Detection and Mitigation of Rule Conflicts in Smart Home

Yaodong Zhang*
* School of Cyber Science and Technology, Beihang University,
Beijing 100191, China (e-mail: cstzyd@buaa.edu.cn)

Abstract—The rapid advancement of the Internet of Things (IoT) has led to the increasing prevalence of the smart home, which heavily relies on automation rules, typically modeled as Trigger-Condition-Action (TCA). However, the complex interaction among multiple rules can lead to outcomes unintended by users — known as rule conflicts — which inherently pose unpredictable security risks. Existing rule conflict detection approaches often suffer from low efficiency, a lack of dynamic monitoring capabilities for real-time conflicts, and high rates of false negatives and false positives. Furthermore, associated conflict mitigation methods are generally limited, typically involving manual modifications to rule configurations or adherence to generic security policies (such as operation blacklists or whitelists), thus lacking customized handling strategies. This paper proposes a novel framework that integrates offline static rule analysis, runtime dynamic monitoring, and a customized handling strategy to achieve real-time dynamic monitoring and conflict mitigation of rule conflicts in the smart home system. In the static analysis phase, we first design a novel rule model, and by analyzing rule interaction patterns, we generate a Rule Interaction Dependency Graph (where automation rules are nodes and interaction patterns are edges). We then employ a security-level-based formal verification method to identify candidate rule conflicts and extract these conflict relations to form a Rule Conflict Dependency Graph (a subgraph of the former). Finally, we utilize a heuristic approach to generate customized handling strategies for these conflicts. For the runtime dynamic monitoring component, the system continuously captures rule events. For any triggered rule, we identify its corresponding node in the Rule Conflict Dependency Graph and check its neighboring nodes (i.e., rules with direct conflict dependency). Based on these potential conflict relations, we apply an assertion verification mechanism to determine whether the dependency edge from the current triggered rule to a neighbor rule is activated under the current system state. If the assertion verification holds, confirming that the necessary condition for a conflict is met, the system immediately intercepts the default rule actions and executes the pre-generated customized handling strategy. This mechanism effectively prevents conflicts before they manifest. We implement and evaluate our system using a simulated Home Assistant environment containing 17 rules across multiple zones. The results demonstrate that our framework successfully identifies all manually verified conflicts in our dataset, including those caused by the physical channels that other tools often overlook, while significantly reducing the false-positive rate. Our system automatically generates feasible, customized conflict handling strategies and exhibits high performance: the static analysis component finishes within seconds, even for large rule sets, and the dynamic overhead remains at the millisecond level. Our approach provides a comprehensive, effective, and personalized solution for the mitigation of rule conflicts.

Index Terms—Smart home, Rule Conflict, Conflict Detection, Conflict Mitigation

1 Introduction

The smart home represents a crucial application domain of the Internet of Things (IoT), providing unprecedented convenience and comfort to users seeking a higher quality of life by interconnecting various smart devices. The core driving force behind its intelligence lies in automation rules, typically adopting the Trigger-Condition-Action (TCA) model. Through user-defined or pre-configured automation rules, the smart home can sense environmental changes and autonomously execute tasks, greatly simplifying users' daily routines. The Trigger defines the event that initiates the rule; the Condition is the prerequisite judgment for executing the action after the trigger event occurs; and the Action is the specific operation performed when the condition is met. For instance, a rule designed to maintain a constant temperature can be expressed as: ⟨ Indoor temperature is below 24°C(Trigger), Air conditioner is off (Condition), Turn on the air conditioner and set to 27°C(Action) ⟩.

However, as the number of rules deployed in the home increases, this convenience is often accompanied by hidden dangers. In complex smart home systems, the concurrent execution of multiple rules can lead to unexpected rule interactions, subsequently resulting in serious consequences known as Rule Conflicts. These conflicts not only disrupt the intended automation flow but may also cause security threats or physical damage. As shown in Figure 1, Rule R1 (Smoke triggers sprinkler) and Rule R2 (Water leak triggers water valve closure) are individually logical when running independently. Yet, when they coexist, the execution of R1 triggers R2, leading to the closing of the water valve, which in turn cuts off the water supply to the sprinkler system, directly hindering firefighting efforts. This system-level anomaly, arising from the interaction of individually legitimate rules, represents a major challenge facing current smart home security.

In response to this issue, extensive research has been conducted in academia [1]–[27]. Existing detection methods are primarily categorized into two types: Security Policy-based and Interaction Pattern-based. The former relies on predefined security rules (e.g., "The door must be closed when no one is home") to detect violations [4], [15], [23]; the latter identifies specific rule interaction patterns (e.g., cyclic triggering) to pre-warn potential risks [3], [15], [22], [24].

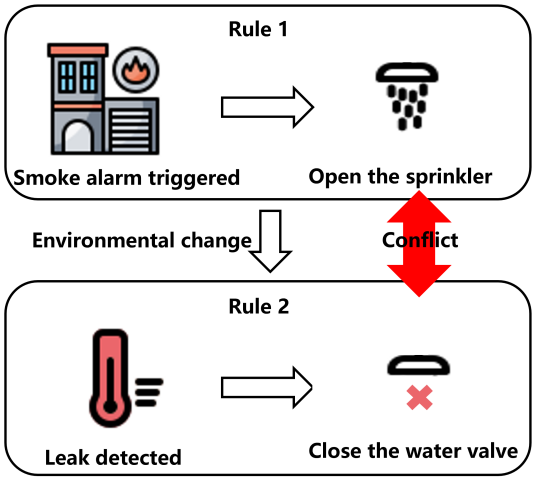


Fig. 1. Rule Conflict Example

Regarding Conflict Mitigation, mainstream approaches include static Rule Reconfiguration, enforcing generic policies, or customized handling strategies for specific scenarios.

Despite the progress achieved by existing solutions, severe challenges remain, mainly reflected in the difficulty of balancing detection accuracy and handling flexibility. Firstly, concerning rule conflict detection: 1) Generic policies lack personalization. The smart home environment is highly heterogeneous, and user preferences are intensely subjective. The same rule interaction (e.g., smoke triggering the sprinkler) might have drastically different security implications in different homes (e.g., a home with a live-fire fireplace vs. a typical home). A one-size-fits-all strategy leads to a high rate of false positives or false negatives. 2) Limitations in physical channel awareness. Many rules interact indirectly through the physical environment (e.g., temperature, light). Ignoring zone characteristics (e.g., thermal isolation between the bedroom and the living room) can lead to incorrect judgments about the interaction path. For example, SafeChain [26], while claiming to be dynamic, is primarily based on static model updates, lacking real-time perception of runtime physical states; while IoTGuard [15] overlooks the complexity of the physical channel, focusing only on application-level action conflicts.

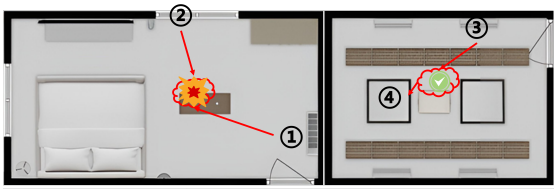
Secondly, concerning conflict mitigation: 1) Fragility of static fixes. Simple rule reconfiguration (e.g., deleting a rule) often breaks the user's intended automation functionality, or even introduces new problems. 2) Coarse granularity of dynamic interception. Existing dynamic systems like IoTSafe [4] can perform preventative interception but often employ a generic blocking strategy, lacking fine-grained, customized handling strategy capabilities for different conflict types. Although IoTMediator [22] attempted customized handling of rule conflicts, it essentially sets up different conflict stage templates for different rule conflict patterns and requires active user selection, lacking an automated execution method. Furthermore, this work still falls short in distinguishing between "benign rule interaction" and "malicious rule conflict."

This paper proposes a solution based on the following core observations: (1) Rule interaction exists not only at the logical layer but also propagates through the physical channel across different zones; thus, an enhanced model incorporating zone characteristics must be established. (2) The judgment of rule conflict is subjective; we must distinguish between objective "rule interaction" and subjectively harmful "rule conflict." (3) The runtime characteristics of the smart home allow us to intervene within the millisecond window before a conflict manifests. Based on these, we propose a hybrid framework combining the comprehensiveness of static analysis with the real-time nature of dynamic monitoring. We first construct a Rule Interaction Dependency Graph (RIDG) through formal verification, extract the Rule Conflict Dependency Graph (RCDG) using user-defined entity security configurations, and generate a customized handling strategy for each conflict. At runtime, the system utilizes an assertion verification mechanism to monitor the real-time activation of conflict paths, and a controller performs precise interception.

In summary, our main contributions are as follows:

- We propose a novel method for rule conflict classification based on physical channel awareness and zone characteristics. By introducing the TCAE model and the Rule Interaction Dependency Graph (RIDG) structure, we cover all types of rule interaction and clearly delineate the boundary between "interaction" and "conflict," effectively solving the problems of false negatives and false positives caused by the physical channel.
- We design and implement a hybrid conflict management framework. The static phase performs precise conflict pre-screening combined with entity security configurations; the dynamic phase utilizes an efficient assertion verification mechanism to monitor and confirm the occurrence of conflicts in real-time at runtime, achieving low-overhead, precise detection.
- We implement an automated and personalized conflict mitigation mechanism. The system can automatically generate customized handling strategies based on the conflict type and security configuration, and perform preventative interception at the critical moment of conflict occurrence, thus ensuring safety while maximizing system usability.

2 Motivating Example and Problem Analysis



Serial number	Device	Zone
①	Heater	Bedroom
②	Window	Bedroom
③	Heater	Greenhouse
④	Skylight	Greenhouse

Fig. 2. Motivating Example

This section provides a concrete example to illustrate the manifestation of rule conflicts and analyzes the limitations of existing conflict detection and mitigation methods. **Fig**ure 2 shows the floor plan of two zones within a smart home system: the bedroom (left) and the greenhouse (right). ① is the heater in the bedroom, ② is the window in the bedroom, ③ is the heater in the greenhouse, and ④ is the skylight in the greenhouse.

Consider the following two automation rules set in the greenhouse: Rule 1 (R1): When sunset is detected (trigger), turn on the heater (action); Rule 2 (R2): When the temperature reaches 30°C(trigger), open the window and turn off the heater (action).

2.1 Challenge: Diversity of Conflict Influencing Factors

Defining rule conflicts in the smart home environment is highly complex. Unlike traditional software logic errors, whether the interaction between automation rules constitutes a ”conflict” depends not only on the logical structure of the rules themselves but is also significantly influenced by various external factors. Existing research often struggles to comprehensively capture these diverse influencing factors, leading to high rates of false positives or false negatives in conflict detection. Specifically, these challenges are manifested in the following three aspects:

1. Difficulty in Capturing User Preferences (Subjectivity). The definition of a rule conflict largely depends on the subjective intent of the user. The same rule interaction pattern may be an expected feature for some users while posing a serious threat to others. As shown in Figure 2, when R1 and R2 are deployed in the greenhouse, R1’s heating causes the temperature to rise, thereby triggering R2 to open the skylight. This is typically a closed-loop control intentionally designed by the user to maintain a constant temperature. However, if the same rules are applied to the bedroom (e.g., R1 controls the bedroom heater, and R2 controls the bedroom window), opening the window automatically at night poses a significant security risk, and this ”unintended consequence” constitutes a rule conflict. Furthermore, for specific groups, differences in preferences are even more critical. For instance, patients with photosensitive epilepsy are extremely sensitive to light changes. If the interaction of two rules causes lights to flicker frequently (a deadlock loop), this may be a mere nuisance to ordinary users but a severe safety threat to such patients. Existing methods struggle to capture such fine-grained and subjective user preferences.

2. Smart Home System Diversity. Automation rules often interact indirectly through the physical environment (i.e., physical channels), and this interaction is highly dependent on the physical location of devices and zonal characteristics. For example, R1 turning on the heater causes the temperature to rise. If the trigger condition of R2 is ”Open the window when the living room temperature is above 30°C,” and the heater controlled by R1 is located in the bedroom, the execution of R1 will not actually trigger R2 due to the thermal isolation between the bedroom and the living room. However, if this physical isolation characteristic of the ”Zone” is ignored and reliance is placed solely on the logical inference that ”heater causes temperature rise,” the detection system will generate false positives. Many current methods lack the capability to precisely model such complex physical environments and regional characteristics.

3. Difficulty in Capturing Entity Safety Configurations. Determining whether a rule interaction is harmful typically requires reliance on security policies. Previous methods depended on generic security policies (e.g., ”The door must remain locked when no one is at home”). However, the smart home environment is personalized, and generic security policies cannot cover all scenarios. For ordinary users, manually defining a detailed Entity Safety Configuration for every device in the home is not only too high a barrier but also extremely cumbersome. The lack of accurate entity safety configurations tailored to a specific home environment makes it difficult for the system to distinguish between permissible device state changes and dangerous rule conflicts.

2.2 Our Solution: User-Centric Configuration and Interaction

To address the aforementioned challenge of the diversity of conflict influencing factors, this paper proposes a solution that combines refined configuration with user-friendly interaction.

First, we introduce an environment and safety modeling approach based on configuration files. To address the issue of system diversity, we design a Physical Channel Configuration method that includes Zone, Channel, and Trend. This allows the system to accurately understand that a ”bedroom heater” only affects the ”bedroom temperature.” Simultaneously, to address the lack of security policies, we introduce the Entity Safety Configuration, which explicitly defines the preferred safety state of critical devices when a conflict occurs (e.g., a window tends to be ”Closed”).

Second, to address the difficulty in capturing user preferences and the complexity of cofiguration, we adopt a user-friendly interactive approach. We utilize natural language processing techniques to translate obscure rule codes and potential interaction relationships into easily understandable natural language descriptions. For instance, instead of displaying complex logical symbols, the system presents to the user: ”It is detected that the execution of Rule R1 (If sunset, turn on heater) may unexpectedly trigger Rule R2 via the Temperature channel.” This enables ordinary users to easily comprehend the consequences of rule interactions and perform secondary confirmation based on personal preferences (e.g., whether to accept automatic window opening in the greenhouse). Through this ”User-in-the-loop” approach, we effectively resolve the ambiguity caused by subjectivity, ensuring the accuracy and personalization of conflict detection.

3 Design

3.1 Overview

We propose a novel hybrid framework designed to handle rule conflicts in the smart home through a combination of static analysis and dynamic monitoring, automatically executing customized handling strategies before the conflict manifests. As shown in Figure 3, the system is divided into

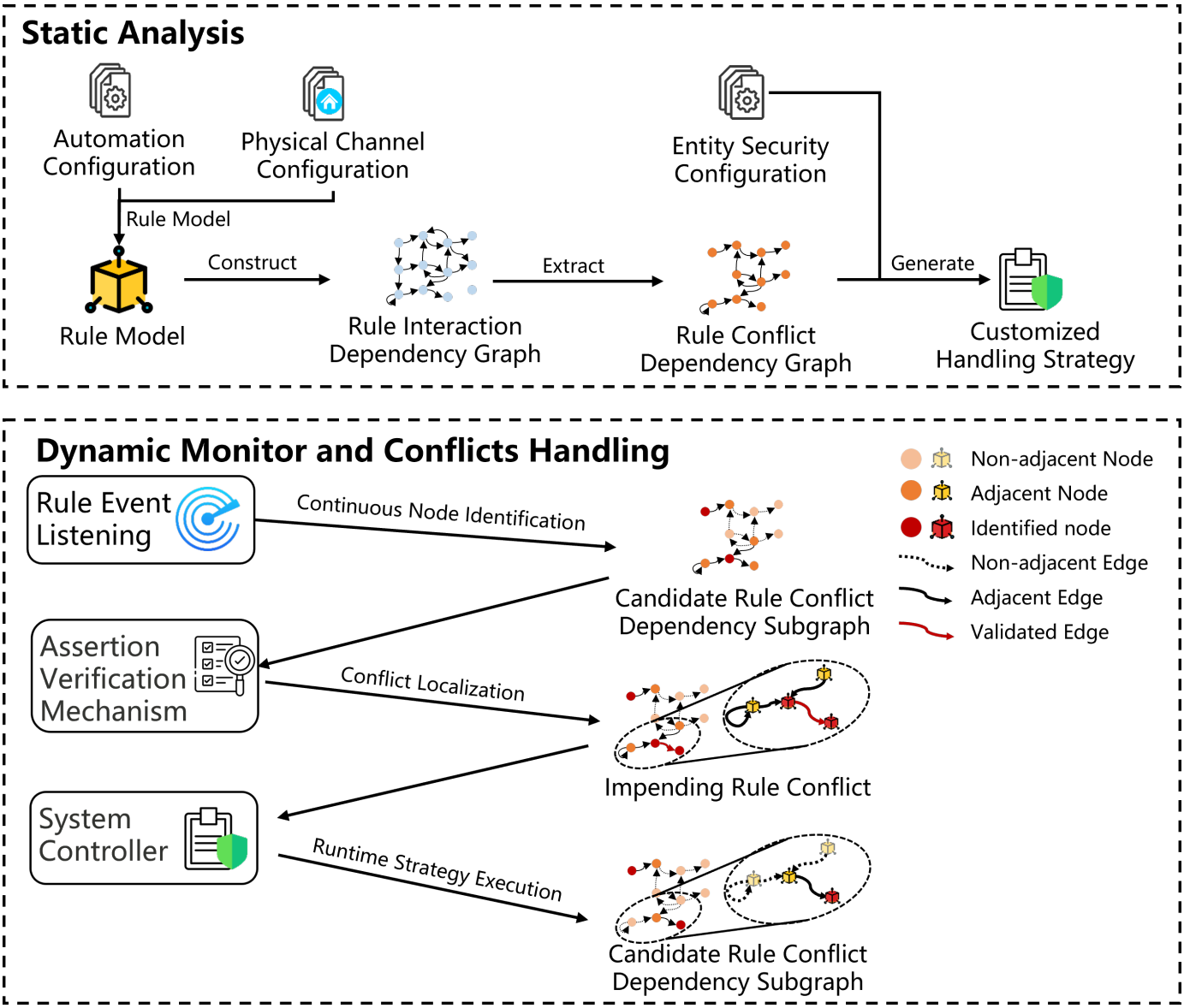


Fig. 3. Overall Architecture of Our Approach

two main phases. 1) Static Analysis Phase: The system first parses the automation configuration and physical channel configuration to construct the rule model. Subsequently, it builds the Rule Interaction Dependency Graph (RIDG) through formal verification. Next, by incorporating the entity security configuration, the system extracts potentially risky interactions from the RIDG, forming the Rule Conflict Dependency Graph (RCDG), and generates a customized handling strategy for each conflict node. 2) Dynamic Monitoring and Conflict Mitigation Phase: The system monitors rule events in real-time and maps them to the nodes of the RCDG. Utilizing the assertion verification mechanism, the system detects whether the currently triggered rule has activated a conflict edge in the dependency graph. Once the impending conflict is confirmed, the runtime controller immediately intercepts the default operation and executes the pre-defined conflict mitigation strategy.

3.2 Static Analysis

Before formally introducing the static phase, it is first necessary to clarify our approach to classifying rule interactions and rule conflicts. Based on previous core observations, we summarize the types of interactions between rules as shown in Figure 4: the execution result of one rule can have direct or indirect impacts on the trigger, condition, or action attributes of another rule. Accordingly, we classify rule interactions into the following categories: Trigger Interaction, Condition Interaction, Action Interaction, Indirect Trigger Interaction, Indirect Condition Interaction, and Indirect Action Interaction. Correspondingly, we define six types of rule conflicts: Trigger Conflict, Condition Conflict, Action Conflict, Indirect Trigger Conflict, Indirect Condition Conflict, and Indirect Action Conflict.

3.2.1 Rule Modeling

Smart home rules are typically stored in static files, from which the Trigger (T), Condition (C), and Action

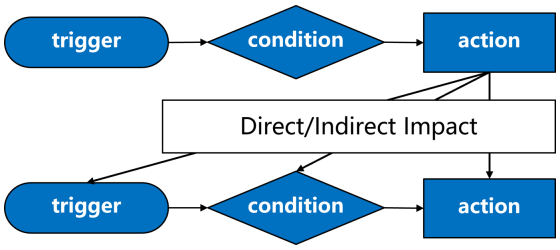


Fig. 4. Rule Interaction Pattern

(A) attributes of the rule can be extracted and modeled as the Trigger-Condition-Action (TCA) model. However, the standard TCA model fails to capture indirect interactions generated through the physical environment (i.e., "side channels"), such as interactions arising from the effect of different rules on the bedroom temperature. To address this issue, we introduce the Physical Channel Configuration and propose the enhanced TCAE model. We define the physical channel attribute $E = [e^1, e^2, \dots]$, where $e^i = (zone, channel, trend)$. The $zone$ represents the affected zone, such as the kitchen, living room, or the entire home; the $channel$ represents the physical quantity, such as temperature, humidity, or light intensity; and the $trend$ represents the direction of influence, such as increase or decrease. For example, if a rule turns on the bedroom heater, its action attribute E_A will contain $\langle bedroom, temperature, increase \rangle$. Through this modeling, the system can identify implicit associations across zones or those based on environmental changes. For the trigger or condition of a rule, if e is $\langle kitchen, temperature, increases \rangle$, it means the execution result of other rules may cause the kitchen temperature to rise, thereby making the current rule's trigger easier to fire or its condition easier to satisfy. For a rule's trigger or condition, if e is $\langle kitchen, temperature, increases \rangle$, it implies that the execution results of other rules causing the kitchen temperature to rise may facilitate the triggering of the current rule or satisfy its condition. For the action of a rule, if e is $\langle kitchen, temperature, increases \rangle$, it means the execution of the current rule may cause the kitchen temperature to rise. The attribute E is determined based on the user's real home setup and relevant rules. For example, if a home does not have any humidity-related rules, the $channel$ does not need to be set to "humidity." Thus, a rule can be re-modeled in the form of $R = \langle T, C, A, E \rangle$ or $R = \langle T, C, A, E_T, E_C, E_A \rangle$.

Through the above modeling method, we can clearly obtain the trigger, condition, action, and corresponding physical channel attributes for each automation rule. By setting translation templates for automation rules, and taking the two rules set in the greenhouse from the Motivating Example and Problem Analysis section as examples, the configuration content for R1 is `Rule:R1⟨ T={platform:input datetime, entity id:system time, event:sunset}, C={null}, A={platform:input boolean, entity id: heater, action:turn on}, TE={null}, CE={null}, AE={greenhouse-temperature-up} ⟩`. Based on the template, this can be translated into: "Rule R1: If sunset, turn on the heater, which will cause the greenhouse temperature to rise." The configuration content for R2 is: `Rule:R2 ⟨ T={platform: numeric state, entity id:temperature sensor, above:30},C={null},A={platform:input boolean, entity id: heater, action:turn off | platform:input boolean, entity id: skylight, action:turn on}, TE={greenhouse-temperature-up}, CE={null}, AE={greenhouse-temperature-up | greenhouse-temperature-down} ⟩`. Based on the template, this can be translated into: "Rule R2: If the temperature sensor is above 30 degrees Celsius, turn off the heater and turn on the skylight. The temperature may rise or fall."

3.2.2 Construction of Rule Interaction Dependency Graph

After completing rule modeling, we utilize the formal verification method to identify all possible rule interaction patterns and represent them in the form of a directed graph. We define the Rule Interaction Dependency Graph (RIDG) as $G_I = (V, E_I)$, where the vertex set V represents all automation rules, and the edge set E_I represents the rule interaction relationships.

Based on Figure 4, if a logical relationship defined in Table 1 exists between rule R_i and R_j (e.g., the action A_i of R_i triggers the trigger T_j of R_j), a directed edge $R_i \rightarrow R_j$ is established in the graph. The formal verification module iterates through all rule pairs, verifying whether they satisfy the six interaction types (Trigger Interaction, Condition Interaction, Action Interaction, and their corresponding indirect types). The resulting RIDG encompasses all logically and physically possible associations within the system, providing a complete search space for subsequent conflict detection.

The formal verification method is shown in Table 1. Assume rules $R_1 = \langle T_1, C_1, A_1, E_{T_1}, E_{C_1}, E_{A_1} \rangle$ and $R_2 = (T_2, C_2, A_2, E(T_2), E(C_2), E(A_2))$ represent two rules configured in the same system. T, C, A, and E denote the corresponding attributes. We use $\rightarrow$ to denote facilitating a trigger to fire or a condition to be met, and $\nrightarrow$ to denote prohibiting a condition from being met. For channel attributes, $\perp$ signifies that two channel attributes have the same $zone$ and $channel$ but opposite $trend$ (e.g., kitchen temperature increase versus kitchen temperature decrease). Additionally, $\perp$ also indicates that two actions conflict with each other (e.g., turning on the air conditioner versus turning off the air conditioner). "=" signifies that two channel attributes have the same $zone$, $channel$, and $trend$, or that two triggers or conditions are identical.

The definition of rule interaction patterns allows for the creation of translation templates for describing rule interactions. Continuing with the example of the two rules set in the greenhouse from the Motivating Example and Problem Analysis section, we find that the execution action of automation rule R1 may cause automation rule R2 to be triggered, which conforms to the Trigger Interaction pattern. Thus, a template translation can be set as: "The execution action of automation rule R1 raises the temperature, leading to the triggering of automation rule R2."

TABLE 1
Formal Analysis Expression

Classification	Expression
Trigger Interaction	$(\exists a_1 \in A_1) \wedge (a_1 \rightarrow T_2)$
Condition Interaction	$((A_1 \nrightarrow C_2) \wedge ((T_1 \neq T_2) \wedge (R_1 \neq R_2))) \vee ((A_1 \rightarrow C_2) \wedge ((T_1 \neq T_2) \wedge (R_1 \neq R_2)))$
Action Interaction	$(\exists a_1 \in A_1) \wedge (\exists a_2 \in A_2) \wedge (a_1 \perp a_2)$
Indirect Trigger Interaction	$(\exists a_1 \in A_1) \wedge \left(E_{a_1} = E_{T_1}\right)$
Indirect Condition Interaction	$\left(\left(E_{A_1} \perp E_{C_2}\right) \vee \left(E_{A_1} = E_{C_2}\right)\right) \wedge (R_1 \neq R_2)$
Indirect Action Interaction	$(\exists a_1 \in A_1) \wedge (\exists a_2 \in A_2) \wedge (D_{a_1} \neq D_{a_2}) \wedge (E_{a_1} \perp E_{a_2})$

3.2.3 Extraction of Rule Conflict Dependency Graph

Not all rule interactions are harmful. To reduce false positives and identify genuine rule conflicts, we introduce the Entity Safety Configuration to filter the RIDG. We define the Rule Conflict Dependency Graph (RCDG) as a subgraph of the RIDG, $G_C = (V_C, E_C)$, where $E_C \subseteq E_I$ only contains interaction edges that violate the safety configuration. The entity safety configuration is a set of safe and dangerous states for safety-sensitive device entities, used to represent the states that should be prioritized or avoided when a conflict occurs in the smart home system. The entity safety configuration will be automatically configured by combining existing safety property research data in the IoT domain. For example, a door should remain "closed," and a fire sprinkler head should remain "enabled." Additionally, users can define custom configurations; for instance, for lights with flashing and color-changing capabilities, the system automatically reads the available states, and a user with photosensitive epilepsy can define the "flashing and color-changing" state as a threat.

We employ a heuristic algorithm based on safety value (sf) to construct the RCDG. For every edge in the RIDG (i.e., the interaction relationship of the rule pair R_1, R_2), the system calculates the degree of deviation of the rule execution result from the entity safe state. If the interaction leads to a result that violates the preferred safe state (e.g., $sf < 0$) or tends toward a dangerous state, the interaction is marked as a candidate rule conflict and retained in the RCDG; otherwise, the edge is pruned. This graph-based extraction method effectively narrows the focus from all interactions to only those interaction paths with potential risk. We employ a heuristic algorithm based on a "safety score" (sf) to construct the RCDG. The sf value is calculated using a linear accumulation function based on the Entity Safety Configuration. For example, if the execution result of the rule is to close the door, which conforms to the safe state "closed" of the entity "door," the sf value for this rule is incremented by one from its original value; conversely, it is decremented. The default value is zero.

3.2.4 Generation of Customized Handling Strategy

For every conflict edge in the RCDG, the system automatically generates a customized handling strategy. Unlike generic "blocking" policies, our method intelligently decides whether to cancel a rule, execute only the rule with higher priority, or execute them in a specific order, based on the logic in Table 2 and the magnitude of the sf value. The generated strategies are passed to the dynamic monitoring module along with the RCDG.

The determination of a rule conflict differs for various types of rule interaction:

- For Trigger Interaction and Indirect Trigger Interaction, the focus is on whether the safety value (sf) of the second rule is greater than zero. If it is less than zero, it is determined to be a rule conflict.
- For Condition Interaction and Indirect Condition Interaction, if the first rule prohibits the condition of the second rule, and the sf of the second rule is greater than zero, it is determined to be a rule conflict; conversely, if the first rule enables the condition of the second rule to be met, and the sf of the second rule is less than zero, it is also determined to be a rule conflict. This indicates that the influence of the former rule either prevents a rule that conforms more closely to the entity safety state from being executed or causes a rule that contradicts the entity safety state to be executed.
- For Action Interaction and Indirect Action Interaction, if the safety value of one of the two involved rules is not zero, it is determined to be a rule conflict. It should be noted that even if the safety values of both rules are greater than zero, it may still be determined to be a rule conflict, as the execution results of the two rules in this type of interaction are mutually contradictory, and typically only one of them should be executed.

Thus, multiple handling strategies can be set for each type of rule conflict. Specific conflict handling strategies can be selected according to Table 2.

In addition, the system will use translation templates to convert all rule interactions, candidate rule conflicts, and conflict handling strategies into natural language that is easy for the user to understand. Users can then adjust the rule interactions and candidate rule conflicts based on personal preferences.

3.3 Dynamic Monitoring and Conflict Mitigation

3.3.1 Rule Event Listening and Graph Mapping

The core idea of dynamic monitoring is to focus on rule events in real-time, thereby tracking the state of nodes in the RCDG. The Rule Event Listening module continuously captures trigger events in the system. When rule R_i is triggered, the system does not view it in isolation but identifies it as an Identified Node in the RCDG. The system immediately checks whether this node has outgoing edges in the graph, i.e., whether adjacent nodes exist. If no adjacent

TABLE 2
Handling Strategy Decision

Classification	Decision	Options
Trigger Conflict Indirect Trigger Conflict	$sf_2 \geq 0$	Not rule conflict
	$sf_1 \geq 0 \wedge sf_2 < 0$	Only execute R_1
	$sf_1 < 0 \wedge sf_2 < 0$	Neither rule will be executed
Condition Conflict Indirect Condition Conflict	$(A_1 \rightarrow C_2 \wedge sf_2 \geq 0) \vee (A_1 \nrightarrow C_2 \wedge sf_2 \leq 0)$	Not rule conflict
	$A_1 \rightarrow C_2 \wedge sf_2 < 0$	Do not execute R_2
	$(A_1 \nrightarrow C_2) \wedge (sf_2 > 0) \wedge (sf_1 < 0)$	Execute R_2
Action Conflict Indirect Action Conflict	$sf_1 \geq 0 \wedge sf_2 \geq 0$	Not rule conflict
	$sf_1 \geq 0 \wedge sf_2 < 0$	Only execute R_1
	$sf_1 < 0 \wedge sf_2 \geq 0$	Only execute R_2
	$sf_1 < 0 \wedge sf_2 < 0$	Neither rule will be executed

nodes exist, the rule is currently safe; if they do exist, it indicates a potential cascade conflict risk, requiring assertion verification on the edge connecting the two nodes.

A Node Forgetting Mechanism is also designed: if the execution action of a rule is interrupted (e.g., a rule controls the air conditioner to turn on), the rule, after being successfully triggered and executed, is considered an Identified Automation Node. Subsequently, if a user manually or another rule's execution action changes the state of the device controlled by this rule (e.g., turns off the air conditioner), the automation rule node will be forgotten.

Through dynamic identification of nodes and forgetting nodes, we achieve dynamic tracking of a long-running smart home system, maintaining focus on currently running rules and those about to be executed.

3.3.2 Assertion Verification Mechanism

Even if an edge $R_i \rightarrow R_j$ exists in the RCDG, a conflict does not necessarily occur at the current moment (e.g., if the two rules fire in isolation rather than through a rule interaction). Therefore, we introduce the Assertion Verification Mechanism for runtime confirmation.

When node R_i in the RCDG is monitored, if its adjacent node R_j is identified but not yet executed, the system performs an assertion check on the edge between the two nodes. The assertion logic is shown in Table 3. The assertion verification primarily involves the following functions:

- $obs()$: Observing the occurrence of an event (including the triggering of the trigger $obs(T)$,the passing of a condition check $obs(C)$, the failing of a condition check $obs(\neg C)$ and the execution of an action $obs(A)$
- $intime(X, Y, \delta)$: Returns true if the time interval between events X and Y is less than δ (default $\delta = 0.1s$).

Thus, corresponding assertion verification methods exist for different types of rule conflicts, as shown in Table 3. Only when all assertion conditions are met does the system determine that the candidate rule conflict has transformed into an Impending Rule Conflict.

For example, suppose two rules R_1 and R_2 are classified as a Trigger Conflict type of rule conflict during static analysis. The assertion verification process is as follows: (1) Step one: observe the occurrence of T_1, C_1, and A_1, indicating that rule R_1 has been triggered, passed the condition check, and successfully executed the expected action; (2) Step two: observe the occurrence of T_2 and C_2, indicating that rule R_2 has been triggered and passed the condition check, and the platform is about to execute the default action of R_2; (3) Step three: observe that the time interval between the occurrence of events A_1 and T_2 is very short, less than the threshold δ. If all three steps are true, it can be concluded that the execution of rule R_1 led to the triggering and impending execution of rule R_2. To prevent coincidence in real-world scenarios (i.e., independent execution of R_1 and R_2 close in time), δ is set to a minimal threshold (e.g., 0.1s or 10ms). The system's low runtime overhead (millisecond level) ensures that valid conflicts are not missed due to processing delays. There is no need to worry that system processing time overhead will cause the threshold δ to be too small, leading to the failure to detect a rule conflict, as the time overhead in the system is extremely low, typically at the millisecond level.

TABLE 3
Assertion Verification Expression

Classification		Expression
Trigger Conflict		$obs(T_1), obs(C_1), obs(A_1)$ $obs(T_2), obs(C_2)$ $intime(A_1, T_2, \delta)$
Condition Conflict	Make Conditions Forbidden	$obs(C_2)$ $obs(T_1), obs(C_1), obs(A_1)$ $obs(\neg C_2)$ $intime(A_1, \neg C_2, \delta)$
	Make Conditions Satisfied	$obs(\neg C_2)$ $obs(T_1), obs(C_1), obs(A_1)$ $obs(C_2)$ $intime(A_1, C_2, \delta)$
Action Conflict		$obs(T_1), obs(C_1), obs(A_1)$ $obs(T_2), obs(C_2)$
Indirect Trigger Conflict		$obs(T_1), obs(C_1), obs(A_1)$ $obs(T_2), obs(C_2)$
Indirect Condition Conflict	Make Conditions Forbidden	$obs(C_2)$ $obs(T_1), obs(C_1), obs(A_1)$ $obs(\neg C_2)$ $intime(A_1, \neg C_2, \delta)$
	Make Conditions Satisfied	$obs(\neg C_2)$ $obs(T_1), obs(C_1), obs(A_1)$ $obs(C_2)$ $intime(A_1, C_2, \delta)$
Indirect Action Conflict		$obs(T_1), obs(C_1), obs(A_1)$ $obs(T_2), obs(C_2)$

3.3.3 Runtime Controller and Strategy Execution

The Runtime Controller is responsible for assisting state reading during normal operations. It monitors rule trigger events and rule condition check events, and sends these events to the Rule Event Listening module to achieve node identification. It monitors the status of devices controlled

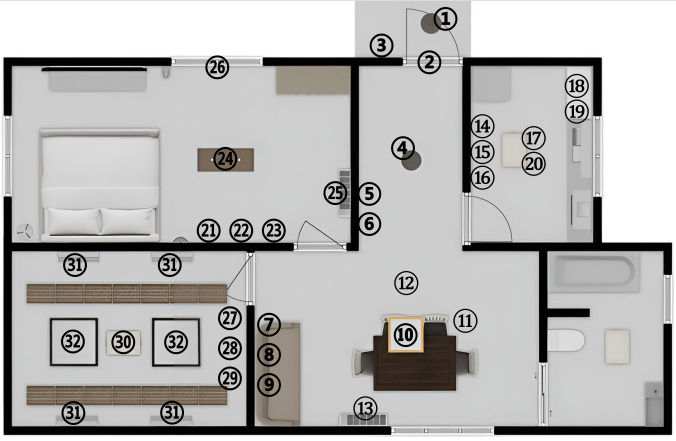
by the identified nodes and sends an event to the event listening module when the device status changes, thus enabling the forgetting of the identified nodes. It is also responsible for obtaining the system state, including device entity statuses, system time, and event occurrence times, to assist with assertion verification.

In addition, the core function of the Runtime Controller is to execute conflict mitigation. Once assertion verification confirms an Impending Rule Conflict, the Runtime Controller takes over device control and performs conflict mitigation. According to the customized handling strategy generated in the static analysis phase, the controller will intercept the default rule execution flow and execute the customized handling strategy (e.g., preventing the execution of R_j or enforcing the execution of R_i), thereby eliminating security risks before the rule truly executes.

4 Evaluation

In this section, we primarily evaluate the system from the following perspectives: 1) the effectiveness of rule conflict detection; 2) the effectiveness of automated rule conflict handling; and 3) the overall system performance.

4.1 Smart Home Testbeds



Serial number	Device	Entity security status
③	Smart door lock	Close
⑱	Water valve	Open
⑳	Sprinkler	Open
㉖	Window	Close

Device				
①Light	②Brightness sensor	③Smart door lock	④Light	⑤Motion sensor
⑥Brightness sensor	⑦Motion sensor	⑧Brightness sensor	⑨Humidity and temperature sensor	
⑩Light	⑪Humidifier	⑫Underfloor heating	⑬Air conditioner	⑭Motion sensor
⑮Brightness sensor	⑯Water leak detection sensor		⑰Light	⑱Water valve
⑲Smoke detector	⑳Sprinkler	㉑Motion sensor	㉒Brightness sensor	
㉓Temperature sensor	㉔Light	㉕Heater	㉖Window	㉗Motion sensor
㉘Brightness sensor	㉙Temperature sensor	㉚Light	㉛Heater	㉜Window

Fig. 5. Smart Home Floor Plan

We implemented a complete system and conducted virtual testing. For the virtual testing, we selected the Python-based open-source smart home platform, Home Assistant. Home Assistant supports devices from various manufacturers and provides virtual devices to enable automation functions. On this platform, users can define custom automation rules via a visual Web interface or by directly editing configuration files. Home Assistant uses YAML configuration files to describe automation rules, which include the triggers, conditions, and actions of the rules.

The virtual test employs the smart home floor plan shown in Figure 5. The simulated smart home environment consists of seven different Zones: Outdoor, Porch, Living Room, Kitchen, Bathroom, Bedroom, and Greenhouse. Relevant smart home devices are deployed in each Zone, with relevant smart home devices deployed in each zone, totaling 32 devices. Depending on the user's smart home device configuration, the system's environmental perception capabilities vary. For instance, if the Bedroom is equipped with brightness and temperature sensors, the system can capture changes in brightness and temperature but cannot detect humidity changes. Since the Living Room is equipped with a humidity sensor, it can capture changes in the Living Room's humidity in addition to brightness and temperature. Table 4 details the Physical Channel Configuration within the smart home system in the test environment. Each physical channel includes three parameters: Zone, Physical Quantity, and Trend. In this home layout, certain channels in specific Zones influence each other; for example, brightness and temperature attributes in the Porch and Living Room are interdependent, whereas those in the Living Room and Bedroom are independent.

TABLE 4
Classification of Side Channel Attributes Across Different Home Zones

Zone	Channel	Trend
outdoor	brightness	increase/decrease
porch	brightness	increase/decrease
kitchen	brightness,temperature	increase/decrease
living room	brightness,humidity,temperature	increase/decrease
bedroom	brightness,temperature	increase/decrease
greenhouse	brightness,temperature	increase/decrease

Table 5 lists the 17 automation rules configured across these seven Zones, along with the corresponding Zones and devices. By analyzing the devices controlled by the actions of all rules and their state options, the system provides selection templates and explanations: "The following are all devices controlled by the rules. Please select the options for safety-sensitive devices that make you feel more secure, to avoid unintended state changes caused by rule interactions. For example, choose configurations such as keeping doors and windows always closed, cameras always on, and setting default parameters for air conditioners and fans." The system provides default recommendations for door locks, smoke detectors, and windows. Here, we assume the physical security configuration in the smart home system is as shown in Figure 5, with special attention to the state settings of four safety-sensitive devices: the Water Valve ⑱ is preferred to be Open, the Sprinkler ⑳ is preferred to be Enabled, the Bedroom Window ㉖ is preferred to be Closed, and the Smart Door Lock ③ is preferred to be Locked.

4.2 Effectiveness of Conflict Detection

To evaluate the effectiveness of rule conflict detection, we first identified all rule interactions and rule conflicts through

TABLE 5
Testbed: Details of Experimental Smart Home Rules (R1-R17)

Rule ID	Content	Zone	Devices
R1	When the door lock is opened, if the brightness sensor is below 50lux, turn on the outdoor light and porch light.	Outdoor	① ② ③ ④
R2	When the greenhouse light is turned off, unlock the door.	Porch	③
R3	When motion is detected in the porch, if the brightness sensor is below 50lux, turn on the porch light.	Porch	④ ⑤ ⑥
R4	When motion is detected in the living room, if the brightness sensor is below 50lux, turn on the living room light.	Living Room	⑦ ⑧ ⑩
R5	When the living room humidity is below 40%, and the humidifier is off, turn on the living room humidifier.	Living Room	⑨ ⑪
R6	When the living room humidity is above 60%, turn off the living room humidifier.	Living Room	⑨ ⑪
R7	When the living room temperature is below 24°C, turn off the living room floor heating and turn on the underfloor heating.	Living Room	⑨ ⑫
R8	When the living room temperature is above 30°C, turn off the living room air conditioner and set it to 27°C.	Living Room	⑨ ⑬
R9	When motion is detected in the kitchen, if the brightness sensor is below 50lux, turn on the kitchen light.	Kitchen	⑭ ⑮ ⑰
R10	When a water leak is detected, close the main water valve.	Kitchen	⑯ ⑱
R11	When the smoke detector is triggered, turn on the sprinkler.	Kitchen	⑲ ⑳
R12	When motion is detected in the bedroom, if the brightness sensor is below 50lux, turn on the bedroom light.	Bedroom	㉑ ㉒ ㉔
R13	At 7 PM, turn on the bedroom heating.	Bedroom	㉕
R14	When the bedroom temperature reaches 30°C, open the window and close the heating.	Bedroom	㉓ ㉕ ㉖
R15	When the door lock is locked, turn off the porch light and greenhouse light.	Outdoor	③ ④ ㉜
R16	At 7 PM, turn on the greenhouse heating.	Greenhouse	㉛
R17	When the greenhouse temperature reaches 32°C, open the window and close the heating.	Greenhouse	㉙ ㉛ ㉜

manual inspection, and then proceeded to test and evaluate the system. The system first automatically models the automation rules and performs rule interaction detection to construct the Rule Interaction Dependency Graph. Then, combining the entity security configuration generated via the interaction interface, it generates the Rule Conflict Dependency Graph (comprising candidate rule conflicts) and generates corresponding customized conflict resolution strategies for each candidate rule conflict. We then manually triggered rule conflicts to assess the system's detection and mitigation capabilities, while using a similar method to compare the detection results with IoTMEDIATOR.

In the static analysis phase, the system determined the home's Physical Channel Configuration and entity security attributes through user interaction. The directed graph composed of rule interactions across all seven Zones is shown in Figure 6. Our system detected a total of 72 pairs of rule interactions, including 2 pairs of Rule Trigger Interaction pattern, 2 pairs of Rule Condition Interaction pattern, 10 pairs of Rule Action Interaction pattern, 7 pairs of Indirect Rule Trigger Interaction pattern, 15 pairs of Indirect Rule Condition Interaction pattern, and 22 pairs of Indirect Rule Action Interaction pattern. Multiple rule interactions may exist between two rules. In the Rule Interaction Dependency Graph, directed edges indicate that one automation rule exerts an influence on the trigger, condition, or action of another automation rule; thus, a single edge may encapsulate multiple interaction relationships. Furthermore, we use red directed edges to mark interaction relationships where one automation rule constitutes a candidate rule conflict with another (Rule Conflict Relationship $\in$ Candidate Rule Conflicts $\in$ Rule Interactions).

The rule interaction list clearly displays the interactions between rules. Here is an example of a normal interaction. The action of automation rule R5 is to turn on the Living Room humidifier, which increases the humidity in the Living Room, making the trigger condition of rule R6 ("humidity above 60%") easier to satisfy. This also transitions rule R6's condition from an unsatisfied state to a satisfied state. Both rule R5 and rule R6 operate the "humidifier," so their actions technically "conflict." However, since their execution results do not have negative impacts on safety-sensitive devices, they are not considered rule conflicts. Furthermore, during the subsequent dynamic monitoring in manual trigger tests, the actions of these two rules do not occur simultaneously to cause indeterminate states, as the trigger times of the two rules do not coincide in real-world scenarios.

Here is another example of a rule conflict: Rule R10 and Rule R11. Rule R10 is designed to close the water valve promptly upon detecting a water leak to avoid continuous damage. Rule R11 is designed to open the fire sprinkler if the smoke detector detects smoke, to prevent a fire. The action of Rule R10 may cause the water valve to close, rendering the system unable to extinguish a fire when smoke is detected. The system detects that the interaction between these two rules may have a negative impact on the safety-sensitive device "fire sprinkler," leading to safety issues; thus, the interaction of Rule R10 on Rule R11 is listed as a candidate rule conflict. Conversely, the execution of Rule R11 implies a fire detection. At this time, the normal operation of the fire sprinkler may trigger the water leak sensor, thereby triggering Rule R10, which attempts to close the main water valve. Therefore, the interaction of Rule R11 on Rule R10 is also listed as a candidate rule conflict.

Since the goal of this system is to detect and mitigate rule conflicts hidden within rule interactions, it does not inspect individual rules for malicious intent. However, it can detect whether malicious rules negatively impact sensitive devices during rule interactions. Rule R2 and Rule R15 serve as an example of malicious rule injection. Assume R2 and R15 are malicious rules injected by an attacker without the user's knowledge (for instance, contained within a bundle of automation rules from a marketplace and applied by the user with a one-click installation). When the door lock is locked, the porch light and greenhouse light are turned off. However, the turning off of the greenhouse light serves as the trigger for Rule R2. Once R2 is triggered, it unlocks the door, providing an opportunity for the attacker.

There is also a case where results differ due to different Zones. The interaction patterns between Rule R13 and Rule R14, and between Rule R16 and Rule R17, are identical. However, the former occurs in the Bedroom involving the Bedroom Window and Heating, while the latter occurs in the Greenhouse involving the Greenhouse Skylight and Heating. Both interactions were successfully detected. Yet,

because the Bedroom Window is classified as a safety-sensitive device that should not be triggered without user knowledge, it is necessary to prevent rule interactions from shifting the device state to a dangerous one. Consequently, the former is considered a rule conflict, whereas the latter is deemed safe.

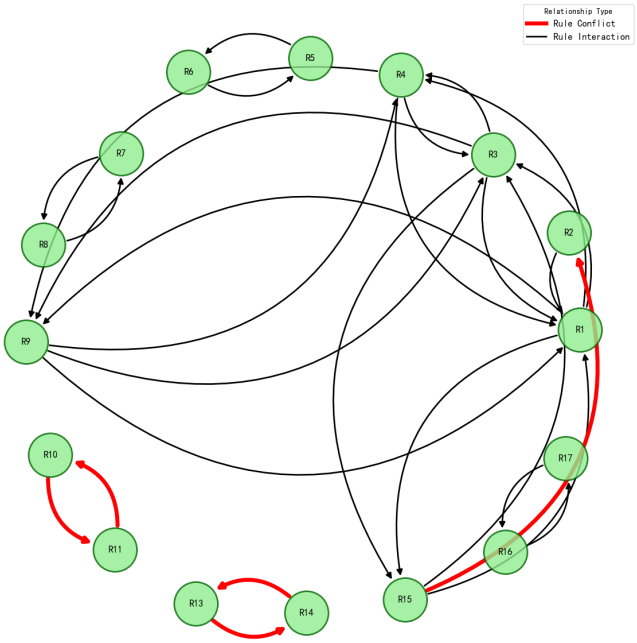


Fig. 6. Rule Interaction Dependency Graph

Next, we manually triggered all rule interactions based on the intended purpose of the rules. The conflict test results for 13 rules across all seven Zones are presented in Table 6.Specifically, the interaction $R10 \rightarrow R11$ constitutes a rule conflict: R10 closes the water valve, rendering R11's action (turning on the sprinkler) ineffective for fire suppression. R11-R10 constitutes a rule conflict: When R11 executes, it triggers R10, and their actions are mutually exclusive; furthermore, R10's execution closes the main water valve, impacting R11's action. R13-R14 constitutes a rule conflict: The execution of R13 triggers R14, their actions are mutually exclusive, and R14's action opens the window, potentially creating an unexpected unsafe state for the user. Additionally, the rule conflict between Rule R15 and Rule R2 was correctly detected: While R15's trigger appears to turn off lights to conserve resources, it actually serves to trigger the malicious Rule R2 to unlock the door.

The test results indicate that IoTMEDIATOR failed to detect the R11-R10 rule conflict because it cannot capture the influence of environmental factors. However, based on rule pattern matching, it identified the rule conflicts in R13-R14 and R16-R17. Meanwhile, IoTMEDIATOR detected many other rule interactions, but determining whether they constitute rule conflicts requires users to select based on subjective preferences. For example, regarding the actions of R5 and R6, IoTMediator identified that Rule R5 turns on the Living Room humidifier to increase humidity, enabling Rule R6's condition, while Rule R6 turns off the humidifier, enabling Rule R5's condition. Additionally, there are special cases. The malicious Rule R2 accidentally interacted with Rule R1, but the impact was limited to the action of the malicious Rule R2 itself; their interaction did not produce a malicious combined effect. Similarly, for Rule R3 and Rule R15, their interaction does not bring malicious impacts but fits the "Potential Race Condition" pattern, causing it to be incorrectly reported to the user, thereby generating false positives.

In summary, our approach precisely captures all rule interactions and effectively distinguishes rule conflicts by meticulously classifying rule interaction patterns and assessing whether the endpoint of the interaction leads to malicious behavior. It detects all possible conflict mechanisms, whether two rules impact the same physical device or the same physical environment to produce a negative effect on the smart home; whether one rule's action directly or indirectly triggers another rule; or whether one rule's action directly or indirectly invalidates a satisfied condition or enables an unsatisfied one. Furthermore, it effectively minimizes false positives, largely preventing normal rule interactions from being misclassified as conflicts. Simultaneously, by employing a simple and intuitive interaction method, it assists users in deciding whether to classify other rule interactions as candidate rule conflicts based on personal preferences.

4.3 Effectiveness of Conflict Mitigation

Our approach selects a designated conflict resolution strategy for each rule conflict based on the rule interaction pattern, utilizing the entity security configuration generated via user interaction. Users can also review and modify the resolution strategies. During system runtime, if a rule conflict is detected to be imminent, our solution intercepts it before it occurs and executes the corresponding conflict resolution strategy, thereby mitigating the rule conflict. In this section, we evaluate the effectiveness of the conflict resolution strategy selection.

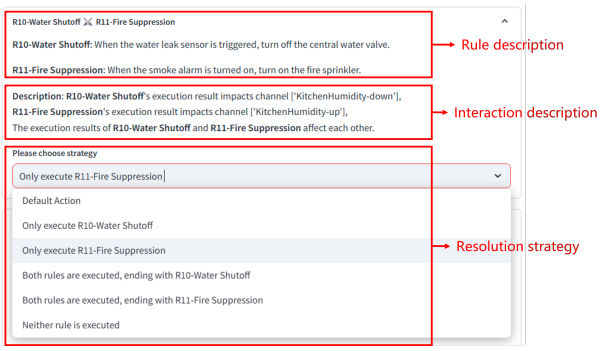


Fig. 7. Example of Automated Rule Conflict Resolution

To evaluate the usability of automated rule conflict handling, this section presents specific examples to demonstrate how the system automatically handles a specific rule conflict, and compares it with the effective handling solutions generated by an LLM based on the rule conflict information provided by the system.

Figure 7 illustrates the Indirect Rule Action Conflict corresponding to Rule R10 and Rule R11. The conflict scenario

TABLE 6
Result of Rule Conflict Detection

	Classification	R2-R1	R3-R15	R5-R6	R6-R5	R10-R11	R11-R10	R13-R14	R14-R13	R15-R2	R15-R3	R16-R17	R17-R16
Ours	Rule Trigger Conflict									✓			
	Rule Condition Conflict												
	Rule Action Conflict							✓	✓				
	Indirect Rule Trigger Conflict						✓	✓					
	Indirect Rule Condition Conflict												
	Indirect Rule Action Conflict					✓	✓	✓	✓				
IoTMediator	Condition Enabling/Disabling			×	×								
	Race Condition												
	Potential Race Condition		×	×	×			✓	✓		×	×	×
	Chained Execution	×*								✓			
	Action Revert												
	Infinite Loop												
	Condition Bypass												

is as follows: Rule R10 executes first (detects water leak, closes main water valve), and then Rule R11 is triggered (smoke alarm activated, fire sprinkler enabled). At this point, R10's action renders R11's action ineffective. The figure displays the rule descriptions: "R10-Water Shutoff's execution result impacts channel ['KitchenHumidity-down'], R11-Fire Suppression's execution result impacts channel ['KitchenHumidity-up'], The execution results of R10-Water Shutoff and R11-Fire Suppression affect each other." Below this, multiple conflict resolution strategy options are displayed, including executing default actions (not considered a conflict), executing only R10's action, executing only R11's action, executing R10's then R11's actions sequentially, executing R11's then R10's actions sequentially, or executing neither.

According to the entity security configuration, Rule R10's action closes the water valve; using a linear evaluation function, its corresponding safety value parameter is $sf_{R10} = -1$. Rule R11's action enables the fire sprinkler; its corresponding safety value parameter is $sf_{R11} = 1$. Consequently, the Generation of Resolution Strategy Module automatically recommends the "Only execute R11" strategy to prevent the actual occurrence of the rule conflict. Additionally, the interface provides options for handling strategies, allowing users to select the conflict resolution strategy that best fits their preferences.

To assess the expressiveness of the rule conflict descriptions generated by our Rule Modeling module and to explore the potential for AI-assisted decision-making, we conducted an experiment using a Large Language Model (LLM). Still taking Figure 7 as an example, we utilized the `gpt-4o-2024-11-20` model via the OpenAI API with default parameters. The system prompt is presented in Appendix A. The response obtained for the aforementioned example is as follows:

```
{ "policy": "2", // Only execute R11
"reason": "In this case of Indirect Rule
Action Conflict, the sprinkler system
activated by R11 is crucial for safety
during a fire, and its priority should
supersede the water cutoff triggered by
R10 to ensure effective fire suppression.
Therefore, only R11 should execute." }
```

The alignment between the LLM's recommendation and our system's deterministic calculation validates two key aspects of our framework:

- **Information Completeness:** The natural language descriptions generated by our translation templates successfully capture the logical and physical context required for conflict resolution. The LLM was able to deduce the correct strategy solely from the text we generated, proving that our model preserves critical dependency information.
- **Explainability:** The rationale provided by the LLM (as shown in the `reason` field) serves as an excellent explanation for ordinary users. By bridging the gap between complex rule logic and user understanding, this approach demonstrates that our framework can effectively function as a transparent "Copilot" for smart home management, rather than a black-box controller.

Through AI's understanding of rule conflict descriptions, the recommended results for rule conflict resolution strategies align with expectations, which to some extent indicates that the currently extracted rule conflict information can be well understood by users, enabling them to make correct and effective choices, and providing them with the option to use intelligent tools for recommending rule conflict resolution strategies.

4.4 Performance

Performance plays a crucial role in the detection and handling of automation rule conflicts. Poor performance (e.g., high resource consumption hindering deployment, or slow execution preventing effective real-time operation or blocking normal system functions) renders a system impractical. To test system performance, we divided the evaluation into static analysis performance and dynamic monitoring and conflict handling performance, based on the system architecture.

In the static detection performance evaluation, the process iterates through all automation rules for modeling and performs pairwise and self-interaction analyses for each rule (determining whether two rules fit a specific rule interaction pattern). Combined with relevant configurations, it generates the Rule Interaction Dependency Graph and the Rule Conflict Dependency Graph. Consequently, for n rules, the time complexity of static detection is $O(n^2)$. We used AI-generated sets of 100 and 1000 rules as input to evaluate the static detection time in seconds. For clarity, we recorded the average of three detection runs. Figure 8 displays the results. It can be observed that even with 1000 rules, static detection requires only about 12 seconds. The

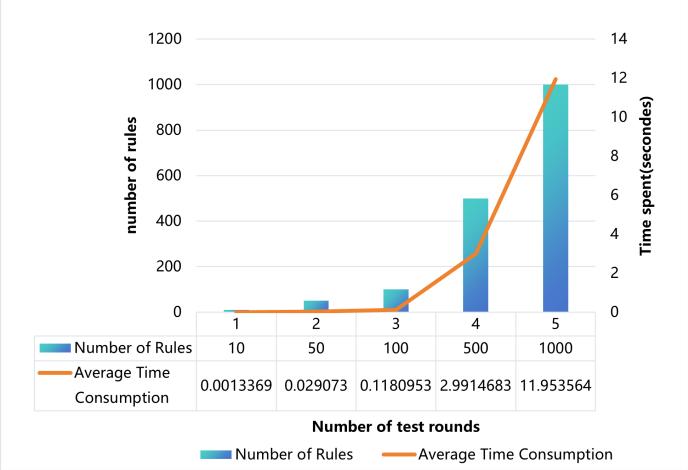
TABLE 7
Performance of Communication Functions

Function Name	Description	Avg. Latency	Max. Latency
get_entity_state	Get entity state	1.193×10^{-3}s	1.953×10^{-3}s
time_now	Get HomeAssistant system time	9.765×10^{-4}s	9.825×10^{-4}s
command_send	Send conflict handling command	9.643×10^{-4}s	9.787×10^{-4}s

Note: Each function is tested 10 times.

processing latency scales with data volume consistent with the expected complexity. It is worth noting that the number of automation rules in a typical smart home is far fewer than 100; thus, ordinary users will perceive no delay. In rare cases where the rule count exceeds 500, although static detection may take several seconds, it operates in an offline mode. It only needs to run once after rule updates and does not block the dynamic operation of the smart home system.

Fig. 8. Performance of Static Analysis



The dynamic monitoring and rule conflict handling operations are closely intertwined; significant latency would render the solution impractical. Therefore, we evaluated these two parts together. The computational load for rule conflict dynamic monitoring and executing conflict handling strategies is low; latency is primarily attributed to communication and execution functions. Consequently, our testing focused on evaluating the performance of key functions in the runtime controller. The descriptions and performance results of these functions are presented in Table 7. The `get_entity_state` function retrieves entity states, and `time_now` retrieves the system time; both assist in assertion verification. The `command_send` function is used to retrieve and execute specific commands within the conflict handling strategies.

The average and maximum execution latencies of functions in the experimental environment are at the millisecond level. Furthermore, the assertion verification and rule conflict handling command generation steps involve only a limited number of communication function calls, keeping the total time consumption at the millisecond level.

5 Related Work

In this section, we review existing work related to smart home automation rules and conflicts. As shown in Table 8, prior research has extensively investigated rule conflicts arising from the execution of automation functions (rule execution) in smart home systems. However, few studies have clearly distinguished between rule interaction and rule conflicts. A survey by Huang et al. [28] further points out the lack of detailed discussion on the unique characteristics of rule conflict detection—including problem definition, datasets, detection methods, and conflict types—remains limited, lacking a unified definition.

Several approaches focus on defining security policies or attributes to detect conflicts. Soteria [23], for instance, defines general attributes related to rule interaction patterns and application-specific attributes for defining security policies. Abnormal behavior is detected when these attributes are violated. Similarly, IoTGuard [15] utilizes both rule interaction patterns and specific security policies, defining 30 application-specific policies, 2 platform-specific trigger execution policies, and 4 general policies for detection. IoTSan [29] and IoTSafe [4] also follow a policy-based approach. IoTSan provides 45 predefined security attributes that users can select from. When a new smart application is installed, it checks for potential conflicts by verifying if the selected security attributes hold within the application's configuration. IoTSafe defines a syntax for security policies to enable the detection and prevention of rule conflicts, supporting enforcement or user notification before conflicts occur.

Another line of work focuses on detecting rule conflicts by identifying problematic rule interaction patterns. IoTCom [24], HomeGuard [**?**], and IoTMediator [22] fall into this category. IoTCom defines seven types of coordination threats and employs static analysis to detect potential rule interactions, including those related to side channels. HomeGuard and its subsequent work IoTMediator identify and enumerate rule interaction threats based on empirical observation. They combine static and dynamic detection methods to find these interaction threats and offer customized mitigation strategies for each type.

Some current studies discuss physical channels, where an automation rule may interact with another by influencing a physical channel. Works such as iRuler [27], IoTIE [2], IoTSafe [4], and IoTCom [24] support the detection of physical channels. However, only iRuler and IoTSafe support Zone awareness, which helps avoid false positives like the following scenario: one rule turns on a heater (increasing temperature in the bedroom), while another rule opens a window if the temperature is high (but the window is in the

TABLE 8
Comparison of Different IoT Conflict Handling Approaches

Name	DIC	Physical Channel	Zone Awareness	Detection Method	Auto Handling	Customized Handling	Timing of Handling	User Preference	User Effort	Manual Control	Generalization
Soteria [23]	T	F	F	Static	F	F	F	F	High	F	Low
IoTGuard [15]	T	F	F	Dynamic	T	F	Pre-event	F	Med	F	Low
SafeChain [26]	T	F	F	Static	F	F	F	F	Med	F	Med
iRuler [27]	T	T*	T*	Static	F	F	F	F	Low	F	Low
IoTIE [2]	T	T	T*	Static	F	F	F	F	Low	F	Med
IoTSafe [4]	T	T	T	Hybrid	T	F	Pre-event	F	Very High	F	High
IoTCom [24]	F	T	F	Static	F	F	F	F	High	F	Med
IoTMediator [22]	F	F	F	Hybrid	T	T	Pre-event	T	High	T	High
Ours	T	T	T	**Hybrid**	T	T	**Pre-event**	T	**Low**	T	**High**

Note: "DIC" means Distinguish between Interaction and Conflict. "Pre-event" means handling the conflict before it occurs.

living room). Without Zone awareness, this non-conflicting situation might be incorrectly flagged.

Compared to rule conflict detection, discussions on conflict mitigation are significantly scarcer. Most existing works merely issue alerts after detecting a rule conflict. For instance, IoTIE [2] suggests running checks immediately after installing a new application (i.e., automation rules) to prevent rule conflicts before formal usage. However, this arbitrary approach struggles to balance the wide variety of automation rules. IoTGuard [15] and IoTSafe [4] enforce security policies to prevent operations that lead the system into risky states. However, these policies originate from existing research or user definitions, making them difficult to generalize across diverse households. This places a heavy burden on ordinary users who are not security experts. Moreover, as devices evolve, users may add new devices that creating a "security vacuum" or vulnerability window where no policies are available, leaving openings for attackers. IoTMediator [22] proposes customized handling strategies for different rules, essentially generating corresponding options based on rule interaction patterns. Users simply need to select the correct solution. However, the paper does not mention the method or timing of user interaction (e.g., whether it occurs after static detection or when a conflict is found during dynamic monitoring). It relies entirely on user selection, preventing effective automation. Furthermore, if false positives occur—which our test results indicate is likely—it presents users with many unnecessary choices.

Some works also pay attention to user preferences. For example, some support setting user-defined policies or prompting users to modify automation rules based on conflict handling information, thereby allowing users to express their preferences to some extent. However, the expression of user preferences is inextricably linked to user-friendly interaction for ordinary users. Ordinary users typically do not understand the setting of custom security policies and find it difficult to modify automation rules to correctly handle rule conflicts while maintaining their preferences. IoTMediator [22] accepts user preferences to a certain degree by providing conflict handling strategy templates for users to select. However, it neglects user preferences during conflict detection, aggressively classifying rules configured by users for normal cyclic execution as interaction threats.

In this paper, we categorize the generalization ability of various solutions based on the system's tolerance for diverse smart home environments. Stronger generalization indicates a higher capacity to accept different home layouts and a variety of devices, as well as less reliance on general predefined security policies. We also categorize user effort based on the difficulty of system deployment. Higher user effort indicates that the system is more complex to deploy and enable, while lower effort implies it is easier to deploy and apply to smart home systems. These two metrics, in addition to the effectiveness of rule conflict detection and mitigation, are of significant concern to users.

6 Conclusion

Addressing the safety and usability issues caused by automation rule conflicts in smart homes, this paper proposes a comprehensive solution combining static analysis with dynamic monitoring and mitigation.

We utilize translation templates to convert complex rule, interaction, and conflict information into easy-to-understand natural language, thereby collecting user preferences and key configuration information during simple interactions with users. We introduce the TCAE rule model to capture the characteristics of physical channels and leverage formal verification to comprehensively identify six types of rule interactions, generating a Rule Interaction Dependency Graph. Within this graph, we accurately locate candidate rule conflict relationships, effectively reducing resource consumption during dynamic monitoring, and significantly improving conflict detection accuracy while reducing false positives.

Furthermore, the system can automatically generate customized handling strategies based on safety priorities. Through dynamic assertion verification, it triggers real-time conflict interception and the enforcement of customized handling strategies, thereby effectively mitigating conflicts before they actually occur.

Evaluation results demonstrate the superiority of our approach in detecting complex conflicts (including those involving physical channels). The automated and customized conflict handling strategies are feasible and easy to understand. Meanwhile, the system performance overhead remains within an acceptable range, proving the practical value of this method in enhancing the safety and reliability of smart home systems.

References

[1] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pp. 272–285, 2020.

[2] Z. Chen, F. Zeng, T. Lu, and W. Shu, "Multi-platform application interaction extraction for iot devices," in *2019 IEEE 25th international conference on parallel and distributed systems (ICPADS)*, pp. 990–995, IEEE, 2019.

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

[4] W. Ding, H. Hu, and L. Cheng, "Iotsafe: Enforcing safety and security policy withreal iot physical interaction discovery," in *Network and Distributed System Security Symposium*, 2021.

[5] B. Huang, H. Dong, and A. Bouguettaya, "Conflict detection in iot-based smart homes," in *2021 IEEE International Conference on Web Services (ICWS)*, pp. 303–313, IEEE, 2021.

[6] X. Li, L. Zhang, and X. Shen, "Diac: An inter-app conflicts detector for open iot systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–25, 2020.

[7] D. Xiao, Q. Wang, M. Cai, Z. Zhu, and W. Zhao, "A3id: an automatic and interpretable implicit interference detection method for smart home via knowledge graph," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 2197–2211, 2019.

[8] M. Nakamura, H. Igaki, and K.-i. Matsumoto, "Feature interactions in integrated services of networked home appliances," in *Proc. of Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*, pp. 236–251, Citeseer, 2005.

[9] H. Igaki and M. Nakamura, "Modeling and detecting feature interactions among integrated services of home network systems," *IEICE transactions on information and systems*, vol. 93, no. 4, pp. 822–833, 2010.

[10] H. Ibrhim, S. Khattab, K. Elsayed, A. Badr, and E. Nabil, "A formal methods-based rule verification framework for end-user programming in campus building automation systems," *Building and Environment*, vol. 181, p. 106983, 2020.

[11] P. Pradeep, A. Pal, and K. Kant, "Automating conflict detection and mitigation in large-scale iot systems," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 535–544, IEEE, 2021.

[12] M. Shehata, A. Eberlein, and A. Fapojuwo, "Using semi-formal methods for detecting interactions among smart homes policies," *Science of Computer Programming*, vol. 67, no. 2-3, pp. 125–161, 2007.

[13] Y. Sun, X. Wang, H. Luo, and X. Li, "Conflict detection scheme based on formal rule model for smart building systems," *IEEE Transactions on Human-Machine Systems*, vol. 45, no. 2, pp. 215–227, 2014.

[14] F. Alharithi, *Detecting conflicts among autonomous devices in smart homes*. PhD thesis, Florida Institute of Technology, 2019.

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

[16] A. A. Hamza, I. T. Abdel Halim, M. A. Sobh, and A. M. Bahaa-Eldin, "Hsas-md analyzer: a hybrid security analysis system using model-checking technique and deep learning for malware detection in iot apps," *Sensors*, vol. 22, no. 3, p. 1079, 2022.

[17] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno, "Detecting feature interactions in home appliance networks," in *2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pp. 895–903, IEEE, 2008.

[18] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, "Understanding and automatically detecting conflicting interactions between smart home iot applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1215–1227, 2020.

[19] M. Yagita, F. Ishikawa, and S. Honiden, "An application conflict detection and resolution system for smart homes," in *2015 IEEE/ACM 1st international workshop on software engineering for smart cyber-physical systems*, pp. 33–39, IEEE, 2015.

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

[21] D. Chaki and A. Bouguettaya, "Fine-grained conflict detection of iot services," in *2020 IEEE International Conference on Services Computing (SCC)*, pp. 321–328, IEEE, 2020.

[22] H. Chi, Q. Zeng, and X. Du, "Detecting and handling {IoT} interaction threats in {Multi-Platform}{Multi-Control-Channel} smart homes," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1559–1576, 2023.

[23] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated {IoT} safety and security analysis," in *2018 USENIX annual technical conference (USENIX ATC 18)*, pp. 147–158, 2018.

[24] M. Alhanahnah, C. Stevens, B. Chen, Q. Yan, and H. Bagheri, "Iotcom: Dissecting interaction threats in iot systems," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1523–1539, 2022.

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

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

[27] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 1439–1453, 2019.

[28] B. Huang, D. Chaki, A. Bouguettaya, and K.-Y. Lam, "A survey on conflict detection in iot-based smart homes," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–40, 2023.

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

APPENDIX A
SYSTEM PROMPT CONFIGURATION

Role Definition: You are a smart home expert who can handle conflicts in automation rules. **A. Defined Conflict Types:**

- Rule Trigger Conflict
- Rule Condition Conflict
- Rule Action Conflict
- Indirect Rule Trigger Conflict
- Indirect Rule Condition Conflict
- Indirect Rule Action Conflict

B. Detailed Conflict Descriptions:
Trigger Conflict: The execution of rule i causes the rule to be triggered.
Condition Conflict: The execution of rule i prohibits or allows the conditions of rule j.
Action Conflict: The execution actions of two rules conflict.
Indirect Trigger: The impact of rule i on another channel unexpectedly causes rule j to be triggered.
Indirect Condition: The impact of rule i on another channel causes the conditions of rule j to be allowed or disabled.
Indirect Action: Two rules interact with different devices that affect the same channel property.
Precedence Note: For Action Conflicts, status is updated based on execution order.

C. Rule Context and Execution Order:

- Index order starts from 0.
- `R1`: First rule (executed first).
- `R2`: Second rule (executed later).

D. Handling Strategies (Action Conflicts):
(Applies to Action & Indirect Action Conflicts) `{'Default Action': 0, 'Only execute {R1}': 1, 'Only execute {R2}': 2, 'Both executed, end with {R1}': 3, 'Both executed, end with {R2}': 4, 'Neither is executed': 5}`

E. Handling Strategies (Trigger/Condition):
(Applies to Trigger/Condition & Indirect types) `{'Default Action': 0, 'Only execute {R2}': 1, 'Default execution {R2}': 2, 'Cancel execution of {R2}': 3}`

F. Required Input Data Format:
The rules are given in the order of execution.

- R1 Info: `<name|id, content, description>`
- R2 Info: `<name|id, content, description>`
- Description of rule violation

G. Safety and Rationale Requirement: Select a strategy considering user perspective, resources, and safety (e.g., prevent unintended door openings, ensure valve function during a fire).

H. Required Output JSON Structure:
The response must be in the following JSON structure, without markdown or extra explanations.

```
{'policy': "index of policy",
  'reason': "reason why you choose this policy"}
```