	N_{all}	N_{gt}	Precision, Recall				
					HomeGuard [26]	iRuler [54]	IoTGuard [22]	IoTSafe [30]	Ours
T_1	5 & 7	CAI – Potential RC	24	6	**0.25, 1.00**	**N/A**	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	11 & 12	CAI – Chained Execution	128	2	**—, 0.00**	**N/A**	**N/A**	**0.50, 0.50**	**1.00, 1.00**
	13 & 14	CAI – Chained Execution	256	63	**0.25, 1.00**	**0.25, 1.00**	**1.00, 1.00**	**0.25, 1.00**	**1.00, 1.00**
	3 & 4	CAI – Race Condition	16	4	**0.25, 1.00**	**0.25, 1.00**	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	3 & 8	CAI – Action Revert	16	2	**0.13, 1.00**	**0.13, 1.00**	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	16	1	**0.06, 1.00**	**0.06, 1.00**	**N/A**	**N/A**	**1.00, 1.00**
	9 & 10	CAI – Condition Bypass	8	1	**N/A**	**0.13, 1.00**	**N/A**	**N/A**	**1.00, 1.00**
T_2	set *home* mode & 5	CMAI – Condition Disabling	8	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	set *home* mode & 6	CMAI – Condition Enabling	8	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	16	0	**0.00, —**	**0.00, —**	**N/A**	**N/A**	**—, —**
	3 & 4	CAI – Infinite Loop	192	6	**0.03, 1.00**	**0.03, 1.00**	**1.00, 1.00**	**N/A**	**1.00, 1.00**

device states is denoted as N_{all}. We observe that several combinations of the test groups indeed cause *real* interactions, while others do not. We record the observation results as *ground truths*. The number of combinations per test group that cause real interactions is denoted as N_{gt}. After that, we repeat the above enumeration process; instead of manual observation, we run IoTMediator and prior systems alongside to detect interactions under each enumeration in every test group. For each test group, the testing results of each work on all combinations is compared against the ground truths to calculate two detection performance metrics: ***precision*** (i.e., the ratio of correctly reported cases to all reported cases) and ***recall*** (i.e., the ratio of correctly reported cases to all problematic cases).

Note that only IoTMediator has a global view on multiple platforms. To exclude the impact of the global view issue and only compare the interaction detection capacity, we let prior systems in comparison have a global view on multiple platforms (by manually migrating all apps to a single platform). The results are summarized in Table 7. By comparing N_{all} and N_{gt}, we know that a candidate only causes a real interaction when the involved devices are in certain state combinations, a small portion of all possible ones. Thus, static approaches [26, 54] usually have a low detection precision (i.e., ***high false detection rate***) since they do not take the real-time device state into considerations and simply report potential interactions (i.e., candidates). On the contrary, dynamic approaches [22, 30] have a high detection precision (mostly equal to 1.00) because they only report real interaction cases that have been observed in runtime. However, for each of the prior systems [22,26,30,54], there are multiple test groups (marked with **N/A** in Table 7) that cannot be detected, since they do not consider some of the interaction patterns in their designs.

Moreover, none of the prior systems can detect any CMAI since they cannot capture manual controls. To sum up, the three prior systems [22, 26, 54] have low precision for all the test groups. IoTGuard has precision of 1 for only 5 out of 12 test groups. The four prior systems [22, 26, 30, 54] have a Recall of 1 for no more than 6 out of the 12 test groups. On the other hand, IoTMediator can detect all interaction patterns since it can monitor control behaviors from multiple control channels. This shows that IoTMediator has an advantage of detecting all interaction patterns, since it can monitor control behaviors from multiple control channels and it considers all the interaction patterns in design.

A test group 11 & 12 in testbed T_1 is interesting and we discuss it below. Although the static approach HomeGuard [26] can detect chained execution in general, it fails to identify the interaction between rules 11 and 12. Rules 11 and 12 have exclusive conditions “if the home is not in (energy) *saving* mode” and “if the home is in *saving* mode”, respectively, so they are considered impossible to run in a row. However, when the heater (turned on by rule 11) is heating the room (observed by rule 12), the home mode may be changed from other modes to *saving* mode, which makes the condition of rule 12 true. Therefore, HomeGuard misses the detection of chained execution between rule 11 and 12. On the other hand, IoTSafe [30] achieves 0.50 precision and 0.50 recall, because IoTSafe applies one-time testing result (i.e., the heater ⑫ can/cannot heat the room and increase the temperature ⑯ to above 75°F) to *predict* whether rule 11 always triggers rule 12 in the future run. However, the heating process may be affected by other factors such as heater interruption or season difference, making the runtime has an opposite result to the one-time testing one. IoTMediator detects the interactions accurately since it fully utilizes the real-time information.

6.2.2 One-Week Testing

While the microbench experiment highlights the coverage of various initial device state combinations, this experiment examines real-world scenarios. Specifically, we run both testbeds T_1 and T_2 in a realistic setting: the residents are asked to behave normally. We collect one week of the device event and command logs in both testbeds for evaluating the detection performance of our IoTMediator and two dynamic approaches IoTGuard [22] and IoTSafe [30]. Similar to the microbench experiment, we give IoTGuard and IoTSafe a global view to detect interaction threats. Table 8 shows the results. IoTGuard and IoTSafe achieve the same performance in most test groups except 11 & 12 and 13 & 14. In the dynamic testing, the heater ⑫ heats the room and makes the temperature sensor ⑯ measurement exceeds the threshold. With this knowledge for detection, IoTSafe can always detect the real interaction in 11 & 12, if any (i.e., recall is 1.00). Rule 11 only triggers rule 12 once in its 7 executions. IoTSafe has a 1.00 precision (compared to 0.25 in the microbench experiment) in the group 13 & 14 because rules 13 and 14 are always triggered in a chain in the one-week running. IoTSafe reports many false alarms (i.e., precision is 0.14) since it uses testing result rather than dynamic runtime information to detect interactions. In contrast, IoTMediator detects interactions in all test groups effectively and accurately, consistent with the results in the microbench experiment.

6.3 Interaction Threat Handling

We compare IoTMediator with two related work IoTGuard [22] and IoTSafe [30] regarding handling of interaction threats. IoTGuard [22] enforces both generic interaction handling policies (G) and context-aware security policies (C). IoTSafe [30] only provides context-aware security policies (C). Our IoTMediator allows users to choose a threat-tailored solution for each threat instance. For a meaningful comparison, IoTGuard and IoTSafe are assumed to have a global view and control over multiple platforms although their designs do not have these features. Based on the recorded ground truth in Section 6.2.1, we manually trigger apps and reproduce the interaction in each test group for three times. In every reproduction, we run one of the three approaches (IoTGuard, IoTSafe, and IoTMediator) to handle the interaction. When running IoTGuard and IoTSafe, we choose the appropriate policies from their pre-defined policies. As to IoTMediator, it generates handling options for each threat instance and has the user choose one as the solution.

Table 9 shows the policies/solutions for handling all test groups and the testing results. A test group is considered “handled properly” if the handling matches the user’s preference and is confirmed to cause no safety risks by the researchers. We find that only one test group 11 & 12 in testbed T_1 can be handled properly by the context-aware security policies from IoTSafe and only one test group 3 & 4 in testbed T_2 can be handled properly by the generic policies provided by IoTGuard. This shows that it is very difficult, if not impossible, for security experts to pre-define policies that can handle all the interaction threats because the automation apps and user demands could be very diverse and complex in different smart homes. On the contrary, our IoTMediator generates threat-tailored solutions according to the interaction patterns and the involved apps (and manual controls) in each test group, and hence it can handle interaction threats much more effectively.

6.4 Latency

IoTMediator introduces an extra latency L to the system. The latency L consists of the computation latency L_1 for interaction detection and handling, and the additional transmission delay. The additional transmission delay includes the event transmission delay L_2 from IoTMediator to the platform (including decryption and encryption at the hub) and the command transmission delay in the reverse direction. Thus, L approximately equals to the sum of computation latency L_1 and a round-trip transmission delay $2*L_2$, i.e., $L = L_1 + 2*L_2$. We obtain L_1 by measuring the average computation time it takes for IoTMediator to process an incoming event or command, and obtain L_2 by measuring the elapsed time *from* the moment IoTMediator receives an event *to* the moment the platform receives the event. Three platforms are evaluated in the above way: SmartThings, Alexa and openHAB. Note

Table 8: One-week experiment in realistic settings for the comparison of interaction detection between dynamic approaches (with global view) and ours. N_1 and N_2: the number of times the first and second apps in a test group totally execute. N_{gt}: the number of times a test group causes real interactions. Similar to Table 7, **N/A** means that an approach cannot detect the interaction pattern and "**—**" means that the value cannot be computed due to "divided by zero".

Testbed	**Test Group**	**Interaction Pattern**	N_1	N_2	N_{gt}	**Precision, Recall**		
						IoTGuard [22]	**IoTSafe [30]**	**Ours**
T_1	5 & 7	CAI – Potential RC	6	7	6	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	11 & 12	CAI – Chained Execution	7	1	1	**N/A**	**0.14, 1.00**	**1.00, 1.00**
	13 & 14	CAI – Chained Execution	2	2	2	**1.00, 1.00**	**1.00, 1.00**	**1.00, 1.00**
	3 & 4	CAI – Race Condition	785	214	214	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	3 & 8	CAI – Action Revert	14	8	7	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	31	12	5	**N/A**	**N/A**	**1.00, 1.00**
	9 & 10	CAI – Condition Bypass	79	461	382	**N/A**	**N/A**	**1.00, 1.00**
T_2	set *home* mode & 5	CMAI – Condition Disabling	8	8	2	**N/A**	**N/A**	**1.00, 1.00**
	set *home* mode & 6	CMAI – Condition Enabling	8	398	8	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	33	16	0	**N/A**	**N/A**	**—, —**
	3 & 4	CAI – Infinite Loop	25	19	12	**1.00, 1.00**	**N/A**	**1.00, 1.00**

Table 9: Evaluation results of interaction handling.

Testbed	**Test Group**	**Solutions Provided By Each Work**	**Handled Properly?**		
			IoTGuard [22]	**IoTSafe [30]**	**Ours**
T_1	5 & 7	IoTGuard [22]: (G) two or more events cannot trigger two conflicting actions; (C) N/A. IoTSafe [30]: (C) N/A. Ours: The execution order matters and the final device state should be decided by rule 7. Thus, if action "lock front door ⑭ and turn off outlets ⑪⑫" is executed first, execute it again 30 seconds after "unlock front door ⑭" is executed. (This example is discussed in Section 5.3).	✗	✗	✓
	11 & 12	IoTGuard [22]: (G) a rule action cannot be taken if it triggers another rule; (C) N/A. IoTSafe [30]: All windows ⑲ should be closed when the user ① is away. Ours: Action "turn on the window opener switch ⑲" should not be taken.	✗	✓	✓
	13 & 14	IoTGuard [22]: (G) a rule action cannot be taken if it triggers another rule; (C) N/A. IoTSafe [30]: (C) N/A. Ours: Action "unlock the door ⑭, and turn on lights ⑦⑧⑨⑩ and microwave outlet ⑪" to be taken under the condition: "the user ① is present".	✗	✗	✓
	3 & 4	IoTGuard [22]: (G) the same event cannot trigger two conflicting actions; (C) N/A. IoTSafe [30]: (C) N/A. Ours: When conflicting, action "turn off heater ⑫" should be taken and "turn on heater ⑫" be blocked.	✗	✗	✓
	3 & 8	IoTGuard [22]: (G) a rule action cannot be taken if it triggers another rule; (C) N/A. IoTSafe [30]: N/A. Ours: Allow action "turn off the heater outlet ⑫" to be taken to override "turn on the outlet for heater ⑫" under the condition: "motion ⑤ is inactive".	✗	✗	✓
	1 & 2	IoTGuard [22]: (G) N/A; (C) N/A. IoTSafe [30]: (C) N/A. Ours: Action "turn on floor lamp ⑦ and ceiling lamp ⑧" should be taken.	✗[1]	✗[1]	✓
	9 & 10	IoTGuard [22]: (G) the same event cannot trigger repeated actions; (C) N/A. IoTSafe [30]: (C) N/A. Ours: Take action "turn on the light ⑥" when the time is between 9am and 11pm.	✗	✗	✓
T_2	set *home* mode & 5	IoTGuard [22]: (G) N/A; (C) N/A. IoTSafe [30]: (C) N/A. Ours: Action "turning on camera ⑰" should be taken.	✗	✗	✓
	set *home* mode & 6	IoTGuard [22]: (G) N/A; (C) N/A. IoTSafe [30]: N/A. Ours: Action "turn off camera ⑱" should be taken.	✗	✗	✓
	1 & 2	IoTGuard [22]: (G) N/A; (C) N/A. IoTSafe [30]: (C) N/A. Ours: Do nothing since no real interaction occurs.	✗	✗	✓
	3 & 4	IoTGuard [22]: (G) a rule action cannot be taken if it triggers another rule; (C) N/A. IoTSafe [30]: (C) N/A. Ours: When the two rules form a loop, only action "turning off ceiling lights ①②③" should be taken.	✓	✗	✓

[1] Interactions cannot be handled properly since it cannot be detected correctly in the first place.

Table 10: Average latency introduced by IoTMediator (in seconds).

Platform	Computation Latency L_1	One-Way Transmission Latency L_2	Total Latency L
Alexa	0.236	0.243	0.722
SmartThings	0.236	0.198	0.632
openHAB	0.236	0.144	0.524

that the other two platforms IFTTT and Philips Hue do not provide a convenient way for the researchers to obtain the exact time when they receive an event. We benchmark L_1 and L_2 during the experiments in Sections 6.2 and 6.3.

As shown in Table 10, IoTMediator introduces a total latency of 0.722, 0.632 and 0.524 second on Alexa, SmartThings and openHAB, respectively. Note that users experience little interruption in actual usage because (1) most rules (e.g., Rules with RIDs 3-8 and 11-15 in the testbed T_1, and 4-9 and 11-12 in the testbed T_2) are not time-critical; and (2) the sub-second extra latency is small compared to the original operation time, which ranges from 1-3 seconds in our tests.

7 Limitations and Discussion

Scalability and Extension. This work focuses on smart homes. Large spaces, such as a campus, can be divided into several smaller subspaces, such as buildings/rooms. This way, interaction threats in subspaces can be detected in a scalable way. The challenges are how to properly divide a large space and how to detect threats across subspaces. We leave this as future work. To further increase scalability, the computation can be offloaded from a local hub to one or multiple servers. Moreover, this work considers the most common interaction patterns, where an automation rule interacts with another rule (or manual control). Actually, the interaction effect of two rules can be modeled as a virtual rule, which interacts with another rule. Existing techniques [15] can be applied to studying such special cases of interaction threats.

User efforts. To use IoTMediator, a user needs to change the way of adding new devices and migrate the existing devices to IoTMediator. Specifically, the user first connects a device to IoTMediator, which then creates a virtual device for the device and connects it with the user-specified IoT platform(s). Another effort is that the user needs to choose a handling solution from the recommended options for each detected interaction. A concern is that users are error-prone in making decisions, as they do in configuring automation apps. However, unlike configuring automation apps without a global view, users of IoTMediator are better informed when choosing a solution: they are prompted with the interaction context and the effect of each handling option.

Hybrid threat handling. Our future work is to study how to combine our runtime threat-tailored handling with static-analysis-time handling and generic policies. For example, we consider allowing users to resolve the obviously problematic threat candidates (such as the example shown in Figure 6(c)) all in once with a setup wizard-like UI before the dynamic verification phase, which could reduce user burden at runtime. Moreover, in urgent scenarios, generic safety policies are enforced as soon as possible to get rid of the user response time, and threat-tailored solutions are applied for further handling.

Attack surface. Like many IoT security solutions [22, 25, 30, 39, 53] (such as PFirewall [25], Peekaboo [38], IoTGuard [22] and IoTSafe [30]), IoTMediator adds a mediation module, which could become a potential attack target and a single point of failure. Many existing techniques, such as firewalls and IDS, can be used to enhance the mediator. Note IoTMediator does not introduce new protocols; it uses the same protocols used by IoT hubs to connect IoT devices and those provisioned by platforms to connect virtual devices.

8 Related Work

Smart Home Security. Smart home security has been extensively studied, but not much research has studied the unique threats and challenges raised by multi-platform and multi-control-channel systems. Fernandes et al. [31] and Mi et al. [47] unveil the security vulnerabilities on IoT platforms, SmartThings and IFTTT, respectively. Much work investigates IoT application security [18, 20, 36, 45, 48, 55]. For example, Westworld [45] presents the first dynamic symbolic executor for IoT apps to find their bugs. Solutions have been proposed to enhance IoT authentication [40, 41, 56, 59], privacy [25, 44, 57], voice commands [62], access control [28, 39], firmware [52], anomaly detection [19, 32], etc. Researchers utilize security policies [46] to ensure that devices are in safe states. Yuan et al. [61] report the design flaws in the IoT device access delegation mechanisms across multiple IoT clouds. CGuard [35] highlights an IoT device usually can be controlled through different communication channels, such as Zigbee/ZWave and Bluetooth, and detects inconsistencies between the policies imposed to different communication channels. Fu et al. [33] exploits vulnerable timeout mechanisms of IoT protocol stacks to launch IoT phantom-delay attacks. Recent work [24] presents new interaction-based attacks that exploit different delays on different platforms.

Interaction Threats. Interaction threats draw much research attention. Many works are done to categorize [15, 26, 54], understand [16, 17, 29], detect [14, 15, 21, 22, 26, 27, 34, 37, 43, 49, 50, 54, 60], simulate [27] and handle [42] interaction threats. For instance, HomeGuard [26] is the first that systematically categorizes and formally describes Cross-App Interaction threats. However, they only consider interaction threats in single-platform homes. A few works [22, 23] recognize challenges in multi-platform homes, but they both convert IFTTT rules into equivalent SmartThings apps, and use SmartThings to run all the rules; essentially, they still detect interaction threats in a single-platform home. Our work is the **first** that conducts cross-platform interaction threat detection. Moreover, existing works only detect interactions between automation apps. Our work is the **first** that detects interactions between automation apps and various manual controls. Our work is also the **first** that provides threat-tailored handling.

9 Conclusion

We presented IoTMediator, the first system that detects IoT interaction threats in multi-platform homes. A new family of interaction threats has been identified and studied, which concerns the interaction between manual controls and automation. IoTMediator uses two-way mediation and device ID translation to conduct cross-platform interaction checking. It is also the first system that provides threat-tailored handling. It generates handling options according to threat instance information and interprets the threat context and consequence to users for decision-making. We evaluated IoTMediator with 44 IoT devices, five IoT platforms, and 35 automation apps in two smart-home testbeds, showing that IoTMediator significantly outperforms prior work.

Acknowledgement

We thank the anonymous reviewers for their valuable suggestions. This work was supported in part by the US National Science Foundation (NSF) under grants CNS-2204785, CNS-2205868, CNS-2309477, CNS-2309550 and CNS-2310322.

References

[1] Fragmentation in IoT – one roadblock in IoT deployment. https://www.cleantech.com/fragmentation-in-iot-one-roadblock-in-iot-deployment/, 2017.

[2] IoTBench test suite. https://github.com/IoTBench/IoTBench-test-suite, 2019.

[3] AWS Lambda function. https://aws.amazon.com/lambda/, 2021.

[4] A comprehensive guide to smart home device compatibility. https://www.adt.com/resources/smart-home-device-compatibility, 2021.

[5] Create your service and connect to IFTTT. https://platform.ifttt.com/docs#2-create-your-service-and-connect-to-ifttt, 2021.

[6] IFTTT Realtime API. https://platform.ifttt.com/docs/api_reference#realtime-api, 2021.

[7] Smartthings public github repository. https://github.com/SmartThingsCommunity/SmartThingsPublic, 2021.

[8] Amazon Alexa. https://developer.amazon.com/en-US/alexa/devices/smart-home-devices, 2022.

[9] Google Home - your smart home just got even smarter. https://home.google.com/the-latest/, 2022.

[10] IFTTT. https://ifttt.com/discover, 2022.

[11] openHAB – an open-source platform for empowering home automation. https://www.openhab.org/, 2022.

[12] Philips Hue. https://www.philips-hue.com/en-us, 2022.

[13] SmartThings. https://www.smartthings.com/, 2022.

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

[15] Mohannad Alhanahnah, Clay Stevens, Bocheng Chen, Qiben Yan, and Hamid Bagheri. IoTCom: Dissecting interaction threats in IoT systems. *IEEE TSE*, 2022.

[16] Musard Balliu, Massimo Merro, and Michele Pasqua. Securing cross-app interactions in IoT platforms. In *IEEE CSF*, 2019.

[17] Musard Balliu, Massimo Merro, Michele Pasqua, and Mikhail Shcherbakov. Friendly fire: cross-app interactions in IoT platforms. *ACM TOPS*, 24(3):1–40, 2021.

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

[19] Simon Birnbach, Simon Eberz, and Ivan Martinovic. Peeves: Physical event verification in smart homes. In *ACM CCS*, 2019.

[20] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity IoT. In *USENIX Security Symposium*, 2018.

[21] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated IoT safety and security analysis. In *USENIX Security Symposium*, 2018.

[22] Z Berkay Celik, Gang Tan, and Patrick McDaniel. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *NDSS*, 2019.

[23] Zhao Chen, Fanping Zeng, Tingting Lu, and Wenjuan Shu. Multi-platform application interaction extraction for IoT devices. In *IEEE ICPADS*, 2019.

[24] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Delay wreaks havoc on your smart home: Delay-based automation interference attacks. In *Oakland*, 2022.

[25] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. PFirewall: Semantics-aware customizable data flow control for smart home privacy protection. In *NDSS*, 2021.

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

[27] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering end users in debugging trigger-action rules. In *CHI*, 2019.

[28] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A Gunter, Xiaoyong Zhou, and Michael Grace. HanGuard: SDN-driven protection of smart home wifi devices from malicious mobile apps. In *ACM WiSec*, 2017.

[29] Wenbo Ding and Hongxin Hu. On the safety of IoT device physical interaction control. In *ACM CCS*, 2018.

[30] Wenbo Ding, Hongxin Hu, and Long Cheng. IoTSafe: Enforcing safety and security policy with real IoT physical interaction discovery. In *NDSS*, 2021.

[31] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *Oakland*, 2016.

[32] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. HAWatcher: Semantics-aware anomaly detection for appified smart homes. In *USENIX Security*, 2021.

[33] Chenglong Fu, Qiang Zeng, Xiaojiang Du, and Siva Likitha Valluru. IoT phantom-delay attacks: Demystifying and exploiting IoT timeout behaviors in smart homes. In *DSN*, 2022.

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

[35] Yan Jia, Bin Yuan, Luyi Xing, Dongfang Zhao, Yifan Zhang, XiaoFeng Wang, Yijing Liu, Kaimin Zheng, Peyton Crnjak, Yuqing Zhang, et al. Who's in control? on security risks of disjointed IoT device management channels. In *ACM CCS*, 2021.

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

[37] Keyu Jiang, Hanyi Zhang, Weiting Zhang, Liming Fang, Chunpeng Ge, Yuan Yuan, and Zhe Liu. TapChain: A rule chain recognition model based on multiple features. *Security and Communication Networks*, 2021.

[38] Haojian Jin, Gram Liu, David Hwang, Swarun Kumar, Yuvraj Agarwal, and Jason I Hong. Peekaboo: A hub-based approach to enable transparency in data processing within smart homes. In *Oakland*, 2022.

[39] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. FACT: Functionality-centric access control system for IoT programming frameworks. In *ACM Symposium on Access Control Models and Technologies*, 2017.

[40] Xiaopeng Li, Fengyao Yan, Fei Zuo, Qiang Zeng, and Lannan Luo. Touch well before use: Intuitive and secure authentication for IoT devices. In *MobiCom*, 2019.

[41] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. T2pair: Secure and usable pairing for heterogeneous IoT devices. In *ACM CCS*, 2020.

[42] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. Systematically debugging IoT control system correctness for building automation. In *ACM International Conference on Systems for Energy-Efficient Built Environments*, 2016.

[43] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: building an internet of safe things. In *ACM IPSN*, 2015.

[44] Xuanyu Liu, Qiang Zeng, Xiaojiang Du, Siva Likitha Valluru, Chenglong Fu, Xiao Fu, and Bin Luo. Sniffmislead: Non-intrusive privacy protection against wireless packet sniffers in smart homes. In *RAID*, 2021.

[45] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart apps on IoT cloud platforms. In *ACSAC*, 2021.

[46] Sunil Manandhar, Kevin Moran, Kaushal Kafle, Ruhao Tang, Denys Poshyvanyk, and Adwait Nadkarni. Towards a natural perspective of smart homes for practical security and safety analyses. In *Oakland*, 2020.

[47] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *Internet Measurement Conference*, 2017.

[48] Richard Mitev, Markus Miettinen, and Ahmad-Reza Sadeghi. Alexa lied to me: Skill-based man-in-the-middle attacks on virtual assistants. In *ACM Asia CCS*, 2019.

[49] Julie L Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. IoTA: a calculus for internet of things automation. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017.

[50] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. IotSan: fortifying the safety of IoT systems. In *ACM CoNEXT*, 2018.

[51] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chun-An Chu, Z Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. Discovering physical interaction vulnerabilities in IoT deployments. *arXiv preprint arXiv:2102.01812*, 2021.

[52] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing vulnerable home IoT devices with an in-hub security manager. In *IEEE PerCom Workshops*, 2017.

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

[54] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action IoT platforms. In *ACM CCS*, 2019.

[55] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *NDSS*, 2018.

[56] Chuxiong Wu, Xiaopeng Li, Fei Zuo, Lannan Luo, Xiaojiang Du, Jia Di, and Qiang Zeng. *Use It-No Need to Shake It!* accurate implicit authentication for everyday objects with smart sensing. *IMWUT*, 6(3):1–25, 2022.

[57] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 7:63457–63471, 2019.

[58] Moosa Yahyazadeh, Proyash Podder, Endadul Hoque, and Omar Chowdhury. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *ACM Symposium on Access Control Models and Technologies*, 2019.

[59] Heng Ye, Qiang Zeng, Jiqiang Liu, Xiaojiang Du, and Wei Wang. Easy peasy: A new handy method for pairing multiple cots IoT devices. *IEEE TDSC*, 2022.

[60] Yinbo Yu and Jiajia Liu. TAPInspector: Safety and liveness verification of concurrent trigger-action IoT systems. *IEEE TIFS*, 2022.

[61] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, Xiaofeng Wang, Deqing Zou, Hai Jin, and Yuqing Zhang. Shattered chain of trust: Understanding security risks in cross-cloud IoT access delegation. In *USENIX Security Symposium*, 2020.

[62] Qiang Zeng, Jianhai Su, Chenglong Fu, Golam Kayas, Lannan Luo, Xiaojiang Du, Chiu C Tan, and Jie Wu. A multiversion programming inspired approach to detecting audio adversarial examples. In *DSN*, 2019.

[63] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. HoMonit: Monitoring smart home apps from encrypted traffic. In *ACM CCS*, 2018.

A Implementation Details of Messenger

The device virtualization module handles interactions between virtual devices and platforms, including adding/removing devices to/from platforms, pushing device events to platforms, responding to the platform's pulling for device states, and receiving commands from platforms. Each commodity integration platform supports at least one technique for handling the above device-platform interactions. A prior work PFirewall [25] has implemented and proven the feasibility of SmartThings [13] and openHAB [11]. This work extends the device virtualization to another three popular IoT platforms: Alexa [8], Philips Hue [12], IFTTT [10]. We will present the details of the three platforms and refer the interested readers to the literature [25] for those of SmartThings and openHAB.

Alexa. We achieve the integration with Alexa by exposing REST APIs on Messenger and developing a Lambda function on AWS [3]. The Lambda function receives the `authorization` (for token exchanging), `discovery`, `reportstate` and `interface` (for controlling devices) directives from the Alexa cloud and then communicate with Messenger via the REST APIs to execute the directives. After receiving responses from Messenger, the Lambda function responds to the Alexa cloud with the results. In addition, Messenger can push device events asynchronously to the event gateway on the Alexa cloud, without the mediation of the Lambda function. This above processes are similar to those of SmartThings except that an extra hop (i.e., Lambda function) is introduced.

IFTTT. To connect with IFTTT, Messenger is implemented as a *service* [5] that exposes endpoints for IFTTT to query. The *trigger* endpoints allow IFTTT to query the most recent 50 instances of a specific event type (e.g., motion active) and the *action* endpoints enable IFTTT to control a device by sending requests; in these requests, IFTTT specifies parameters including device identifier, attribute name and value. Typically, IFTTT polls the trigger endpoints every 15 minutes, which hinders the timely capture of new events and imposes large latency to automation. To address this, Messenger notifies IFTTT of any new trigger-related event type through the Realtime API [6] and then IFTTT will poll the corresponding trigger endpoint to acquire the data.

Philips Hue. Different from the above platforms, Philips Hue connects its devices and runs automation apps on the local Hue bridge (i.e., hub). The Hue cloud mainly provides the messaging function, delegating devices (connected to the Hub bridge) to other clouds such as SmartThings, Alexa, IFTTT. Both the Hue bridge and cloud support device-accessing APIs. Messenger access Hue devices by accessing the APIs provided by the Hue bridge on the local network. That is to say, Messenger cannot intervene in the original communication between the Hue devices and platform (i.e., the bridge and cloud). However, Messenger still has the view and control on the Hue devices through the APIs.

B Dynamic Verification of Other Patterns

In this appendix, we present the dynamic verification process of interaction patterns that are not discussed in the main text.

CAI - Condition Disabling. IoTMediator listens for the first event $\mathcal{E}(T_1)$. If observed, $\mathcal{E}(T_1)$ will trigger R_1 when it arrives the platform running R_1. To determine if R_1 will pass its condition checking, IoTMediator checks if R_1's condition C_1 satisfies by querying the local database since it synchronizes with the remote database maintained by the platform running R_1. If true, IoTMediator continues waiting for the command $\mathcal{C}(A_1)$ issued by R_1. When $\mathcal{C}(A_1)$ is observed, IoTMediator checks if R_2's condition C_2 was true before forwarding the command. After forwarding the command, IoTMediator will observe a new event $\mathcal{E}(\neg C_2)$ which makes C_2 turns from true to false, as indicated by $A_1 \Rightarrow C_2$ in the candidate screening. Thus, IoTMediator keeps monitoring if C_2 remains false until observing an event $\mathcal{E}(T_2)$ that triggers R_2. If so, a condition disabling instance between R_1 and R_2 is identified in the runtime and this CAI candidate is verified to be a real interaction threat.

$\texttt{obs}\,\mathcal{E}(T_1), \texttt{match}\,\mathcal{S}(C_1)$, /* R_1 will execute */
$\texttt{obs}\,\mathcal{C}(A_1), \texttt{match}\,\mathcal{S}(C_2)$, /* C_2 was true before R_1 executes */
$\texttt{obs}\,\mathcal{E}(\neg C_2) \xrightarrow{\texttt{match}\,\mathcal{S}(\neg C_2)}$ /* C_2 becomes and remains false */
$\texttt{obs}\,\mathcal{E}(T_2)$. /* Until R_2 is triggered */

CAI - Condition Enabling. The verification of a condition enabling candidate is highly similar to that of condition disabling, except that it checks if R_1 enables R_2's condition rather than disables, as presented below.

$\texttt{obs}\,\mathcal{E}(T_1), \texttt{match}\,\mathcal{S}(C_1)$, /* R_1 will execute */
$\texttt{obs}\,\mathcal{C}(A_1), \texttt{match}\,\mathcal{S}(\neg C_2)$, /* C_2 was false before R_1 executes */
$\texttt{obs}\,\mathcal{E}(C_2) \xrightarrow{\texttt{match}\,\mathcal{S}(C_2)}$ /* C_2 becomes and remains true */
$\texttt{obs}\,\mathcal{E}(T_2)$. /* Until$R_2$ is triggered */

CAI - Race Condition. When observing the event $\mathcal{E}(T_1)$ which triggers both rules R_1 and R_2, IoTMediator checks if both conditions C_1 and C_2 are true. If so, both rules will proceed to take contradictory actions upon their platforms receive the trigger event, i.e., the candidate R_1 and R_2 are verified to cause a real threat.

$\texttt{obs}\,\mathcal{E}(T_1)$, /* Both rules will be triggered */
$\texttt{match}\,\mathcal{S}(C_1), \texttt{match}\,\mathcal{S}(C_2)$ /* Both rule conditions are true */

CAI - Condition Bypass. To verify a condition bypass candidate in the real time, IoTMediator only needs to verify that when both rules R_1 and R_2 are triggered by the same event, the condition of one rule is evaluated to be `true` while that of another rule `false`, i.e., the exclusive or $\oplus$ of the evaluations of both conditions yields `true`. The symbolic representation of the dynamic verification process is shown below.

$\texttt{obs}\,\mathcal{E}(T_1)$, /* Both rules will be triggered */
$\texttt{match}\,\mathcal{S}(C_1) \oplus \texttt{match}\,\mathcal{S}(C_2) = \texttt{true}$ /*One condition holds*/

CAI - Infinite Loop. The dynamic verification of infinite loop, i.e., whether two rules R_1 and R_2 triggers each other alternately, doubles the steps of chained execution. Suppose infinite loop starts from R_1, the process is shown below. Note that IoTMediator also verifies the case where infinite loop starts from R_2, which is symmetric to the former case.

$\texttt{obs}\,\mathcal{E}(T_1), \texttt{match}\,\mathcal{S}(C_1), \texttt{obs}\,\mathcal{C}(A_1)$, /* R_1 executes */
$\texttt{match}\,\mathcal{S}(\neg T_2)$, /* The trigger of R_2 was false */
$\texttt{obs}\,\mathcal{E}(T_2), \texttt{match}\,\mathcal{S}(C_2), \texttt{obs}\,\mathcal{C}(A_2)$, /*$R_1$ triggers R_2*/
$\texttt{match}\,\mathcal{S}(\neg T_1)$, /* The trigger of R_1 was false */
$\texttt{obs}\,\mathcal{E}(T_1), \texttt{match}\,\mathcal{S}(C_1)$ /*R_1 will be triggered in turn by R_2*/

CMAI - Potential Race Condition. Manual control typically has a higher priority than automation since it allows a user to set devices to the desirable state (including overriding an automation result). Consider the example shown in Figure 8(b): a user wants to use a manual command to stop the alarm after the automation app sounds the alarm upon the detection of a kitchen smoke. However, the user is annoyed if the automation app triggers and sounds the alarm again and again within a short period after the user stops the alarm. Thus, we only consider it as an interaction threat when an automation app R_3 runs after c and consequently overrides the manual control.

$\texttt{obs}\,\mathcal{C}(c)$ /* Observe a manual control c */
$\xrightarrow{\texttt{match}\,\mathcal{S}(c)}$ /* The state changed by c remains unchanged */
$\texttt{obs}\,\mathcal{E}(T_3), \texttt{match}\,\mathcal{S}(C_3)$ /* UntilR_3 executes */

CMAI - Condition Enabling. To verify a candidate, the dynamic verification component checks if a manual control c changes an automation rule R_3's condition C_3 from false to true and then if C_3 remains true until the rule R_3 is triggered. If so, the candidate is verified to cause a real interaction threat and vice versa.

$\texttt{obs}\ \mathcal{C}(c)$, /* Observe a manual control c */
$\texttt{match}\,\mathcal{S}(\neg C_3)$, /* The condition of R_3 was false */
$\texttt{obs}\ \mathcal{E}(C_3)$ /* c yields an event which makes R_3's condition true */
$\xrightarrow{\texttt{match}\,\mathcal{S}(C_3)}$ /* The condition of R_3 remains true */
$\texttt{obs}\ \mathcal{E}(T_3)$ /* Until R_3 is triggered */

CMAI - Condition Disabling. Similar to the condition enabling case, the dynamic verification component inspects if c changes C_3 from true to false and then if C_3 remains false until the rule R_3 is triggered. If so, the candidate if verified and otherwise IoTMediator continues to verify this candidate in the next observation of the manual control c.

$\texttt{obs}\ \mathcal{C}(c)$, /* Observe a manual control c */
$\texttt{match}\,\mathcal{S}(C_3)$, /* The condition of R_3 was true */
$\texttt{obs}\ \mathcal{E}(\neg C_3)$ /* c yields an event which makes R_3's condition false */
$\xrightarrow{\texttt{match}\,\mathcal{S}(\neg C_3)}$ /* The condition of R_3 remains false */
obs $\mathcal{E}(T_3)$ /* Until R_3 is triggered */

Table 11: Handling options and explanation templates for interaction threat patterns that are not listed in Table 4 in the main text.

Interaction Pattern (*P*)	**Handling Options & Explanation Templates**
CAI - Condition Bypass	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\backslash\mathcal{S}(C_2),\rightarrow\mathcal{C}(A_1)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_2)\backslash\mathcal{S}(C_1),\rightarrow\mathcal{C}(A_2)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\cap\mathcal{S}(C_2),\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Execute action A_1 when either or both of the two rules are triggered.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\backslash\mathcal{S}(C_2),\nrightarrow\mathcal{C}(A_1)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_2)\backslash\mathcal{S}(C_1),\nrightarrow\mathcal{C}(A_2)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\cap\mathcal{S}(C_2),\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Execute action A_1 when the conditions of both rules are true.
	Option 3: $\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\backslash\mathcal{S}(C_2),\rightarrow\mathcal{C}(A_1)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_2)\backslash\mathcal{S}(C_1),\nrightarrow\mathcal{C}(A_2)\rangle+\langle\langle R_1,R_2,P\rangle,\mathcal{S}(C_1)\cap\mathcal{S}(C_2),\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Let the first rule work and disable the second.
CAI - Condition Enabling	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Action A_2 should not be executed.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\emptyset,\emptyset\rangle$
	Explanation Template: Action A_2 should be executed.
CAI - Condition Disabling	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\Rightarrow\mathcal{S}(A_2)\rangle$
	Explanation Template: Action A_2 should be executed.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\emptyset,\emptyset\rangle$
	Explanation Template: Action A_2 should not be executed.
CAI - Race Condition	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\nrightarrow\mathcal{C}(A_1)\rangle$
	Explanation Template: When the two rules conflict, action A_2 should be executed and A_1 should be blocked.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\emptyset,\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: When the two rules conflict, action A_1 should be executed and A_2 should be blocked.
CAI - Action Revert	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\nrightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Action A_2 should not be executed to override A_1.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\emptyset,\rightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Action A_2 should be executed to override A_1.
	Option 3: $\langle\langle R_1,R_2,P\rangle,\texttt{cond}\subseteq\mathcal{V},\rightarrow\mathcal{C}(A_2)\rangle$
	Explanation Template: Allow action A_2 to be executed to override A_1 under a certain condition `cond` (`cond` is configurable and can be specific device states and/or time period).
CAI - Infinite Loop	**Option 1:** $\langle\langle R_1,R_2,P\rangle,\emptyset,\Rightarrow\mathcal{S}(A_1)\rangle$
	Explanation Template: When the two rules form a loop, only action A_1 should be executed.
	Option 2: $\langle\langle R_1,R_2,P\rangle,\emptyset,\Rightarrow\mathcal{S}(A_2)\rangle$
	Explanation Template: When the two rules form a loop, only action A_2 should be executed.
CMAI - Condition Enabling	**Option 1:** $\langle\langle c,R_3,P\rangle,\emptyset,\nrightarrow\mathcal{C}(A_3)\rangle$
	Explanation Template: Action A_3 should not be executed.
	Option 2: $\langle\langle c,R_3,P\rangle,\emptyset,\emptyset\rangle$
	Explanation Template: Action A_3 should be executed.

Table 12: Microbench experiment for the performance comparison of interaction detection between prior approaches (**no global view**) and ours. **N/A** denotes that a work does not consider a specific interaction pattern or its instances are all caused by cross-platform interaction (and, hence, the work cannot detect them without a global view). "**—**" means that the value cannot be computed due to "divided by zero".

Testbed	**Test Group**	**Interaction Pattern**	N_{all}	N_d	**Precision, Recall**				
					HomeGuard [26]	**iRuler [54]**	**IoTGuard [22]**	**IoTSafe [30]**	**Ours**
T_1	5 & 7	CAI – Potential RC	24	6	**0.25, 1.00**	**N/A**	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	11 & 12	CAI - Chained Execution	128	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	13 & 14	CAI – Chained Execution	256	63	**0.25, 1.00**	**N/A**	**1.00, 1.00**	**0.25, 1.00**	**1.00, 1.00**
	3 & 4	CAI - Race Condition	16	4	**N/A**	**N/A**	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	3 & 8	CAI - Action Revert	16	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	16	1	**0.06, 1.00**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	9 & 10	CAI - Condition Bypass	8	1	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
T_2	set *home* mode & 5	CMAI - Condition Disabling	8	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	set *home* mode & 6	CMAI - Condition Enabling	8	2	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	16	0	**0.00, —**	**0.00, —**	**N/A**	**N/A**	**—, —**
	3 & 4	CAI - Infinite Loop	192	6	**N/A**	**N/A**	**N/A**	**N/A**	**1.00, 1.00**

Table 13: One-week experiment for the performance comparison of interaction detection between prior dynamic approaches (**no global view**) and ours.

Testbed	**Test Group**	**Interaction Pattern**	N_1	N_2	N_{gt}	**Precision, Recall**		
						IoTGuard [22]	**IoTSafe [30]**	**Ours**
T_1	5 & 7	CAI – Potential RC	6	7	6	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	11 & 12	CAI – Chained Execution	7	1	1	**N/A**	**N/A**	**1.00, 1.00**
	13 & 14	CAI – Chained Execution	2	2	2	**1.00, 1.00**	**1.00, 1.00**	**1.00, 1.00**
	3 & 4	CAI – Race Condition	785	214	214	**1.00, 1.00**	**N/A**	**1.00, 1.00**
	3 & 8	CAI – Action Revert	14	8	7	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	31	12	5	**N/A**	**N/A**	**1.00, 1.00**
	9 & 10	CAI – Condition Bypass	79	461	382	**N/A**	**N/A**	**1.00, 1.00**
T_2	set *home* mode & 5	CMAI – Condition Disabling	8	8	2	**N/A**	**N/A**	**1.00, 1.00**
	set *home* mode & 6	CMAI – Condition Enabling	8	398	8	**N/A**	**N/A**	**1.00, 1.00**
	1 & 2	CAI – Condition Disabling	33	16	0	**N/A**	**N/A**	**—, —**
	3 & 4	CAI – Infinite Loop	25	19	12	**N/A**	**N/A**	**1.00, 1.00**

C Detection Results without Global View

The results in Table 7 and 8 are obtained by assuming that prior systems also have a global view over the multiple platforms. As prior systems actually do not present a way of obtaining a global view over the multiple platforms, we examine their performance without a global view. The results of *microbench* and *one-week* experiments in this setting are shown in Table 12 and 13, respectively, illustrating that prior systems have poor performance in interaction detection for the multi-platform smart homes. The results highlight one of the main contributions of our work on cross-platform IoT interaction threat detection.