

# TAPFixer: Automatic Detection and Repair of Home Automation Vulnerabilities based on Negated-property Reasoning

Yinbo Yu<sup>1,2</sup>, Yuanqi Xu<sup>1</sup>, Kepu Huang<sup>1</sup>, Jiajia Liu<sup>1</sup>

<sup>1</sup>*National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology, School of Cybersecurity, Northwestern Polytechnical University, China*

<sup>2</sup>*Research & Development Institute of Northwestern Polytechnical University in Shenzhen, China  
Email: {yinboyu, liujiajia}@nwpu.edu.cn, {yuanqixu, huangkepu}@mail.nwpu.edu.cn*

## Abstract

Trigger-Action Programming (TAP) is a popular end-user programming framework in the home automation (HA) system, which eases users to customize home automation and control devices as expected. However, its simplified syntax also introduces new safety threats to HA systems through vulnerable rule interactions. Accurately fixing these vulnerabilities by logically and physically eliminating their root causes is essential before rules are deployed. However, it has not been well studied. In this paper, we present TAPFixer, a novel framework to automatically detect and repair rule interaction vulnerabilities in HA systems. It extracts TAP rules from HA profiles, translates them into an automaton model with physical and latency features, and performs model checking with various correctness properties. It then uses a novel negated-property reasoning algorithm to automatically infer a patch via model abstraction and refinement and model checking based on negated-properties. We evaluate TAPFixer on market HA apps (1177 TAP rules and 53 properties) and find that it can achieve an 86.65% success rate in repairing rule interaction vulnerabilities. We additionally recruit 23 HA users to conduct a user study that demonstrates the usefulness of TAPFixer for vulnerability repair in practical HA scenarios.

## 1 Introduction

A recent spurt of progress in advanced technology (e.g., artificial intelligence, 5G, and cloud computing) has incredibly improved the automation and intelligence of the Internet of Things (IoT). To better provide automatic service for end users, an IoT programming framework named *Trigger-Action Programming* (TAP) has been widely applied in many home automation (HA) platforms, e.g., Samsung SmartThings [11], Apple Homekit [4], IFTTT [5], Home Assistant [3], Mi Home [7], and so on. In general, a TAP rule is defined in the form of “**IF** the *trigger* occurs **WHILE** the *condition* is met, **THEN** perform the *action*.” It describes how to operate a device (*action*) under the state constraint (*condition*) when an event or a state change (*trigger*) occurs in the HA system. For example,

“**IF** the user is present **WHILE** the CO<sub>2</sub> is above a predefined value, **THEN** open the window for 10 min”. TAP rules have greatly facilitated and enriched users’ lives.

TAP rules make it easier to connect various devices for collaborative automation. However, the surge in interactions between rules, especially those involving the complicated physical space (*i.e.*, the HA physical environment), challenges the correctness of TAP-based HA systems [16, 22, 28]. For example, turning the heater on can increase the temperature, which can then activate or enable other TAP rules. End users have little understanding of the incomprehensible relationship between the logical (*i.e.*, smart device behaviors defined by programs) and physical space. Hence, it is hard for users to configure TAP rules that align with their intentions [47], as well as manually identify and fix rule interaction vulnerabilities, resulting in unexpected device status (e.g., the AC and window are both turn on) or even safety risks (e.g., the door is unlocked when the user is not at home) in the HA system.

Recently, a significant number of advanced techniques have been proposed to secure TAP-based HA systems, including automation generation [35, 47], threat detection [12, 46], access control [18, 41], privacy leakage analysis [14, 31], and anomaly detection [16, 29, 42]. While these techniques make great contributions to analyzing rule interaction vulnerabilities, there is a noticeable lack of attention to resolving and preventing them. Some dynamic control-based methods [21, 28, 33, 43] are proposed to control rule enforcement at runtime to avoid risks according to specified safety policies. However, they do not eliminate the root cause of vulnerabilities (*i.e.*, rule semantic flaws) and can introduce additional running overhead. In contrast, static-based methods [17, 34, 47] can generate rule patches to correct rules. Unfortunately, these methods neglect dynamic factors (e.g., latency and physical interactions) that may change the practical effect of rule executions, and therefore have limited repair capabilities [16, 22].

To address these limitations, in this paper, we focus on vulnerability static repairing and design TAPFixer, an automatic vulnerability detection and repair framework for securing TAP-based home automation. To our best knowledge, TAP-

Fixer is the first work that can essentially detect and fix rule interaction vulnerabilities both in the logical and physical space. It is orthogonal to dynamic control-based methods in that it can statically reduce risks as possible before rule execution, allowing the latter to run with a lower running cost (*i.e.*, few enforcement policies) to prevent risks caused by unpredictable events (e.g., human interference with devices).

Our major contributions are summarized as follows:

- We design a formal model of rule interactions using finite automaton, which formalizes and embeds physical operating features into rule syntax to enable accurate static vulnerability detection. With such a model and a set of designed correctness properties, TAPFixer detects rule interaction vulnerabilities through model checking.
- We design a novel negated-property reasoning algorithm that can automatically construct rule patches to fix and radically eliminate vulnerabilities both in the logical and physical space. Given a violation of a property (*i.e.*, a counterexample), its core idea is to use negated properties to reason about negated counterexamples through an abstraction and refinement process, thereby identifying possible repair patches of the violation.
- We conduct a benchmark to evaluate the accuracy of TAPFixer in comparison to existing approaches. We then apply TAPFixer to market HA apps, where TAPFixer can obtain an 86.65% success rate in repairing found vulnerabilities on average. We also conduct a user study and performance analysis of TAPFixer to demonstrate how well it detects and repairs rule interaction vulnerabilities.

## 2 Preliminary

### 2.1 TAP Rule Formulation

In this section, we formulate TAP rules and their interactions based on existing advanced research [20, 39]. The execution of a TAP rule follows the logical sequence of trigger-condition-action:  $r := \langle t, c \rangle \mapsto a$ , where  $t$  is a *trigger*,  $c$  is a *condition*,  $a$  is an *action*, and  $\mapsto$  denotes a relationship of the sequential execution.  $r$  can be formulated as a set of constraints and assignments on **entity attributes**  $\mathbb{A}$  which are the intuitive abstract state expression of entities including logical states (e.g., time) and physical objects (including smart devices and physical attributes, e.g., humidity). To achieve a more accurate formal analysis of rule executions, we classify entity attributes as *Immediate* and *Tardy* types [46]. While the former (e.g., the state of a switcher) changes instantaneously, the latter (e.g., temperature) takes a period of time to make changes.

**Rule Syntax:** for  $r := \langle t, c \rangle \mapsto a$ , it can be defined as (1)  $t$  specifies a constraint on a certain attribute to activate  $r$ , e.g., humidity < 30%; (2)  $c$  is a set of constraints on one or more attributes, and all of them must be evaluated to be true for action executions; (3)  $a$  represents one or more assignments

on attributes. Actions can be divided into two types: *Immediate* and *Extended*. While the former (e.g., turning lights on) completes instantaneously, the latter (e.g., lowering the temperature to 20°C) needs a period of time to complete.

**Rule Semantic:** it is the abstract template describing characteristics of entities in TAP rules (e.g., working modes of the security camera), which can be formalized as follows:

$$r_S := t_S \times c_S \times a_S, \quad (1)$$

where  $t_S$ ,  $c_S$ , and  $a_S$  denote the semantics of trigger, condition, and action, respectively.

**Rule Configuration:** it is the abstract representation of attribute values used in TAP rules (e.g., night mode of the security camera), which can be formalized as follows:

$$r_C := t_C \times c_C \times a_C, \quad (2)$$

where  $t_C$ ,  $c_C$ , and  $a_C$  denote the configuration of trigger, condition, and action, respectively.

### 2.2 Rule Interaction Vulnerabilities

Using simplified rule syntax to automate complex HA environments is hard to avoid without errors, even for professional users [47]. The rule vulnerabilities may be caused by several reasons [16, 22, 45], e.g., user misconfiguration, misleading or deceptive rule apps, or state deception via device spoofing or channel injection. The fundamental root cause is logical flaws in rule semantics. These flaws are compounded during rule interactions among rules through triggers, conditions, or actions in logical or physical channels, which can lead to more unexpected or even vulnerable states. Existing works [20, 22, 46] have summarized various rule interaction vulnerabilities. In this paper, depending on how rules interfere, we categorize existing vulnerabilities into 3 basic patterns and further define 5 expanded vulnerability patterns based on specific interference contexts in each basic pattern. We give a concise description of these 8 patterns in follows and their details are described in Appendix A due to limited space.

**Basic Vulnerability Pattern (V1-V3).** In general, based on how triggers, conditions, and actions of a rule interfere with others, the basic vulnerability pattern is classified into three categories: Trigger-Interference **V1**, Condition-Interference **V2**, and Action-Interference **V3** vulnerabilities [22]. In **V1**, an action may produce an event that unexpectedly interferes with triggers of other rules and changes the rule context defectively, e.g., opening the light triggers the rule (“if the brightness is high, turn off the light”), leading to light strobe. In **V2**, an action may change the condition satisfaction of other rules and put the rule execution at risk, e.g., the user comes home late and interferes with the safety rule conditioned as sleep mode. In **V3**, rules with the same or different trigger may send conflicting commands to the same device, e.g., the door opens the moment it locks, which may cause a break-in.

**Expanded Vulnerability Pattern (V4-V8).** With in-depth research on vulnerability detection [20, 46], the basic vulnerability pattern is expanded by new vulnerabilities in more

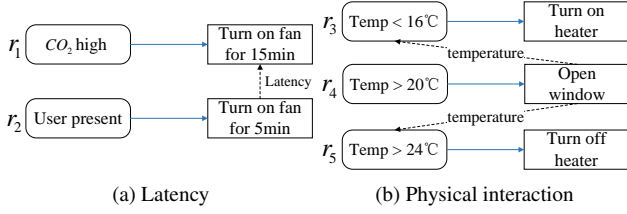


Figure 1: Motivation examples of rule interactions.

specific interference contexts. Expanded contexts mainly depict physical-related and latency-related features. The expanded pattern can be classified into 5 categories. In Expanded Trigger-Interference vulnerabilities (V4), an action may also interfere with triggers of other rules through channel-based interaction describing physical channel shared by actuators and sensors (e.g., both the thermostat and the temperature sensor work on temperature). Fig. 1(a) shows an example of V4. Expanded Action-Interference vulnerabilities (V5-V7) [46] includes *Disordered action scheduling* (V5), *Action overriding* (V6), and *Action breaking* (V7), which describes that an action may override or interrupt actions of other rules due to different time delays. Fig. 1(b) shows an example of V7. Expanded Condition-Interference vulnerabilities (V8) mainly occur when rules work on the same channel attribute but have different preferences.

### 3 Overview

#### 3.1 Motivation and Threat Model

**Motivation:** To enable easy participation in home automation, TAP rules are designed in an accessible form so that anyone without programming experience can easily get started. However, this ease of use eliminates the complex modeling of logical and physical spaces, leaving users prone to configure rules incorrectly. As a result, vulnerabilities in rule interactions are common during rule setting and may lead to unintended safety issues. There are two types of methods to address this issue: dynamic rule enforcement control and static rule syntax correction. However, considering complexities like latencies and physical interactions, these methods may not always prevent or rectify vulnerabilities effectively as expected.

We show two examples as our motivation: In Fig. 1(a): a violation occurs between  $r_1$  and  $r_2$ : if the indoor  $\text{CO}_2$  level rises up ( $>1000$  ppm) within 5 minutes after the user returns home,  $r_2$  will complete first and thus interrupt the ventilating fan in  $r_1$ , causing  $\text{CO}_2$  remaining at a high level; In Fig. 1(b): due to different indoor and outdoor temperatures, opening the window can cause the indoor temperature to drop below  $16^\circ\text{C}$  and fail to heat up to the desired  $24^\circ\text{C}$ . This physical interaction can cause  $r_3$  to be accidentally triggered or  $r_5$  to be blocked, which may lead to a fire hazard due to the heater being on the time while no one is at home. Dynamic methods tend to prevent these defects using access control (e.g., invalidating runtime effects of the fan shutdown in  $r_2$

and predicting temperature changes to prevent unexpected enforcement of  $r_3$  and  $r_5$  in advance). They do not eliminate the root causes of defects and will face the running overhead of dynamic monitoring or prediction. In contrast, static methods fix these defects by modifying or patching rules. However, existing methods suffer from poor repair quality. For example, they can fix the violation in Fig. 1(a) by adjusting the delay or turning the fan back on after  $r_1$  is interrupted, but  $r_1$  is still interrupted. The major reason is that they ignore the root cause of vulnerabilities both in logical and physical spaces.

**Threat model:** We assume that rule interaction vulnerabilities are induced in three aspects: 1) *misconfiguration by users*: the lack of safety practices for HA users can make it challenging to ensure global safety of installed rules, particularly when multiple family members share the HA system; 2) *misleading apps*: ambiguous or incorrect app descriptions may cause users to misinterpret app functionality, leading to rule conflicts. More seriously, attackers can trick users in this way to install malicious apps they published, thereby introducing specific rules to trigger interactive threats [21]; 3) *side-channel attacks*: an attacker can infer user and device activities by sniffing network traffic [44] and physical changes [16] to exploit specific data transmission delay or physical interference to launch unsafe rule interactions. Our TAPFixer mainly focuses on the first two aspects, but also supports checking the third threat by introducing arbitrary rule delay and nondeterminacy into rule models (see §4.2).

#### 3.2 TAPFixer Design Intuition

To secure TAP-based HA systems, we present TAPFixer, an automatic framework to detect and repair rule interaction vulnerabilities, as shown in Fig. 2. TAPFixer takes HA apps (e.g., SmartApp, IFTTT applets) and corresponding configurations installed in the HA platform (*i.e.*, *profiles*) as input, verifies their correctness via model checking with a set of pre-defined correctness properties, and generates repair patches for identified vulnerabilities. In the detection phase (see §4), given a correctness property  $\phi$ , TAPFixer extracts modeling information from profiles, formally models rule interactions ( $\mathcal{M}^\phi$ ), and performs model checking ( $\mathcal{M}^\phi \models \phi$ ) to identify the potential counterexample  $\text{CEX}^\phi$  (*i.e.*, a rule interaction vulnerability).  $\text{CEX}^\phi$  is an execution path that can reach the incorrect state space against  $\phi$  (*i.e.*,  $\neg\phi$ -space shown in Fig. 3). Our detection component is designed based on many existing advanced effects [12, 22, 28, 46], but it can model and analyze TAP rules with more logical and physical features (see Table 9 and 10) to achieve accurate vulnerability detection.

To repair  $\text{CEX}^\phi$ , we design a novel *negated-property reasoning* (NPR) algorithm in TAPFixer to generate rule patches. Its core idea (see Fig. 3) is to perform model checking with the negated property  $\neg\phi$  (*i.e.*,  $\mathcal{M}^\phi \models \neg\phi$ ) to generate possible repair patches. However, it is challenging since (1)  $\mathcal{M}^\phi$  is flawed and includes much invalid information; (2)  $\mathcal{M}^\phi$ 's state space may not contain effective repair clues. To this

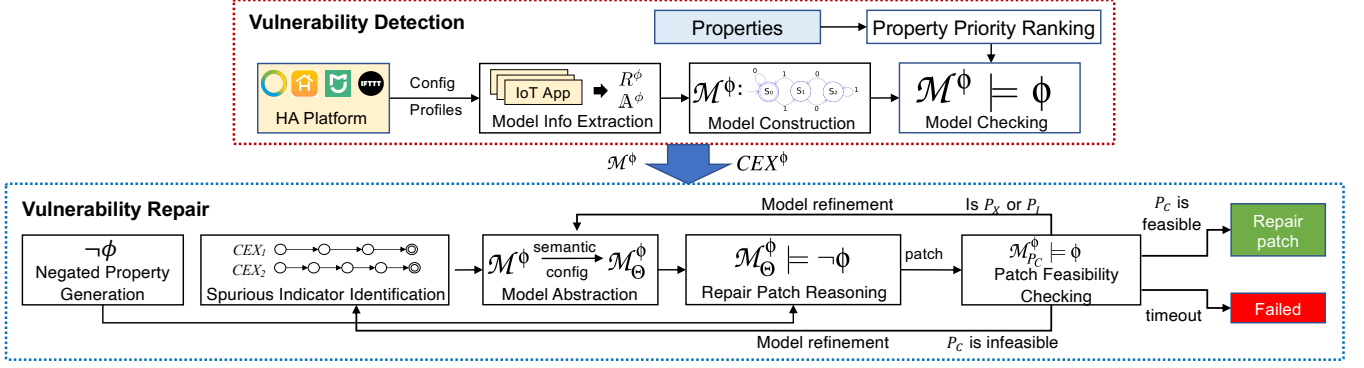


Figure 2: Overview of TAPFixer.

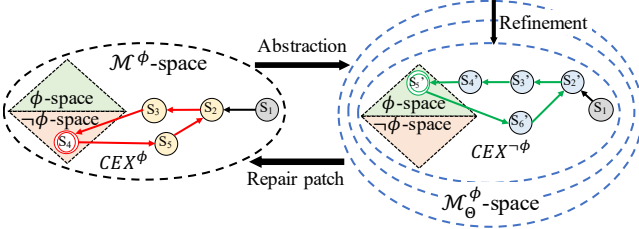


Figure 3: The core idea of our NPR algorithm for vulnerability repair. Model abstraction via interpolation is used to involve a larger state space for patch searching, so the negated counterexample  $CEX^{-\phi}$  can contain repair patches for the vulnerability  $CEX^{\phi}$ .

end, we design an adaptive model abstraction and refinement framework (similar to the CEGAR algorithm [23]) in NPR to derive the accurate patch. Its basic workflow is to abstract  $\mathcal{M}^{\phi}$  to compute an over-approximated model  $\mathcal{M}_{\theta}^{\phi}$ , perform model checking with the *negated property*  $\neg\phi$  to generate negated counterexamples  $CEX^{-\phi}$ , refine  $\mathcal{M}_{\theta}^{\phi}$  both on rule semantics and configurations to eliminate infeasible  $CEX^{-\phi}$ , and construct patches from feasible  $CEX^{-\phi}$  and  $\mathcal{M}_{\theta}^{\phi}$  (see §5).

Our key insight of using model abstraction and refinement for patch generation is that abstraction can extend a larger model state space to introduce more possible repair clues and refinement guided by infeasible negated counterexamples can eliminate impossible and unrelated state transitions. An  $CEX^{-\phi}$  identified from  $\mathcal{M}_{\theta}^{\phi}$  can enter the  $\phi$ -space which provides a possible path to patch the original model  $\mathcal{M}^{\phi}$  to prevent it from entering into the  $\neg\phi$ -space. Taking  $\mathcal{M}^{\phi}$  in Fig. 1(b) with  $\phi$  (“if no one is at home, the heater should be turned off”) for example, by performing NPR with  $\neg\phi$  (“if no one is at home, the heater should be turned on”), we can obtain an interpolated model  $\mathcal{M}_{\theta}^{\phi}$  and  $CEX^{-\phi}$  which include a new execution path (“turn the heater off when the user leaves”) and a new condition constraint (“user is at home”) on  $r_3$ . This path and constraint in  $\mathcal{M}_{\theta}^{\phi}$  make it violate  $\neg\phi$ , but can construct as the repair patch of  $CEX^{\phi}$ .

## 4 Rule Modeling and Vulnerability Detection

### 4.1 Modeling Information Extraction

To formally model TAP rules, TAPFixer first extracts device information from device platform websites (e.g., SmartThings

[25]), transforms different implementations of automation apps into the format of TAP rules  $R$ , and identifies the rule set  $R^{\phi}$  and attribute set  $\mathbb{A}^{\phi}$  associated with a given property  $\phi$ . There are two mainstream implementation methods of the TAP paradigm: general-purpose programming languages (e.g., Groovy in SmartThings apps) and natural languages (e.g., IFTTT applets). Following existing literature [12, 22, 45], TAPFixer uses static program analysis and natural language processing (NLP) techniques to transform these apps into a set of TAP rules  $R$ . Details are described in Appendix B.

Given a set of properties, TAPFixer filters out relative entities from the extracted rule set to compact the model through possible logical or physical interactions in a recursive way. For logical interactions, it first extracts the set of entity attributes  $\mathbb{A}^{\phi}$  presented in the property  $\phi$ , extracts rules  $R^{\phi}$  containing attributes in  $\mathbb{A}^{\phi}$  from the rule set  $R$ , and appends new attributes in  $R^{\phi}$  to  $\mathbb{A}^{\phi}$  accordingly. For physical interactions, we consider 9 frequently used or safety-sensitive *physical channels*, mainly referring to [39]: temperature, illuminance, motion, smoke, humidity, CO, CO<sub>2</sub>, sound, and weather status. Since TAPFixer is performed in a static manner, it cannot predict interaction behaviors in these channels dynamically. Inspired by [46], we manually conduct a qualitative analysis of device effects on physical channels in a few fixed home environments and quantify these effects in Appendix Table 11 based on existing results [16, 28, 46] and our collected sensor data. Note that the accuracy of this manual analysis may reduce as the home environment changes. Hence, to control model errors, we set these quantitative effects to a value range rather than a single value according to our measurement and also model the impact of non-deterministic interferences to a certain extent (see §4.2). With these quantified physical interaction effects, TAPFixer further appends both  $R^{\phi}$  and  $\mathbb{A}^{\phi}$  in the same way. This process is executed recursively until  $R$  is empty or no elements can be appended to  $\mathbb{A}^{\phi}$ .

### 4.2 Rule Interaction Modeling

Utilizing extracted  $R^{\phi}$  and  $\mathbb{A}^{\phi}$ , TAPFixer models rule interactions as a finite automaton (FA). To enhance the accuracy and integrity of modeling, we introduce practical dynamic features to better align with the real world as follows:



**Latency Modeling:** we involve latency as a key element of rule syntax since a rule may be not executed immediately due to different delays [20]: (1) the delay  $l_1$  defined in rules for specifying the execution time (e.g., turn lights on for 10min) or postponing action execution (e.g., the *runIn* function in SmartThings). The key to modeling  $l_1$  is to determine when the delay is completed. TAPFixer both considers explicit and implicit  $l_1$ : the former is modeled as a timer variable configured with a specified timeout value (e.g., for “turning on the fan for 5min”, the timer has a timeout value of 300); the latter has no explicit timer, but a Boolean variable *wait\_trigger* to denote if the targeted status is present after an extended action starts; (2) the delay  $l_2$  on a *tardy attribute* change to a certain value, e.g., time spent on the temperature to 20°C. It is the key factor affecting physical interactions. We discuss it in the next aspect; (3) the platform delay for data updating from devices to the platform [36]. It is caused by updating intervals of sensors, within which a physical entity attribute  $\mathbb{A}_{phy}$  may be inconsistent with its mirror logical variable of  $\mathbb{A}_{log}$  in the platform. We observe that the updating interval varies from a few seconds (e.g., present sensors) to a dozen minutes (e.g., temperature sensors). TAPFixer sets updating interval variables for  $l_3$  of different sensors to model hysteretic updates.

Given the above latency, we further formalize  $r$  as follows:

$$r_{immd} := (t, c) \xrightarrow{l_1} a_{immd}, \quad (3)$$

$$r_{ext} := (t, c) \xrightarrow{l_1} a_{ext} \xrightarrow{l_2} a_{\square}, \quad (4)$$

$$\mathbb{A}_{phy} \xrightarrow{l_3} \mathbb{A}_{log}, \quad (5)$$

where  $r_{immd}$  is an immediate rule,  $r_{ext}$  is an extended rule,  $a_{immd}$  is an immediate action,  $a_{ext}$  is an extended action,  $a_{\square}$  is the completion of  $a_{ext}$ .

**Physical Interaction Modeling:** Device actions can interact with environmental attributes via *physical channels*. This physical interaction extends the way rules interact. We model physical channels based on the physical effects of immediate and tardy attributes and the three following realistic dynamic physical features (listed in Appendix Table 11):

(1) *Implicit physical effect:* in addition to explicit effects of device actions, there are also implicit physical effects. For instance, opening a window can explicitly accelerate air circulation and implicitly affect the indoor temperature. We map device capabilities to physical channels they implicitly affect for rule modeling;

(2) *Joint physical effect:* devices in the same room or sharing the same physical channel may cause joint physical effects at proper distances [28]. In this case, the actual effect of joint operations  $\bigwedge_{i=1}^n a^{(i)}$  is the combination of individual effects. Hence, for each physical channel, we filter out devices affecting it, enumerate their co-execution scenarios, and model the sum of attribute value changes as the joint physical effect.

(3) *Nondeterminacy:* there are non-deterministic physical characteristics that may vary from the case, e.g., the fire intensity, and differences in device execution due to the battery drain. Owing to this dynamic variability, TAPFixer encodes such factors as a range of randomly selected values rather than specific values, e.g., the timing threshold for evaluating the water valve to eliminate fires is set in a random way.

Finally, we formulate the *physical interaction* as follows:

$$\bigwedge_{i=1}^n a^{(i)} \xrightarrow{l_2} \mathbb{A}, \quad (6)$$

where  $\xrightarrow{l_2}$  denotes a physical interaction,  $\bigwedge$  denotes a joint physical relation, and  $l_2$  is the physical effect delay. If  $\mathbb{A}$  is an immediate channel attribute,  $l_2$  is equal to 0; otherwise, we set  $l_2$  according to Appendix Table 11 for tardy attributes.

**Model Construction:** Given  $R^{\Phi}$ ,  $\mathbb{A}^{\Phi}$ , and these above models, TAPFixer translates  $\mathbb{A}^{\Phi}$  as automaton variables and constructs the FA  $\mathcal{M}^{\Phi} := \{S, I, \Sigma\}$ , where  $S$  is a finite set of all states,  $I \subseteq S$  is the initial state set, and  $\Sigma \subseteq S \times S$  is a set of state-to-state transitions. A state  $s_i \in S$  represents as a set of values for all automaton variables. Taking  $r_2$  in Fig. 1(a) for example,  $S$  is permutations between {user.present, user.not\_present} and {fan.on, fan.off}, and  $I$  is a state arbitrarily selected from it.

TAPFixer formulates rule executions, environmental changes, and physical interactions as  $\Sigma$ . For a rule  $r_k := \langle t_k, c_k \rangle \mapsto a_k$  and the environmental attribute set  $\mathbb{A}$ , a transition function  $\varphi(s_i, s_j) \in \Sigma$  is defined as follows:

$$\varphi(s_i, s_j) = \begin{cases} s_i \times (s_j \leftarrow a_k(s_i)), & (t_k(s_i) \wedge c_k(s_i)) \vee \mathbb{A}(s_i) \\ s_i \times s_j, & \text{otherwise.} \end{cases} \quad (7)$$

where  $s_i$  and  $s_j$  are the current and next automaton state, respectively,  $\leftarrow$  means a dependency relationship caused by both logical and physical interactions,  $t_k(s_i)$ ,  $c_k(s_i)$ ,  $a_k(s_i)$ , and  $\mathbb{A}(s_i)$  are the predicate for  $t_k$ ,  $c_k$ ,  $a_k$ , and  $\mathbb{A}$  in the state  $s_i$ , respectively.  $\mathbb{A}(s_i)$  is the prerequisite that represents these environmental attributes that can naturally change (e.g., rainy conditions). The state  $s_i$  will transfer to  $s_j$  by action executions if both  $t_k(s_i)$  and  $c_k(s_i)$  are satisfied or  $\mathbb{A}(s_i)$  is true; otherwise, it remains unchanged. For instance,  $\Sigma$  is the transition between the state {user.not\_present, fan.off} and {user.present, fan.on} that corresponds to  $t_2$ ,  $c_2$ ,  $a_2$  in  $r_2$ .

The rule model may involve a large number of variables, some of which have large ranges (e.g., indoor illuminance can be 0 to 2000 lux), resulting in a large state space for model verification. We note that given a set of rules, an attribute’s values are activated on a limited range. Values outside this range will not affect rules’ activities. Hence, to avoid state explosion, we collect all configurations defined in rules and globally compress the range of all model variables to an optimized smaller range. For example, the concerned value set of temperature in Fig. 1(b) is {16, 20, 24}, so we can have a smaller range [15, 25] rather than the full value range of temperature. Besides, TAPFixer now has not considered floating-point numbers and it performs rounding operations on floating-point

values before value compression. This is because we note that HA users are insensitive to them and generally customize rules with low-precision values, and most home sensors typically only provide coarse-grained measurement values.

### 4.3 Property Specification and Prioritization

TAPFixer uses correctness properties for vulnerability detection. A property is a criterion to describe what automation behavior is safe or not. Generally, it can be expressed in linear temporal logic (LTL) [40] which describes the relative or absolute order of behaviors in the system (e.g., the next state denoted by **X**, the subsequent path denoted by **F**, and the entire path denoted by **G**). Based on properties previously defined in [12, 19, 22, 31, 45–47], we select and refine them according to safety-sensitive and commonly used devices and supply more properties for different scenarios (e.g., properties with permitted latencies). We finally conduct 53 properties for vulnerability detection as shown in Appendix Table 15.

Additionally, TAPFixer also allows users to specify their desired correctness properties. To facilitate users' expression of properties, there are various natural language templates [47], e.g., *[state]* should *[always]* be active, *[event]* should *[never]* happen if *[state<sub>1</sub>, ..., state<sub>n</sub>]*. However, from the perspective of LTL formula, different templates may be equivalent in logic. Hence, we conduct four logical equivalence relations to group 9 types of language templates into 2 types of logical templates for property specification: Event-based and State-based properties, as shown in Table 1. While the former focuses on identifying and handling exceptions timely, the latter focuses on continuously preventing exceptions from occurring, which are often combined to ensure safety. Based on generalized templates, we divide properties into *pre-proposition* and *post-proposition* using the implication symbol  $\Rightarrow$  to specify event- and state-based negated properties. Besides, we introduce the variable recording the previous state based on the jump features of events to distinguish between event and state in the model. We discuss these relations and language templates in Appendix D and Table 12 due to space limitation.

Users can both pick up concerned properties in Table 15 and also specify their properties using the above 2 templates. Given a set of correctness properties, they may conflict with each other if they share the same device capabilities. So, it may be impossible to secure all properties for the same rule set. An example of conflicting properties is “close windows when it rains” and “open windows when CO is detected”. TAPFixer resolves property conflicts by priority ranking, which ensures that the property with a higher priority can be fixed first, even if a conflict exists. We formulate two types of priorities (*pre-proposition* and *post-proposition*) based on the composition of properties: the former mainly describes the priority of environment entities, e.g., CO has a higher priority than rainy weather; the latter mainly describes the priority of device capabilities, e.g., locking the door has a higher priority than opening it. For properties that share the same device ca-

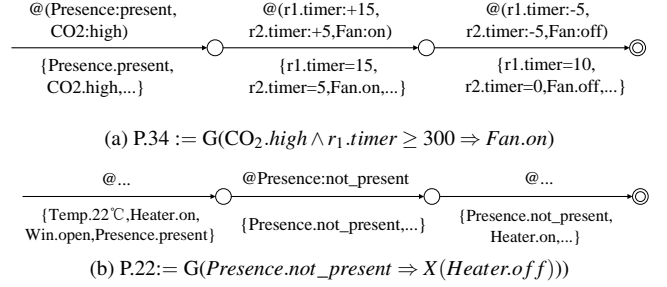


Figure 4: Example of counterexamples and their violating state. We use @ to denote the event and omit entity attributes that do not change or are irrelevant to the violation. The state with double circles denotes the violating state.

pability, TAPFixer prioritizes them based on pre-proposition priority and further sorts them based on post-proposition priority if they have the same priority. Properties with the same final priority are assigned in a random order. We list all correctness properties we used in Appendix Table 15.

### 4.4 Vulnerability Detection

Given an LTL property  $\phi$  and a rule interaction model  $\mathcal{M}^\phi$ , TAPFixer detects rule interaction vulnerabilities via model checking  $\mathcal{M}^\phi \models \phi$ . If there is a vulnerability, the model checker will output a counterexample  $CEX^\phi$ , which is a sequence of automaton states in the FA as follows:

$$CEX^\phi := \langle (s_1, \phi_1), \dots, (s_k, \phi_k) \rangle. \quad (8)$$

A  $CEX^\phi$  contains a state  $s^k$  violating against  $\phi$ , i.e.,  $s^k$  reaches the  $\neg\phi$ -space (see Fig. 3, e.g., the window opening when it rains). For example, Fig. 4 illustrates  $CEX^\phi$  of Fig. 1(a) and (b) against the property P.34 with the permitted time be 10 min and P.22, respectively: in Fig. 4(a), the proposition corresponding to the violating state  $Fan = on$  against P.34 is false since the fan turns off without running for at least 10 min; in Fig. 4(b),  $Presence = not\_present$  is satisfied but  $Heater = off$  in the next state is not for P.22.

## 5 Rule Interaction Vulnerability Repair

To fix rule interaction vulnerabilities, we need a patch that can both fix logical errors and change violating physical effects. In this section, we design a negated-property reasoning (NPR) algorithm in TAPFixer for vulnerability repair. Once TAPFixer finds a counterexample  $CEX^\phi$  for the property  $\phi$  in the model  $\mathcal{M}^\phi$ , it generates the negated property  $\neg\phi$  (see § 5.1) and performs NPR to generate a repair patch shown in Alg. 1. Given  $CEX^\phi$ ,  $\phi$ ,  $\neg\phi$ ,  $\mathbb{A}^\phi$ ,  $rnd = 0$ , and  $\mathcal{M}^\phi$  as input, NPR performs a recursive process similar to counterexample-guided abstraction and refinement (CEGAR) to find a globally feasible repair patch  $P$ : it first analyses  $CEX^\phi$  to obtain the spurious indicator  $I_s$  (Line 3, see §5.1), abstracts  $\mathcal{M}^\phi$  as an over-approximated model  $\mathcal{M}_\Theta^\phi$  and reasons  $P$  using  $\mathcal{M}_\Theta^\phi \models \neg\phi$  (Line 6-7, see §5.2). Then, NPR performs local and global

Table 1: Templates of correctness property and negated property.

Property Type	Natural Language Template	Property LTL Template	Negated Property LTL Template
Event-based	IF $[state_1, \dots, state_n]$ , $[event]$ should $[always]$ happen.	$G(\wedge_{i=1}^n state_i \Rightarrow X(event))$	$G(\wedge_{i=1}^n state_i \Rightarrow \neg X(event))$
State-based	WHEN $[state_1, \dots, state_n]$ , $[state]$ should $[always]$ be active.	$G(\wedge_{i=1}^n state_i \Rightarrow state)$	$G(\wedge_{i=1}^n state_i \Rightarrow \neg state)$

**Algorithm 1: Negated-property Reasoning (NPR)**


---

**Input:** Counterexample  $CEX^\phi$ ; Property  $\phi$  and negated property  $\neg\phi$ ; List of related entity attributes  $\mathbb{A}^\phi$ ; Number of recursive executions  $rnd$ ; Model  $\mathcal{M}^\phi$ ;

```

1 An empty patch  $P$ ;
2 if  $CEX^\phi$  then
3    $I_s \leftarrow \text{searchSpuriousIndicator}(CEX^\phi, \phi)$ ;
4    $P_I, P_X, iter \leftarrow CEX^\phi, CEX^\phi, 1$ ;
5   while  $iter < ITER\_LIMIT$  do
6      $\mathcal{M}_\Theta^\phi \leftarrow \text{abstractModel}(P_I, P_X, \phi, \mathbb{A}^\phi, \mathcal{M}^\phi)$ ;
7      $P \leftarrow \text{reasonPatch}(\mathcal{M}_\Theta^\phi, \neg\phi)$ ;
8      $P_C, P_I, P_X \leftarrow \text{checkLocalFeasibility}(P, \phi, I_s, CEX^\phi)$ ;
9     // classify  $P$  as  $P_C, P_I$  or  $P_X$ 
10    if  $P_C$  then
11       $CEX_{P_C}^\phi, \mathcal{M}_{P_C}^\phi \leftarrow \text{verifyGlobalFeasibility}(\phi, \mathcal{M}^\phi, P_C)$ ;
12      if  $CEX_{P_C}^\phi$  then
13         $rnd \leftarrow rnd + 1$ ;
14        if  $rnd < ROUND\_LIMIT$  then
15           $P \leftarrow \text{NPR}(CEX_{P_C}^\phi, \phi, \neg\phi, \mathbb{A}^\phi, rnd, \mathcal{M}_{P_C}^\phi)$ ;
16        else Break;
17    else  $iter \leftarrow iter + 1$ ;

```

---

**Output:**  $P$

---

feasibility checking to validate the effectiveness of  $P$  for fixing  $CEX^\phi$  (Line 8-14, see §5.3). If feasible, TAPFixer maps predicates in  $P$  into corresponding rules in  $R^\phi$  to fix  $CEX^\phi$  (see §5.4). Take Fig. 1(b) for example, NPR first extracts  $I_s$  from  $CEX^\phi$  in Fig. 4(b) (i.e., the violating state “the heater is on when the user leaves”) as the indicator. Then, it collects entities in  $\mathcal{M}^\phi$  and P.22, expands the state space of  $\mathcal{M}^\phi$  by enlarging its predicate space with non-appearing but possibly-relative entities (e.g., the user’s present status) to construct  $\mathcal{M}_\Theta^\phi$ . With  $\mathcal{M}_\Theta^\phi$  and the negated P.22 (if the user is not at home, the heater should be turned on), NPR reasons a patch solution  $P$  to  $CEX^\phi$  which includes creating a new rule (“if the user is not present then turn off the heater”) and a condition (“the user is at home”) set into  $r_3$ .

## 5.1 Negated Property Generation and Spurious Indicator Identification

The core method of NPR is to use the property  $\neg\phi$  negated to the property  $\phi$  to reason about the repair patch for the identified  $CEX^\phi$ . With our two merged LTL property templates, we give their negated templates in Table 1 and generate  $\neg\phi$ . To prove the soundness of property negation, we manually prove these negations shown in Appendix D.

Besides, the key indicator of whether a patch can fix the

vulnerability  $CEX^\phi$  is the ability to eliminate the violating state  $s^\neq$  in  $CEX^\phi$  (e.g., “the heater is on when the user leaves” in Fig. 4(b)). Hence, we define  $s^\neq$  as the *spurious indicator*  $I_s = s^\neq$  to assess the ability of repair patches. In many cases of  $CEX^\phi$ ,  $s^\neq = \phi_k(s_k)$ . But, there are still some cases:  $CEX^\phi$  is a lasso-shaped path from  $s_1$  which has a cycle (e.g., Fig. 3), so  $\phi_k(s_k) \neq s^\neq$ . Hence, NPR further analyses  $CEX^\phi$  to find  $I_s$ . Based on the given property  $\phi$  and  $CEX^\phi$ , NPR converts the problem of identifying  $I_s$  into the propositional logic satisfiability problem. Specifically, NPR evaluates each automaton state as a judging unit and analyzes each of its entity values based on  $\phi$ ’s LTL encoding [15]. If an entity in a state is unsatisfactorily encoded, this state is considered as  $I_s$ .

## 5.2 Patch Reasoning

The core problem of NPR is to generate an abstract model  $\mathcal{M}_\Theta^\phi$  via model abstraction and refinement, with which NPR can reason a global feasible patch via  $\mathcal{M}_\Theta^\phi \models \neg\phi$ . Both constraints and assignments in  $\mathcal{M}^\phi$  can be expressed by predicates whose entity attributes and values represent model semantics and configurations, respectively. We define two types of predicates: *status predicate* (denoted by  $takeValue(x)$ ) depicts “an entity attribute  $x$  takes the value of  $Value$ ”, where  $x$  is the semantic and  $Value$  is the configuration; *trigger predicate* (denoted by  $isTrigger(y)$ ) depicts “the constraint related to an entity attribute  $y$  is the state trigger”. Note that the trigger predicate is necessary for a rule to represent its existence in the model. A TAP rule can be expressed as a compound proposition in the form of predicates in the model. For example,  $r_3$  in Fig. 1(b) can be expressed as  $isTrigger(Temp) \wedge takeBelow 16(Temp) \mapsto takeOn(Heater.switch)$ . Based on predicates, we now discuss our design of model abstraction via interpolation in terms of semantics and configuration, repair patch reasoning, and reasoning-guided abstraction refinement in NPR as follows:

**Model Semantic Abstraction.** Abstracting model semantics via predicate interpolation can introduce new behaviors to the current state space for patch construction. Given the model  $\mathcal{M}^\phi$  related to a property  $\phi$ , NPR extracts predicates from model transitions to merge as a universal predicate set  $U$  and defines a predicate set  $\Theta$  for semantic interpolation, whose initial items are predicates defined in  $\phi$ . To perform semantic interpolation, NPR chooses these transitions in  $\mathcal{M}^\phi$  that contain these semantics used in  $\Theta$  as interpolation targets. We discuss semantic interpolation in two defective scenarios.

For the defective scenario that requires perfecting existing rules, repairing is the problem of perfecting status predicates in all model states. NPR interpolates each state  $s_i$  in  $\mathcal{M}^\phi$  with the difference predicate set  $U - s_i$ ,

so as to provide a full model space for predicate modification or generation based on new semantics. Take  $r_3$  in Fig. 1(b) for example, its  $U$  and  $S$  consists of predicates on attributes  $\{Temp, Presence, Heater.switch\}$  and  $\{Temp, Heater.switch\}$ , respectively. Hence, its  $\Theta$  consists of predicates on  $\{Presence\}$ , and these predicates are connected by disjunction  $\vee$ . TAPFixer interpolates  $r_3$  as follows:

$$\begin{aligned} &isTrigger(Temp) \wedge takeBelow 16(Temp) \wedge (takeOn(Presence) \\ &\vee takeOff(Presence)) \mapsto takeOn(Heater.switch) \end{aligned}$$

For the defective scenario that requires new rules, repairing is the problem of generating new transitions that have a new trigger predicate. To generate an interpolated transition, NPR includes  $U$  into  $\Theta$  to generate its state with a trigger and state predicate and uses the action of generating the device state desired in  $\phi$  as its action. Take Fig. 1(a) for example, NPR joins predicates of the same type by disjunction  $\vee$  and predicates of different types by conjunction  $\wedge$  to construct an interpolated transition:

$$\begin{aligned} &(isTrigger(CO_2) \vee isTrigger(Fan) \vee isTrigger(Presence)) \\ &\wedge (takeHigh(CO_2) \vee takeModerate(CO_2) \vee takeLow(CO_2)) \\ &\wedge (takeOn(Fan) \vee takeOff(Fan)) \wedge (takePresent(Presence) \\ &\vee takeNotPresent(Presence)) \mapsto takeOn(Fan) \end{aligned}$$

Predicate interpolation can induce a huge state space and may cause a state explosion for satisfiability checking. We observe that in fact, the satisfiability of many predicates can be denoted by one predicate due to physical constraints among them. Hence, NPR introduces a Boolean variable to represent one or more joint predicates. For instance, the satisfiability of  $takeOn(Fan)$  is denoted by  $flag \rightarrow takeOn(Fan)$  and its truth value is equivalent to  $flag$ . For status predicates of the same entity attribute, NPR constrains the sum of their flags to be 1 or 0, since an entity either has only one state at a time or does not exist. Similarly, for the trigger predicate of a TAP rule, NPR constrains its sum to be 1 or 0 since the rule has only one trigger at a time or does not exist. Simplifying the satisfiability of predicates with a single Boolean variable can avoid state explosion and reduce model checking overhead.

**Model Configuration Abstraction.** There are many enumerated (e.g., switch) and numerical configurations (e.g., illuminance) in the HA system. While the former has a small enumeration scope, the latter may have a large range resulting in an excessively large number of predicates. Hence, for abstracting a numerical configuration, NPR only takes a few practical values as its abstraction set for predicate generation, rather than the full value scope to reduce the state space of abstract models: NPR first finds its variable values in rules that are logically and physically related to, as its initial abstraction set; afterward, according to the physical effect received by it in one unit of time, NPR appends the change value of each element in its abstraction set in three units of time to

the set and removes duplicated values. For instance, the initial abstraction set of temperature in Fig. 1(b) is  $\{16, 20, 24\}$ , and its complementary values range from 13 to 27 caused by physical effects, which is much smaller than the original scope of temperature.

Latency is a special model configuration since it contains both numerical and enumerated configurations. We observe that fixing the vulnerability by adjusting or even removing the numerical delay defined in the rules would violate the user's intent. To be user-centric, NPR selects enumerated latency configurations (used to postpone action execution such as *runIn* in SmartThings and *wait\_trigger* in HomeAssistant) as the abstraction scope and follows the above abstraction method to construct their abstraction set, which preserves the desired delay. Take  $r_1$  in Fig. 1(a) for example, the configuration of its action can be abstracted in the predicate form of the original delay type *take15minOn* and other delay type *takeDelayOn* as follows:

$$\begin{aligned} &isTrigger(CO_2) \wedge takeHigh(CO_2) \\ &\mapsto take15minOn(Fan) \vee takeDelayOn(Fan) \end{aligned}$$

For *takeDelayOn* (i.e., waiting for  $CO_2$  to be no longer high, e.g., 1000 ppm), NPR sets a Boolean variable *wait\_trigger* as its abstraction set. NPR performs model configuration abstraction after model semantic abstraction is finished.

**Repair Patch Reasoning.** Through abstraction, the abstract model  $\mathcal{M}_\Theta^\phi$  involves a larger state space which may encompass relevant repair patches. NPR performs the model checking ( $\mathcal{M}_\Theta^\phi \models \neg\phi$ ) to identify a negated counterexample  $CEX_\Theta^\phi$  which is an execution path that can reach the secure  $\phi$ -space. That means  $\mathcal{M}_\Theta^\phi \models \neg\phi$  and  $CEX_\Theta^\phi$  may provide a potential solution for fixing  $CEX_\Theta^\phi$ . Hence, we call  $CEX_\Theta^\phi$  a possible repair patch (i.e.,  $P = CEX_\Theta^\phi$ ). However, due to impossible paths or states introduced by abstraction, the patch may be infeasible or be able to introduce other new vulnerabilities in the original rule model. Hence, NPR validates its feasibility (see §5.3), and for infeasible patches, NPR refines  $\mathcal{M}_\Theta^\phi$  as follows. In addition, it is possible that there is no available patch in the current abstract state space due to inaccurate abstraction or impossible state for  $\phi$ . NPR will further refine the model before timing out.

**Reasoning-guided Abstraction Refinement.** The fundamental reason for infeasible patches is coarse-grained abstraction. To eliminate invalid repair patches, NPR first classifies a repair patch  $P$  as  $P_C$  (correct),  $P_I$  (implausible), and  $P_X$  (plausible but incorrect) according to feasibility checking in §5.3, and refines the granularity of abstraction to exclude infeasible transitions using the invalid patch ( $P_I$  or  $P_X$ ) as follows.

To eliminate interpolations leading to  $P_I$ , NPR extracts the automaton subsequence before the spurious indicator and keeps certain critical automaton variables unchanged within the abstraction. Specifically, for tardy attributes and immediate attributes that are not affected by the device (e.g., rainy



weather and the activation of the motion sensor), NPR collects their values in each state of the subsequence and keeps them unchanged during model refinement. To eliminate interpolations leading to  $P_X$ , NPR extracts these incorrect combinations of predicates in the previous abstraction and removes them by appending an invariant to the abstract model during refinement. For instance, changing the weather to fix the vulnerability in Fig. 4(b) is plausible but incorrect. To eliminate this  $P_X$ , NPR adds a negation invariant  $\neg takeNotRainy(Weather)$  in the next abstraction, which avoids the weather predicate incorrectly taking the value of not rainy.

Through iterations of model abstraction and refinement, NPR iteratively refines the abstraction granularity and eliminates invalid repair patches until a global feasible patch is found. The procedure is repeated *ITER\_LIMIT* times.

### 5.3 Patch Feasibility Checking

Given a patch  $P$ , NPR needs to validate its feasibility for vulnerability repair. A straightforward method is to update the model using  $P$  and verify with  $\phi$ . However, such a method may have an expensive checking overhead. We observe that some patches contain impossible changes to environment constraints (e.g., weather), which can be identified locally. Hence, we design the method of patch feasibility checking for NPR including local and global feasibility checking.

**Local Feasibility Checking.** NPR aims to perform local feasibility checking to distinguish the patch  $P$  as  $P_C$ ,  $P_I$ , or  $P_X$ . We summarize the following two interaction scenarios containing spurious constraint changes:

(1) *Altering physical environment attributes unaffected by device executions* to eliminate the violating state  $s^\neq$  is invalid. According to the position where environment attributes are changed in  $CEX^\phi$ ,  $P$  can be categorized as  $P_I$  or  $P_X$ . For instance, the vulnerability  $CEX^\phi$  of windows opening during rainfall can be repaired by changing the weather to no rain in the preceding state of  $s^\neq$ , but this is obviously not possible. The above example belongs to  $P_X$  since changing weather can eliminate  $s^\neq$ , but is infeasible. Regarding  $P_I$ , taking Fig. 4(b) for example, the implicit effect that opening windows will decrease the indoor temperature only occurs when the outdoor temperature is lower than the indoor temperature; otherwise, the defect rule  $r_5$  will never be blocked (e.g., the outdoor temperature is 26°C). This scenario belongs to  $P_I$  since it changes the fixed outdoor temperature from 12°C to 21°C, which is unrealistic and cannot eliminate  $s^\neq$  in  $CEX^\phi$ .

(2) *The non-elimination of the violating state* in logical environment  $CEX^\phi$  is another unsatisfied case. Vulnerabilities may not disrupt the entire rule execution and safe executions can still exist in the rest of rule interactions, even if the current vulnerability is not eliminated. Take the scenario in Fig. 4(b) for example, there may be a safe situation where rules are not intersected when the indoor temperature is 18°C and the heater is off while the user is not at home before the violation

occurs, but the violating state still occurs in the same place. Such scenarios also correspond to  $P_X$  since they are relevant to  $\phi$  but cannot eliminate the violating state.

To identify spurious patches, NPR extracts environment constraints from the patch  $P$  and compares them with those in  $CEX^\phi$  to classify  $P$  as  $P_I$  or  $P_X$  as follows: (1) to identify  $P_I$ , NPR adopts unaffected entities to determine the constraint satisfiability after abstraction since their execution paths should remain unchanged during model abstraction. Hence, NPR compares states in  $P$  with these in  $CEX^\phi$ . If there are different values of any unaffected attributes, the constraint is unsatisfied and  $P$  is set as  $P_I$ ; (2) to identify uncontrollable  $P_X$ , NPR figures out the transition in  $P$  leading to its  $s^\neq$  and checks if it can be manipulated by the HA system. If not controllable, NPR sets  $P$  as  $P_X$ ; (3) to identify  $P_X$  that fails to eliminate  $s^\neq$ , NPR compares  $P$  with  $I_s$ : if  $I_s$  exists in  $P$ . It means the violating state  $s^\neq$  in  $CEX^\phi$  can not be eliminated by  $P$ . Hence,  $P$  is classified as  $P_X$ .

**Global Feasibility Checking.** A correct repair patch needs to be able to fix the identified vulnerability  $CEX^\phi$  without introducing new vulnerabilities in other rule interactions (*i.e.*, global feasibility). Given a locally feasible patch  $P = P_C$ , NPR further verifies its correctness in the original model  $\mathcal{M}^\phi$ . NPR updates  $\mathcal{M}^\phi$  as a fixed model  $\mathcal{M}_{P_C}^\phi$  by modifying existing transitions, adding new transitions, or removing deleted transitions according to transitions in  $P_C$ . Then, NPR performs model checking  $\mathcal{M}_{P_C}^\phi \models \phi$ . If no counterexample,  $P_C$  is globally feasible and NPR will terminate to output  $P_C$ ; otherwise, NPR performs iterative model refinement to optimize the abstract model until a globally feasible patch is generated.

### 5.4 TAP Rule Repair

Unlike traditional software, repairing TAP rules may change their meanings and the effect the user desires. Hence, inspired by [21, 47], to ensure that the patch can effectively repair vulnerabilities and is user-centric, it must be (1) *accommodating*: original behaviors that are verified to be globally safe should be preserved; (2) *safety-compliant*: the patched TAP system should be globally correct; (3) *valid*: patches should be deployable. Specifically, to be accommodating, TAPFixer retains rules without vulnerabilities instead of modifying them since abstraction on them does not eliminate the violating state and fails to pass the local feasibility check. For defective rules, TAPFixer retains them and repairs them by complementation (both for existing and new rules) rather than discarding them. Since TAPFixer applies a reasoning-guided strategy in model refinement to ensure that the constructed patch is globally feasible,  $P$  is *safety-compliant*. Besides, the repair process is user-centric since predefined correctness properties match user preferences and patch results are reported to users to confirm. To be *valid*, TAPFixer ensures that  $P$  follows rule syntax and physical constraints (see §5.3).

Specifically, once a global feasible patch  $P$  is identified,

TAPFixer uses  $P$  to patch the original rule set  $R^\Phi$  to repair the violation  $CEX^\Phi$ . It first identifies these transitions  $\{\phi_j\}_0^K$  ( $K$  is the number of new transitions) in  $P$  different from  $\mathcal{M}^\Phi$  to construct as the repair policy  $\hat{\Sigma}$ . Following the path of  $P$ , TAPFixer identifies different elements of each transition  $\phi_j$  in  $\hat{\Sigma}$  to generate updating operations to  $R^\Phi$  as follows: if  $\phi_j$  has a new trigger predicate or assignment (*i.e.*, a new action) than existing transitions, it translates  $\phi_j$  into a new TAP rule and sets into  $R^\Phi$ ; if  $\phi_j$  has a partial different status predicate, it updates the corresponding existing rule by modifying its condition or action according to  $\phi_j$ ; if  $P$  skips some existing transitions, it removes corresponding TAP rules. Take Fig. 1(a) for example, the generated  $\{\phi_j\}_0^K$  contains two predicates *takeDelayOn(Fan)* for  $r_1$  and  $r_2$ . Hence, NPR generates a condition (“CO<sub>2</sub> is low”) to both update the activation condition of  $a_\square$  (“turn off the fan”) in  $r_1$  and  $r_2$ .

Finally, these repaired rules are required to be converted into the form of corresponding HA app programs. However, there is no unified programming language used in different HA platforms, *e.g.*, natural language in IFTTT, Groovy in SmartThings, and GUI in MI Home. Hence, TAPFixer currently only focuses on generating repair patches in the form of rule syntax (*i.e.*, rule patches) and outputting found counterexamples with corresponding properties and repair patches to users. We leave program repairs as manual work which may require users to follow the patch to reprogram HA apps using the language provided by the platform. This process is straightforward since HA apps are relatively small and simple.

## 6 Implementation and Evaluation

### 6.1 Implementation

To evaluate TAPFixer, we implement a prototype system of TAPFixer with 4216 lines of code in Python, including: (1) *Modeling Information Extractor*, (2) *Model Builder*, (3) *Model Validator*, (4) *CEX Analyzer*, and (5) *Model Abstractor*. We use a mature symbolic model checker nuXmv [8] as the model-checking engine in TAPFixer so that it can perform unbounded model checking with given properties in the LTL formula to verify rule interaction vulnerabilities and bounded model checking (BMC) for repair patch reasoning to repair them. Besides, we develop 53 relevant correctness properties (shown in Appendix Table 15) based on vulnerability patterns to detect and fix the vulnerability. The source codes of our TAPFixer is available at: <https://github.com/qluTr5th/TAPFixer>.

### 6.2 Experiment Setup

The TAPFixer prototype is evaluated from four aspects: *case study*, *vulnerability verification and repair of real-world HA apps*, *user study*, and *performance benchmarks*. The case study is used to validate TAPFixer’s accuracy for detect-




ing and fixing vulnerabilities in comparison to existing approaches. A set of IoT market apps is then applied to evaluate whether TAPFixer can identify and repair property violations in practice and analyze the repair success rate (RSR) achieved by TAPFixer. We next conduct a user study to investigate potential vulnerabilities introduced during rule configuration by users with varying levels of knowledge about TAP rules, so as to verify the bug-fixing capability of TAPFixer in practical IoT scenarios. Finally, we analyze the time overhead of TAPFixer in vulnerability detection and repair generation.











































































































We evaluate TAPFixer on two most popular HA platforms SmartThings and IFTTT which have been widely studied in the existing literature. We totally obtain 1177 TAP rules from two aspects: 49 groovy apps with a total of 149 rules from a public SmartThings benchmark IoTMAL [6] and 1028 IFTTT applets from the IFTTT dataset used in [36]. Device capabilities provided in [1] are also collected to formulate TAP rules. TAPFixer is evaluated on an HP desktop with 2.30GHz Intel Core i7-11800H and 16GB memory.

### 6.3 Case Study of Accuracy

Our case study is conducted from two aspects: detection and repair. First, to the detection accuracy of TAPFixer, we use a detection benchmark SmartHomeBench [10] (extended based on IoTMAL) to compare TAPFixer with several existing methods, including SOATERIA [19], SAFECHAIN [31], IOTCOM [12], and TAPInspector [46]. Note that SOATERIA and TAPInspector are closed-sourced, but we can use their report results provided in [12, 19, 46]. Hence, we run SAFECHAIN, IOTCOM, and TAPFixer on SmartHomeBench and show the accuracy comparison results in Table 2. Individual apps *ID-1-9* and app groups *Gp-1-3* contain violations only related to rule logic. *Gp-4-6* contain violations related to physical channels. *N1-3* and *Gp-N4-5* provides violation cases corresponding to expanded vulnerabilities. Besides, we further develop a SmartApp *N-6* containing a rule “**IF** smoke detected, **THEN** open water valve for 10min.” with the property P.42 as a benchmark case, which can be used to evaluate the detection capability of nondeterminacy violations. The time threshold for evaluating the water valve to eliminate fires is determined by the fire intensity. The threshold in *N-6* is preset to fixed 10 min, but it may require to 15 min to eliminate the fire due to the greater fire intensity, which causes the water valve to close after 10 min and violates P.42.

We can find that SOATERIA and SAFECHAIN fail to identify expanded vulnerabilities due to limited modeling capabilities for physical and latency factors. IOTCOM supports the analysis of physical channels, so it can detect more violations. TAPInspector supports most cases, but not the nondeterminacy in *N-6*. TAPFixer introduces more comprehensive physical and latency modeling, thus identifying all violations successfully. The fundamental reason is that existing detection methods do not support the modeling of many

Table 2: Accuracy comparison of the vulnerability detection. We use , , and  to denote true positive, false positive, and false negative, respectively.

Benchmark	SOATERIA*	SAFECHAIN	IOTCOM	TAPInspector*	TAPFixer
ID-1					
ID-2					
ID-3					
ID-4	 				
ID-5					
ID-6					
ID-7					
ID-8					
ID-9					
N-1					
N-2					
N-3					
N-6					
Gp-1					
Gp-2					
Gp-3					
Gp-4					
Gp-5					
Gp-6					
Gp-N4					
Gp-N5					

\* results obtained from [12, 19, 46]

physical features (we discuss them in Table 9), leading to a low detection capability.





























Existing static vulnerability repair methods focus on simple rule conflicts and lack attention to expanded complex vulnerabilities. Hence, we do not use SmartHomeBench as the repair benchmark since it contains many simple violations. In turn, we construct a set of flawed rules containing complex vulnerabilities as the benchmark to assess differences in the repair capability between TAPFixer and existing methods. For conciseness, we give the text-based description of these flawed rules in Table 3. *Group 1* contains three rules shown in Fig. 1(b) and involves a vulnerability **V4**. In *Group 2*, the delay used to turn off the AC is mistakenly configured to turn it on. It leads to the vulnerability **V5** that two rules interact in latency disorder and perform conflicting actions, which causes the AC to not turn off automatically after it is on. There is a vulnerability **V6** in *Group 3* that if the user leaves less than ten minutes after returning home, the action of the first rule can override the power-off effect of the second rule. *Group 4* is the extended scenario of Fig. 1(a), where the vulnerability **V7** also occurs between the first and second rule. *Group 5* contains four rules for temperature regulation. If the temperature is below 18°C when the user presents, the vulnerability **V8** occurs since the third rule may be triggered before the second rule blocks its activation. We also consider physical- and latency-related initialization scenarios (*N/A 1-2*) that violate the event-based property P.28 and state-based property P.21 (in Appendix Table 15), respectively.

We summarize the accuracy of TAPFixer and other approaches in Table 4. For these groups, Liang et al. [34], MenShen [17], and AutoTap [47] fail or generate faulty patches.

Table 3: Designed flawed rule groups.

Benchmark	Vulnerability	Rule Description
<i>Group 1</i>	<b>V4</b>	<b>IF</b> temperature < 16°C, <b>THEN</b> turn on heater. <b>IF</b> temperature > 24°C, <b>THEN</b> turn off heater. <b>IF</b> temperature > 20°C, <b>THEN</b> open window.
<i>Group 2</i>	<b>V5</b>	<b>IF</b> after user present 10 min, <b>THEN</b> turn on AC. <b>IF</b> user is present, <b>THEN</b> turn off AC.
<i>Group 3</i>	<b>V6</b>	<b>IF</b> after user present 10 min, <b>THEN</b> turn on blanket. <b>IF</b> user is not present, <b>THEN</b> turn off blanket.
<i>Group 4</i>	<b>V7</b>	<b>IF</b> CO <sub>2</sub> > 1000 ppm, <b>THEN</b> turn on fan for 15min. <b>IF</b> air humidity > 80%, <b>THEN</b> turn on fan for 10min. <b>IF</b> user is present, <b>THEN</b> turn on fan for 5min.
<i>Group 5</i>	<b>V8</b>	<b>IF</b> temperature < 18°C <b>WHILE</b> user is present, <b>THEN</b> turn on heater. <b>IF</b> temperature < 25°C for 20 min <b>WHILE</b> user is present and window is closed, <b>THEN</b> turn off heater. <b>IF</b> temperature > 27°C, <b>THEN</b> open window. <b>IF</b> temperature < 16°C, <b>THEN</b> close window.
<i>N/A 1</i>	P.28 violation	The violation of event-based P.28 with no rules.
<i>N/A 2</i>	P.21 violation	The violation of state-based P.21 with no rules.

Table 4: Repair accuracy comparison of expanded vulnerabilities.

Benchmark	Liang et al. [34]	MenShen [17]	AutoTap [47]	TAPFixer
<i>Group 1</i>				
<i>Group 2</i>				
<i>Group 3</i>				
<i>Group 4</i>				
<i>Group 5</i>				
<i>N/A 1</i>				
<i>N/A 2</i>				

<sup>†</sup> Correctly fixed partial vulnerable rule interactions, but did not fix the rest.

<sup>‡</sup> Correctly fixed partial vulnerable rule interactions, but incorrectly fixed the rest.

Liang et al. only partially fix the vulnerability in *Group 1* (i.e., set a condition “the user is at home” in the first rule) since it only focuses on current trigger conditions, but ignores the one derived from existing actions or new rules. For instance, the fix for the vulnerability in *Group 1* and *N/A 1* requires creating new rules, while the fix for *Group 4* requires the action supplement. Liang et al. consider such fixes to be useless and are therefore classified as false negatives. For groups where they generate false positives (*Group 2-3*), they remove the delay or change the temperature condition expected by the user, which violates the user’s intent. MenShen fails to fix all vulnerabilities since it lacks the formulation of rule semantics. AutoTap fixes vulnerabilities in *Group 1, 5* and *N/A 1-2* to vary degrees with false positives detected in *Group 1-4*. The reason is that it merely focuses on the solution space derived from new rules but neglects the one associated with existing rules. Thus, AutoTap does not prevent the expanded vulnerability from occurring but only reacts to it after it has occurred. For example, AutoTap will generate a patch for *Group 4* to turn on the fan immediately after it is interrupted and still leave the fan to be interrupted when the CO<sub>2</sub> level is high.

In comparison with existing approaches, TAPFixer can fix expanded vulnerabilities in all groups, including configuring from scratch. It is attributed to the accurate patch reasoned



Table 5: Summary of detection and repair results for G1-G7 with 110 rule groups, each of which contains 15-30 TAP rules.

Application scenarios	Fixed violations	Unfixable violations	Safe cases	Generated patches	RSR↑
<b>G1</b> (2 properties)	179	35	6	364	83.64%
<b>G2</b> (21 properties)	1675	277	358	2228	85.81%
<b>G3</b> (6 properties)	459	201	0	902	69.55%
<b>G4</b> (8 properties)	687	59	134	1145	92.09%
<b>G5</b> (9 properties)	870	68	52	1288	92.75%
<b>G6</b> (3 properties)	272	58	0	491	82.42%
<b>G7</b> (4 properties)	402	2	36	419	99.50%
Total	4544	700	586	6837	86.65%

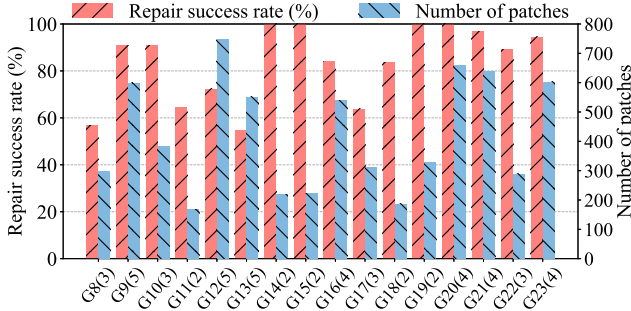


Figure 5: Repair success rate and the number of generated patches for G8-G23 with 110 rule groups. The numbers in parentheses represent the number of properties contained in the property set.

by our proposed negated-property reasoning algorithm. We have also tested the effectiveness of these fixes by deploying these TAP rules on HomeAssistant. In summary, TAPFixer achieves 100% accuracy for this benchmark.

## 6.4 Market App Study of Violation Repair

To evaluate the generality of TAPFixer to repair market apps, we mainly focus on IFTTT and SmartThings. Based on 9 modeled physical channels, we build a set of sensor groups that detect changes in the environment, including (1) temperature sensor; (2) light sensor; (3) presence sensor; (4) smoke sensor; (5) humidity sensor; (6) carbon monoxide sensor; (7) carbon dioxide sensor; (8) sound sensor; (9) weather sensor. By summarizing the general functional scenario of the actuator based on safety properties, we build a set of actuator groups, including (1) light; (2) door control, garage door control, and security camera; (3) air conditioner, heater, and electric blanket; (4) fan, window, sprinkler system, water valve, gas valve, natural gas hot water heater, and alarm; (5) smart plug, oven, and coffee maker. We then randomly select 15-30 rules from the total rule set to form a rule group, which is associated with four categories of sensor groups and two categories of actuator groups at least. In total, we obtained 110 rule groups with an average of 21 rules per group.

With these generated rule groups, we verify and fix rule interaction vulnerabilities based on correctness properties from two aspects: safety and prioritization of application scenarios. The former ensures that all modeling devices operate in a safe status (e.g., all appliances are turned off) in the same appli-

cation scenarios (e.g., when no one is home), while the latter ensures that the same actuator can process safety requests with higher priority if there are conflicting safety properties in different application scenarios. We categorize properties into 7 groups (**G1-G7**) by determining whether the application scenario is the same based on the pre-proposition of properties. We also categorize them into 16 groups (**G8-G23**) based on the prioritized status of the same actuator in the post-proposition of properties. The specific categories (**G1-G23**) for each property are shown in Appendix Table 15.

Following existing evaluation methods [12, 18, 19, 28], we evaluate the accuracy of TAPFixer by manually checking found property violations and patches. The manual checking process is straightforward to perform given the output of TAPFixer since the number of rules is relatively small. The results of **G1-G7** are shown in Table 5. TAPFixer correctly fixes 4544 out of 5244 property violations, achieving an RSR 86.65%. The results of **G8-G23** are shown in Fig. 5. TAPFixer correctly fixes 4460 out of 5335 property violations, achieving an RSR of 83.60%. For all properties in **G1-G23**, TAPFixer successfully repairs 9004 out of 10579 property violations, achieving a total RSR of 85.11%.

TAPFixer fails to fix 1575 property violations in total. We manually analyze these failures and find that there are three major reasons: (1) *Predicate solving errors due to SMT*. Some local vulnerabilities can be fixed only by modifying some of all predicates. However, the SMT solver in nuXmv may unexpectedly specify the satisfiability of predicates that are irrelevant for violation repair, which can cause TAPFixer to fail to fix vulnerabilities correctly; (2) *Limitations on the semantic abstraction of new transitions*. In model semantic abstraction, we currently only consider interpolating numerical variables as trigger predicates, not status predicates. It can invalidate scenarios that require numerical status predicates to fix vulnerabilities; (3) *Too large number of predicates*. We have limited the number of predicates to avoid state explosion. However, the number of predicates in some scenarios can still exceed the upper limit and cause TAPFixer to fail.

We further conduct a comparison with the SOTA AutoTAP [47]. Since AutoTAP cannot automatically extract rules from IoT apps, we randomly select 25 rule groups and manually write them using AutoTAP’s interface to run AutoTAP. AutoTAP only supports modeling limited rule features and does not support modeling of many device capabilities (e.g., garage door, heater, fan, sprinkler, etc.) and physical attributes (e.g., CO, CO<sub>2</sub>, and humidity, etc), which may result in a low RSR due to modeling failures. Hence, to guarantee fairness, we filter out these correctness properties containing device capabilities (e.g., fan and sprinkler in **G6**) and physical attributes (e.g., CO and CO<sub>2</sub> in **G5**) that are not supported by AutoTAP, and finally obtain 22 correctness properties. Besides, we introduce a metric *Modeling Success Rate* (MSR) to assess the integrity of rule modeling and categorize *Repair Failure Rate* (RFR) into RFR-MF and RFR-LIMIT to eval-



Table 6: Comparison between AutoTap and TAPFixer.

Evaluation target	AutoTap	TAPFixer
MSR $\uparrow$	54.23%	100%
RSR $\uparrow$ of G1 (1 property)	20%	44%
RSR $\uparrow$ of G2 (14 properties)	51.43%	74.59%
RSR $\uparrow$ of G3 (1 property)	8%	92%
RSR $\uparrow$ of G4 (4 properties)	44%	94.32%
RSR $\uparrow$ of G7 (2 properties)	38%	91.49%
RFR-MF/RFR-LIMIT $\downarrow$	23.99%/24.57%	0%/20.93%

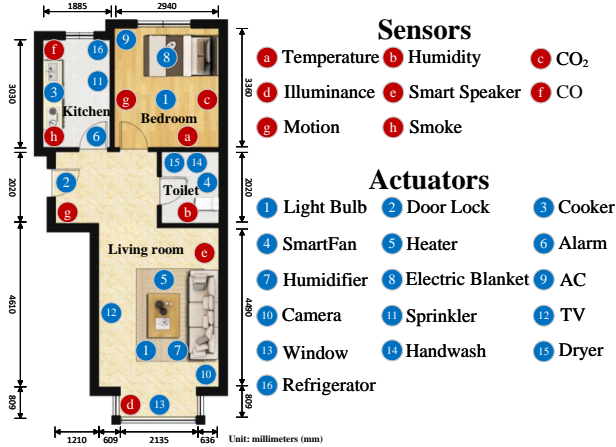


Figure 6: Device layout in the smart home.

uate reasons for repair failures caused by modeling failures and repair algorithm limitations, respectively.

We check MSR, RSR, RFR-MF, and RFR-LIMIT achieved by AutoTAP and TAPFixer and show the comparison results in Table 6. In general, a lower MSR indicates that the constructed model contains fewer rule interactions and thus, is more likely to be reported as a successful fixed case and the RSR should be higher. However, the MSR and RSR of AutoTAP are both lower than TAPFixer. We can find that AutoTAP and TAPFixer totally achieve an RFR of 54.55% and 20.93%, respectively. Among these repair failures, after eliminating these failures caused by modeling failures, although AutoTAP can model rules much less than TAPFixer, it still has a higher RFR-LIMIT (24.57%). The fundamental reason is that the repair capability of AutoTAP is lower than TAPFixer and may repair the rule group incorrectly. For example, in G2 with several rule groups violating the properties P.2, if the target light is controlled by a rule involving a numeric attribute (e.g., temperature), AutoTAP will not generate any repair patches. This is because AutoTAP directly filters out those actions activated by numeric states that could lead to other property violations, thereby avoiding additional analysis costs. If not, AutoTAP can generate a patch rule (“If the light is off while the user is at home, then turn the light on”), which, however, cannot fix the violation. Our TAPFixer can generate a correct patch rule “If the user is not at home, then turn the light off” both in these two cases since it uses an iterative abstraction and refinement process to correct and verify the feasibility of all generated possible patches at a low analysis cost.

Table 7: Number of identified and fixed vulnerabilities in 129 rules.

	V1	V2	V3	V4	V5	V6	V7	V8
# found violations	5	6	9	23	0	4	5	3
# fixed violations	3	6	8	23	0	4	5	3
RSR	60%	100%	89.9%	100%	N/A	100%	100%	100%

## 6.5 User Study

To evaluate users’ acceptance of patches generated by TAPFixer, we conduct a user study on the online questionnaire platform. We use the conceptual HA scenario in Fig. 6 and invite participants to set up TAP rules in the scenario. Since the scenario is fixed and our predefined properties in Table 15 can almost cover all possible cases, we also require participants to pick up properties in Table 15 for violation detection and repair. We received 23 questionnaires and the distributions of our participants are as follows: 47.8% are male and 52.2% are female, 87.0% are 20-35 years old, and 13% are over 35 years old, 52.2% indicate that they have professional experience in computer science, 78.3% own smart devices and 73.9% are familiar with home automation rules.

Given questionnaires specified in natural language, we manually unify different descriptions having the same meanings into the same term, so that we can map entities in users’ rule descriptions into TAP rule syntax accurately and extract TAP rules from these descriptions. We obtain 394 TAP rules with an average of 17 rules per questionnaire taking 28 minutes. We then run TAPFixer to detect and repair interaction vulnerabilities in collected rules. We randomly select found property violations and manually analyze their vulnerability patterns that cause property violations as shown in Table 7. Overall, we analyze 129 flawed rules and TAPFixer achieves an RSR of 94.5% in both basic and expanded vulnerability patterns.

Our findings from this study are as follows: (1) the number of V4 is noticeably more than other identified vulnerabilities. Participants tend to set switches for temperature-related actuators to two separate TAP rules. This results in a vulnerability of V4 among the window, AC, heater, and electric blanket; (2) although the majority of participants have used or learned about home automation, they rarely use conditions of TAP rules and set the delay for device executions, which affect the incidence of condition interference and latency-related issues to some extent; (3) no V5 vulnerability is found since it mainly originates from platform delays and users’ careless misconfiguration, which do not occur in the study.

To evaluate the quality of TAPFixer’s results, we randomly selected 9 property violations with their corresponding TAP rules, original rule descriptions, and generated rule patches to construct a questionnaire to feedback to 23 participants. The average completion time of this questionnaire is 7.3 minutes. We received a total of 103 strong agreements, 92 agreements, and 12 disagreements (3 in V1, 1 in V2, 2 in V3, 1 in V4, 1 in V7, and 4 in V8). The reasons for disagreements can be classified into two types: 1) negating violations; 2) proposing other patches, some of which we found are not able to fix the violation or are similar to patches. Hence, we further ex-

plained the results in detail to participants who disagreed and received 10 agreements again. So, TAPFixer achieved a 99.0% satisfaction rate. The remaining disagreeing participants suggested adding restrictions on rules' lifespans in the patch (e.g., restricting  $r_4$  in Fig. 1(b) to only run in the summer), which is feasible but not supported by TAPFixer yet.

## 6.6 Performance Analysis

Since the complexity of TAPFixer depends on input models, properties, and abstraction and refinement strategies, it is hard to directly quantify. Hence, we evaluate the performance of the overall process in TAPFixer, including rule interaction modeling, vulnerability detection, and repair patch generation. To early terminate the oversized predicate exploration, we set the *ROUND\_LIMIT* and *ITER\_LIMIT* in NPR to 15 and 50, respectively. Within these two iterations, we record the execution time from two perspectives: the benchmark dataset in §6.3 and the realistic market dataset in §6.4.

We first record TAPFixer's analysis time of each test case. To form the 21-rule test case, for each benchmark dataset (*Group 1-5*), we combine rules presented in Table 3 with randomly selected ones from the extracted rule set. TAPFixer does not complement initialization scenarios without rules (*N/A 1-2*). Fig. 7 shows the time for verifying and repairing vulnerabilities in all 7 test cases. For 21-rule test cases (*Group 1-5*), the longest, shortest, average time takes 267.3 seconds in *Group 5*, 93.8 seconds in *Group 3*, and 161.7 seconds respectively. The time of the initialization scenario *N/A 1* and *N/A 2* takes 241 and 215 ms respectively.

We then record TAPFixer's total time of evaluations on market apps in Table 8. Although the verification and repair time of *G1-G7* takes a total of one hour longer than that of *G8-G23* which takes around 6 hours, their average time to create a property violation patch takes about 3 seconds with an overall average of 3.51 seconds. According to the investigation on the efficiency of manual repair conducted by a previous study [34], many users give up debugging a property violation after 45 seconds. Hence, TAPFixer achieves over 15 times improvement compared to manually repairing a property violation.

## 6.7 Discussion and Limitations

**Scalability and Extension.** The proposed techniques in NPR are not limited to any specific platform or scenario in HA systems. They can be applied in other cases where TAP rules are widely used, such as UAV control, hardware description rules, industrial equipment control, etc. In addition to these studied safety properties (*i.e.*, something bad should never happen), we aim to introduce liveness properties (*i.e.*, something good eventually happens) to address latency-related issues in more detail in the future.

**Limitation.** TAPFixer achieves high effectiveness and speed in verifying and repairing vulnerabilities in complex

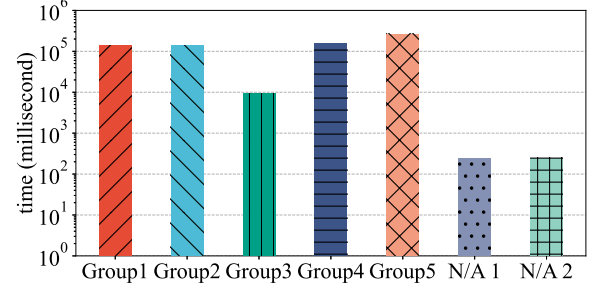


Figure 7: Verification and repair time of each 21-rule benchmark dataset and initialization scenarios.

Table 8: Average patch generation time for market apps.

Market apps	Total time (minute)	Number of generated patches	Avg. generation time per patch (second)
G1-G7 (110 rule groups)	364.070	6837	3.195
G8-G23 (110 rule groups)	431.261	6749	3.834

logical and physical spaces, but it still suffers from three limitations: First, although TAPFixer can achieve an RSR of 86.65% overall, three vulnerability cases cannot be fixed: (1) to address predicate-solving errors, we find that such failures can be avoided by running NPR again or setting the abstraction target to a single rule instead of the rule collection; (2) to address limitations during the semantics abstraction of new rules (§5.2), future work can extend TAPFixer with numerical condition abstraction to address this limitation; (3) possible solutions to avoid the predicate scale timeout are to increase *ITER\_LIMIT*, *ROUND\_LIMIT* and further optimize the predicate scale. Second, TAPFixer relies on our manual qualitative analysis of device effects on physical channels to enhance the comprehensiveness of its modeling capabilities. However, the accuracy of effect models is hard to guarantee as the home environment changes, which limits the performance of TAPFixer. But, fortunately, the home environment will not change frequently once devices are deployed in. In the future, we can introduce an online miner to regularly update effect models, which collects sensor data over a long period and uses SVM to mine the qualitative relationships between channels and device activities, similar to [16]. Third, TAPFixer currently can only provide repair patches. In the future, more reverse engineering techniques should be exploited to generate program patches of rule patches confirmed by users for specific HA platforms. Take SmartThings for example, we can first map the patch into the corresponding inter-procedural control flow graph extracted from apps during rule modeling (see Appendix B) and then transform it into the form of the Groovy abstract syntax tree which can be easily transformed into Groovy codes using Groovy AST Transformation [2].

## 7 Related work

With the prevalence of home automation, safety issues in the TAP-based HA system have gained much attention. Table 9 and Table 10 respectively illustrate the comparison of our research with related detection and repair work over various

Table 9: Comparison between TAPFixer and related detection works.

Related work	Latency	Tardy attribute	Implicit effect	Joint physical effect	Nondeterminacy
IOTCOM [26]	✓	✗	✓	✗	✗
SAFECHAIN [31]	✗	✗	✗	✗	✗
IRuler [45]	✓	✗	✗	✗	✗
IoTGUARD [18]	✗	✗	✗	✗	✗
SOATERIA [19]	✗	✗	✗	✗	✗
TAPInspector [46]	✓	✓	✓	✗	✗
IOTSAN [38]	✗	✗	✗	✗	✗
HOMEGUARD [22]	✓	✗	✗	✗	✗
Jia et al. [32]	✗	✗	✗	✗	✗
IoTSEER [39]	✓	✓	✓	✓	✗
Chi et al. [20]	✓	✗	✗	✗	✗
TAPFixer	✓	✓	✓	✓	✓

Table 10: Comparison between TAPFixer and related repair works.  $phy_1$ ,  $phy_2$ ,  $phy_3$ ,  $phy_4$ , and  $phy_5$  denote tardy attribute, channel-based interaction, implicit physical effect, joint physical effect, non-determinacy, respectively.

Related work	Latency-related			Physical-related				
	$l_1$	$l_2$	$l_3$	$phy_1$	$phy_2$	$phy_3$	$phy_4$	$phy_5$
ESOs [41]	✓	✓	✓	✓	✓	✗	✗	✓
Bastys et al. [14]	✓	✓	✓	✓	✗	✗	✗	✓
He et al. [30]	✓	✗	✗	✓	✗	✗	✗	✓
SmartAuth [43]	✓	✓	✓	✓	✓	✗	✗	✓
ContextIoT [33]	✓	✓	✓	✓	✓	✗	✗	✓
IoTSAFE [28]	✓	✓	✗	✓	✓	✓	✓	✓
IoT-MEDIATOR [21]	✓	✗	✗	✓	✓	✗	✗	✓
Liang et al. [34]	✓	✗	✓	✓	✗	✗	✗	✗
MenShen [17]	✓	✗	✓	✓	✗	✗	✗	✗
AutoTap [47]	✓	✗	✗	✗	✗	✗	✗	✗
TAPFixer	✓	✓	✓	✓	✓	✓	✓	✓

features. We discuss this comparison as follows:

**Rule Vulnerability Detection.** Detecting vulnerabilities in more comprehensive and detailed environments has been widely studied for the past years. SOTERIA [19] proposes a detection method based on LTL model checking. IOTSAN [38] analyses sequentiality and concurrency of rule executions. IOTA [37] proposes the calculus for the domain of home automation to secure rule interactions. Balliu et al. [13] proposes a semantic framework capturing the essence of cross-app interactions and Friendly Fire [?] further optimises it. SAFECHAIN [31] detects hidden attack chains exploiting the combination of rules based on model checking. IOTCOM [12] focuses on vulnerabilities between logical and physical interactions. HOMEGUARD [22] uses SMT techniques to detect cross-app conflicts. IRuler [45] considers the uncertainty of smart devices. Jia et al. [32] handle scenarios about device management channels. Chi et al. [20] studies vulnerabilities caused by delay-based automation interference attacks. TAPInspector [46] implements a comprehensive analysis of rule interactions by introducing latency and connection-based features. IoTSEER [39] alleviates approximation problems through dynamic analysis. TAPFixer follows these advanced methods to model and detect rules. But, we introduce more features to improve the accuracy and integrity of static modeling shown in Table 9 so that NPR can statically generate formal-soundness patches using formal verification.

**Dynamic Rule Enforcement Control.** Existing works de-

velop control policies based on specific concerns in the HA system and dynamically prevent risks. ContextIoT [33] uses the data and control flows of smart apps to build access contexts. SmartAuth [43] investigates authorization mechanisms with different behavioral security levels. Bastys et al. [14] design a short-term and a long-term access control mechanism based on the information flow. He et al. [30] point out the key factors that constitute the complex access control scenario. ESOs [41] studies cross-layer access control and provides corresponding permissions. IoTSAFE [28] identifies real-time physical interactions combined with dynamic and static methods to predict and avoid hazard scenarios. IoT-MEDIATOR [21] provides a more fine-grained access control framework through threat-tailored handling. The comparison between TAPFixer and advanced dynamic-based repair works is shown in Table 10. Compared to these methods, TAPFixer can eliminate the root cause of vulnerabilities.

**Static Fixing Towards Rule Semantic.** Researches on static fixing include Liang et al. [34], MenShen [17], AutoTap [47], and also our work TAPFixer. AutoTap [47] proposes an automaton-based automatic method to fix vulnerabilities, which identifies the violated bridge edge and generates fixes based on it. Liang et al. [34] and MenShen [17] develop semi-automatic formal methods to fix vulnerabilities. They parameterize the syntax of existing rules and solve for specific values that can eliminate the violation. However, none of them perform detailed physical and latency analysis, missing many practical features and vulnerabilities. Additionally, their fixing algorithms are limited in addressing expanded vulnerabilities associated with complex physical and latency issues. The comparison between TAPFixer and advanced static-based repair works is also shown in Table 10. Compared to these methods, TAPFixer can effectively repair vulnerabilities in complex physical environments, especially when dealing with expanded vulnerabilities.

## 8 Conclusion

In this work, we design a novel automatic vulnerability detection and repair framework, TAPFixer, for TAP-based home automation systems. With a comprehensive analysis of existing rule interaction vulnerabilities, TAPFixer can model TAP rules with more practical latency and physical features to capture the accurate rule execution behaviors both in the logical and physical space and identify more interaction vulnerabilities. We propose a novel negated-property reasoning algorithm for TAPFixer so that it is able to accurately generate valid patches for eliminating vulnerabilities both in the logical and physical space. We conduct numerical evaluations of TAPFixer from aspects of accuracy analysis, repair capabilities of market apps, real user study, and execution performance. The results of our evaluation show that TAPFixer can correctly fix different rule interaction vulnerabilities with an excellent performance overhead.

## Acknowledgment

We thank our shepherd and the anonymous reviewers for their valuable feedback. This work was partially supported by the National Natural Science Foundation of China under Grant 62202387 and the Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515110279.

## References

- [1] Device Capabilities Reference. <https://developer.smarthings.com>. 2023.
- [2] Groovy AST Transformation. <https://docs.groovy-lang.org/latest/html/gapi/org/codehaus/groovy/transform/ASTTransformation.html>. 2024.
- [3] HomeAssistant. <https://github.com/home-assistant>. 2023.
- [4] HomeKit. <https://www.apple.com/ios/home>. 2023.
- [5] IFTTT. <https://ifttt.com>. 2023.
- [6] IoTBench test suite. <https://github.com/IoTBench/IoTBench-test-suite>. 2023.
- [7] MI Home. <http://home.mi.com>. 2023.
- [8] nuXmv. <https://nuxmv.fbk.eu>. 2023.
- [9] Selenium automates browsers. That's it! <https://www.selenium.dev/>. 2023.
- [10] SmartHomeBench. <https://github.com/EboYu/SmartHomeBench>. 2023.
- [11] SmartThings. <https://www.smarthings.com>. 2023.
- [12] Mohammad Alhanahnah, Clay Stevens, and Hamid Bagheri. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 272–285, 2020.
- [13] Musard Balliu, Massimo Merro, and Michele Pasqua. Securing cross-app interactions in iot platforms. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, Jun 2019.
- [14] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If this then what? controlling flows in iot apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1102–1119, 2018.
- [15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic Model Checking without BDDs*, page 193–207. Jan 1999.
- [16] Simon Birnbach, Simon Eberz, and Ivan Martinovic. Peeves: Physical event verification in smart homes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1455–1467, 2019.
- [17] Lei Bu, Wen Xiong, Chieh-Jan Mike Liang, Shi Han, Dongmei Zhang, Shan Lin, and Xuandong Li. Systematically ensuring the confidence of real-time home automation iot systems. *ACM Transactions on Cyber-Physical Systems*, 2(3):1–23, 2018.
- [18] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [19] Z.Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *USENIX Annual Technical Conference (ATC)*, May 2018.
- [20] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Delay wreaks havoc on your smart home: Delay-based automation interference attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 285–302. IEEE, 2022.
- [21] Haotian Chi, Qiang Zeng, and Xiaojiang Du. Detecting and handling {IoT} interaction threats in {Multi-Platform}{Multi-Control-Channel} smart homes. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1559–1576, 2023.
- [22] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 411–423, 2020.
- [23] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.
- [24] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. Empowering end users in debugging trigger-action rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, May 2019.
- [25] SmartThings Developers. Device Capabilities Reference. <https://docs.smarthings.com/en/latest/capabilities-reference.html>. 2023-20.



- [26] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2008.
- [27] Wenbo Ding and Hongxin Hu. On the safety of iot device physical interaction control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Oct 2018.
- [28] Wenbo Ding, Hongxin Hu, and Long Cheng. Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery. In *the 28th Network and Distributed System Security Symposium (NDSS)*, 2021.
- [29] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Hawatcher: Semantics-aware anomaly detection for appified smart homes. *USENIX Security Symposium*, Aug 2021.
- [30] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (iot). In *USENIX Security Symposium*, pages 255–272, 2018.
- [31] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. Safechain: Securing trigger-action programming from attack chains. *IEEE Transactions on Information Forensics and Security (TIFS)*, 14(10):2607–2622, 2019.
- [32] 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 *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1289–1305, 2021.
- [33] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. Contextlot: Towards providing contextual integrity to appified iot platforms. In *Network and Distributed System Security Symposium (NDSS)*, volume 2, pages 2–2. San Diego, 2017.
- [34] 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 *Proceedings of the 3rd ACM International Conference on Systems for Energy-efficient Built Environments*, pages 133–142, 2016.
- [35] 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 *IEEE Symposium on Security and Privacy (SP)*, pages 482–499, 2020.
- [36] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference (IMC)*, pages 398–404, 2017.
- [37] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeanin, Cole Schlesinger, and Manu Sridharan. Iota: a calculus for internet of things automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Oct 2017.
- [38] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V Krishnamurthy, Edward JM Colbert, and Patrick McDaniel. Iotsan: Fortifying the safety of iot systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 191–203, 2018.
- [39] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. Discovering iot physical channel vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2415–2428, 2022.
- [40] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Sep 1977.
- [41] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational access control in the internet of things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1056–1073, 2018.
- [42] Amit Kumar Sikder, Hidayet Aksu, and A Selcuk Uluagac. 6thsense: A context-aware sensor-based attack detector for smart devices. In *26th USENIX Security Symposium*, pages 397–414, 2017.
- [43] Yuan Tian, Nan Zhang, Yue-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *USENIX Security Symposium*, volume 5, pages 8–2, 2017.
- [44] Yinxin Wan, Kuai Xu, Feng Wang, and Guoliang Xue. Iotmosaic: Inferring user activities from iot network traffic in smart homes. In *IEEE INFOCOM*, pages 370–379. IEEE, 2022.

- [45] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1439–1453, 2019.
- [46] Yinbo Yu and Jiajia Liu. Tapinspector: Safety and liveness verification of concurrent trigger-action IoT systems. *IEEE Transactions on Information Forensics and Security*, 17:3773–3788, 2022.
- [47] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenburg, Shan Lu, and Blase Ur. Autotap: Synthesizing and repairing trigger-action programs using ltl properties. In *2019 IEEE/ACM 41st international conference on software engineering (ICSE)*, pages 281–291. IEEE, 2019.

## A Rule Interaction Vulnerability Patterns

TAPFixer integrates more features in physical space and uncovers 8 patterns of rule interaction vulnerability which are summarized in Fig. 8.

**V1: Trigger-Interference Basic Pattern.**  $a_i$  of  $r_i$  and  $t_j$  of  $r_j$  share the same immediate channel attribute ( $a_i \cap t_j \cap \mathbb{A}_{immd} \neq \emptyset$ ), which can cause events generated by  $a_i$  unexpectedly triggering  $r_j$  ( $a_i \dashrightarrow t_j$ ) and put the rule execution at risks.

**V2: Condition-Interference Basic Pattern.**  $a_i$  of  $r_i$  and  $c_j$  of  $r_j$  share the same immediate channel attribute ( $a_i \cap c_j \cap \mathbb{A}_{immd} \neq \emptyset$ ), which may lead to  $a_i$  changing the satisfaction of  $c_j$  ( $a_i \Rightarrow c_j$ ) and change the rule context defectively.

**V3: Action-Interference Basic Pattern.**  $r_i$  and  $r_j$  are both immediate rules ( $r_i, r_j \in r_{immd}$ ) and have no latency ( $l_i \notin r_i \cup r_j$ ). Commands from  $a_i$  and  $a_j$  to the same device are conflicting ( $a_i \xrightarrow{\times} a_j$ ) and may override the effect after the secure interaction.

**V4: Tardy-channel-based Trigger Interference.** Similar to V1,  $a_i$  can unexpectedly trigger  $t_j$  sharing the same physical channel ( $a_i \dashrightarrow t_j$ ), but from the tardy channel ( $a_i \cap t_j \cap \mathbb{A}_{tardy} \neq \emptyset$ ). Group1 in Table 3 shows an example of V4.

**V5: Disordered Action Scheduling.**  $r_i$  and  $r_j$  are both immediate ( $r_i, r_j \in r_{immd}$ ) and have conflicting actions ( $a_i \xrightarrow{\times} a_j$ ), but include  $l_i$  compared to V3 ( $l_i \in r_i \cup r_j$ ).  $r_i$  and  $r_j$  will be triggered simultaneously ( $t_i \cap t_j \neq \emptyset$ ), but with the involvement of  $l_i$  and  $l_j$  (if it exists), the expected order of actions is disrupted. Group2 in Table 3 shows an example of V5.

**V6: Action Overriding.** This type of vulnerability is similar to V5, but  $r_i$  and  $r_j$  can be triggered separately ( $t_i \cap t_j = \emptyset$ ). The execution time  $l_i$  is longer than  $l_j$  (if it exists), causing  $a_i$  to overwrite the effect of the previous execution of  $a_j$  ( $a_i \xrightarrow{\times} a_j$ ). Group3 in Table 3 shows an example of V6.

**V7: Action Breaking.** Different from V3 where both rules are immediate,  $r_i$  and  $r_j$  contain at least one extended rule ( $r_i \in r_{ext} \vee r_j \in r_{ext}$ ). Assume  $r_j$  is the extended rule, during

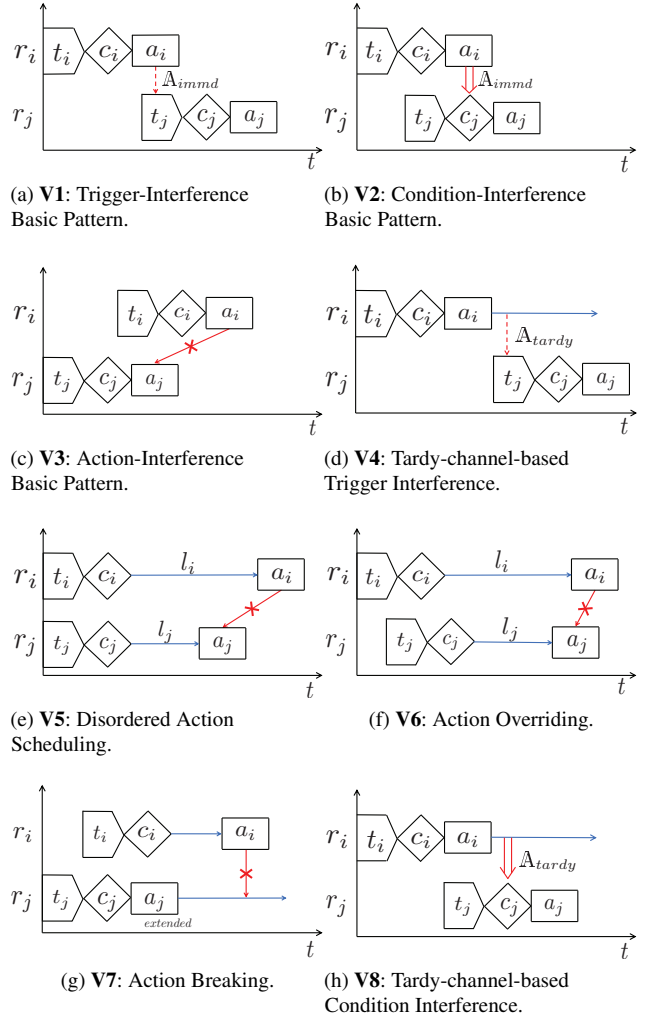


Figure 8: Rule interaction vulnerability patterns.

its execution, the state of the device in  $a_j$  will be stayed for a while.  $r_i$  with different latency preference will complete  $a_i$  before  $r_j$ , interrupting the process of extended  $a_j$  ( $a_i \xrightarrow{\times} a_j$ ). Group4 in Table 3 shows an example of V7.

**V8: Tardy-channel-based Condition Interference.** Similar to V2,  $a_i$  can change the satisfaction of  $c_j$  sharing the same physical channel ( $a_i \Rightarrow c_j$ ), but from the tardy channel ( $a_i \cap c_j \cap \mathbb{A}_{tardy} \neq \emptyset$ ). Group5 in Table 3 shows an example of V8.

## B Model Information Extraction

Depending on existing efforts [12, 46], TAPFixer first extracts inter-procedural control flow graphs (ICFGs) from HA apps and then formalizes ICFGs as TAP rules. It utilizes static program analysis for SmartThings apps written in Groovy and NLP techniques for IFTTT applets.

To extract ICFGs from SmartApps, TAPFixer implements an Abstract Syntax Tree (AST) parser upon the Groovy compiler to perform path-sensitive analyses [26] on AST

nodes. For the closed-source SmartThings API during path analysis, TAPFixer reviews the SmartThings API documentation [1] to manually model the APIs in a manner similar to [24]. With the AST parser and the handling of closed-source features, ICFGs can be extracted accurately. To convert IFTTT applets into ICFGs, TAPFixer utilizes NLP methods [27, 45] for string analysis similar to [12, 18]. It decomposes the applet text into tokens and searches for trigger, condition, and action titles from them. With extracted titles, TAPFixer constructs an ICFG with two nodes, where the input node represents the trigger and condition while the other node represents the action. Note that conditions in IFTTT can also be specified using the filter code in the form of JavaScript snippets, which requires hard manual effects [18] and is not publicly available. Hence, we do not analyze the conditions specified in them like [45, 46]. Using runtime techniques [14, 18] is a possible solution to improve the capability of TAPFixer in the future.

To construct TAP rules from ICFGs, TAPFixer first reformulates device capabilities into JSON format. It then provides a crawler, based on Selenium [9], for users to extract rule configurations not included in source codes. With the obtained information, TAPFixer extracts valid paths from ICFG and converts them into TAP rules similar to [46]. It extracts trigger, condition, and action sets from different domains based on the path location in the app (app initialization, app schedule, and event handling). TAPFixer also extracts device areas, rule semantics, and rule configurations from extracted rule information.

### C Quantification of Physical Channel Interactions

Based on physical interactions studied in existing literature [16, 20, 22, 27, 28, 39, 45, 46], we totally model 9 physical channels as follows: temperature, illuminance, motion, smoke, humidity, CO, CO<sub>2</sub>, sound, and weather status. We consider motion and weather status are not affected by actuator executions, while the rest of the physical channels can be affected by actuators. To quantify physical effects, we collect existing effort analysis results mainly from [16, 28, 46] to conduct a qualitative analysis in Table 11 by associating device effects with their affected physical channels. Besides, we also conduct a simple measurement to further improve the qualitative analysis that we deploy devices in our home to collect measurement results and manually identify these effects from measurement results. Considering that the effects will change as the environment changes, we set these quantitative effects to a value range rather than a fixed value and generate several versions of a rule model by randomly selecting values from the range. This can reduce model errors for different environments and improve the availability of detection and repair results. In the future, we can use an online monitor and

Table 11: Configurations of interactions on physical channels.

Physical channel	Device action	Physical effect
Temperature	ACMode.heat	Rise 1°C in 10-15min
	ACMode.cool	Drop 1°C in 10-15min
	thermostatMode.heat	Rise 1°C in 15-20min
	thermostatMode.cool	Drop 1°C in 15-20min
	heater.on	Rise 1°C in 10-15min
	window.open	Rise or drop 1°C in 10-15min according to Temp difference
Humidity	sprinkler.on	Rise 10% in 10-15min
	fan.on	Drop 10% in 15-20min
	humidifier.on	Rise 10% in 10-15min
	dehumidifier.on	Drop 10% in 15-20min
Smoke	waterVavle.on	Clear smoke in 10-15min
	window.open	Clear smoke in 20-25min
Carbon monoxide	fan.on	Clear CO in 15-20min
	window.open	Clear CO in 15-20min
Carbon dioxide	fan.on	Decrease 1 level in 10-15min
	window.open	Decrease 1 level in 10-15min
Sound	window.close	Decrease 20 db
Illuminance	light.on	Increase 100 lux
	light.off	Decrease 100 lux

machine-learning-based (e.g., SVM used in [16]) methods to mine effort models of devices and regularly update them at runtime.

### D Template Equivalence of Correctness Properties

Natural language templates of correctness properties can meet the majority of smart home scenarios that users expect. Based on whether properties are conditional and their states and/or events descriptions, there are 9 natural language templates [47]. These language templates can be summarized as two types of logical templates: *event-based* and *state-based*, shown in Table 12. The former focuses on identifying and handling exceptions timely, while the latter focuses on continuously preventing exceptions from occurring, which are often combined to ensure safety. We use four equivalent relations to implement template conversion and prove the soundness of the translation equivalence as follows:

**Theorem 1.** *The single-state correctness property is logically equivalent to the multi-state one since it is obviously a special case of multi-state properties where  $n$  in  $\bigwedge_{i=1}^n \text{state}_i$  is equal to 1.*

**Theorem 2.** *An unconditional correctness property is obviously logically equivalent to the corresponding conditional one since it can be viewed as having a condition whose value is True.*

**Theorem 3.** *The natural language template (“[event] should [always] happen if [state<sub>1</sub>, ..., state<sub>n</sub>]”) is logically equivalent*

Table 12: Logically equivalent correctness property types.

Summarised property types	Property types	Natural language templates
Event-based	One-Event Unconditional	[event] should [never] happen
	Event-State Conditional (always)	[event] should [always] happen when [state <sub>1</sub> , ..., state <sub>n</sub> ]
	Event-State Conditional (never)	[event] should [never] happen when [state <sub>1</sub> , ..., state <sub>n</sub> ]
State-based	One-State Unconditional (always)	[state] should [always] be active
	One-State Unconditional (never)	[state] should [never] be active
	Multi-State Unconditional (always)	[state <sub>1</sub> , ..., state <sub>n</sub> ] should [always] occur together
	Multi-State Unconditional (never)	[state <sub>1</sub> , ..., state <sub>n</sub> ] should [never] occur together
	State-State Conditional (always)	[state] should [always] be active while [state <sub>1</sub> , ..., state <sub>n</sub> ]
	State-State Conditional (never)	[state] should [never] be active while [state <sub>1</sub> , ..., state <sub>n</sub> ]

to “[¬event] should [never] happen if [state<sub>1</sub>, ..., state<sub>n</sub>]”.

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \equiv \neg F(\bigwedge_{i=1}^n state_i \wedge X(\neg event)) \quad (9)$$

where ¬event has the constraint opposite to event, e.g., ¬event of turning off the heater is turning it on.

*Proof.* The LTL formula of the natural language template describing [event] should [always] happen if [state<sub>1</sub>, ..., state<sub>n</sub>] is given by

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \quad (10)$$

Applying Law of Excluded Middle  $\psi \Rightarrow \chi \equiv \neg\psi \vee \chi$  to (10), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \equiv G(\neg(\bigwedge_{i=1}^n state_i) \vee X(event)) \quad (11)$$

Applying De Morgan’s Law  $\neg\psi \vee \neg\chi \equiv \neg(\psi \wedge \chi)$  to (11), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \equiv G(\neg(\bigwedge_{i=1}^n state_i \wedge \neg X(event))) \quad (12)$$

Applying the negation propagation of X LTL logic  $\neg X(\psi) \equiv X(\neg\psi)$  to (12), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \equiv G(\neg(\bigwedge_{i=1}^n state_i \wedge X(\neg event))) \quad (13)$$

Applying the negation propagation of G LTL logic  $G(\neg\psi) \equiv \neg F(\psi)$  to (13), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow X(event)) \equiv \neg F(\bigwedge_{i=1}^n state_i \wedge X(\neg event)) \quad (14)$$

Theorem 3 is proved.  $\square$

**Theorem 4.** The natural language template (“[state] should [always] be active when [state<sub>1</sub>, ..., state<sub>n</sub>]”) is logically equivalent to “[¬state] should [never] be active when [state<sub>1</sub>, ..., state<sub>n</sub>]”.

$$G(\bigwedge_{i=1}^n state_i \Rightarrow state) \equiv \neg F(\bigwedge_{i=1}^n state_i \wedge \neg state) \quad (15)$$

where ¬state has attribute values except state, e.g., ¬state of alarm.siren is off, strobe, or both.

*Proof.* The LTL formula of the natural language template describing [state] should [always] be active when [state<sub>1</sub>, ..., state<sub>n</sub>] is given by

$$G(\bigwedge_{i=1}^n state_i \Rightarrow state) \quad (16)$$

Applying Law of Excluded Middle  $\psi \Rightarrow \chi \equiv \neg\psi \vee \chi$  to (17), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow state) \equiv G(\neg(\bigwedge_{i=1}^n state_i) \vee state) \quad (17)$$

Applying De Morgan’s Law  $\neg\psi \vee \neg\chi \equiv \neg(\psi \wedge \chi)$  to (16), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow state) \equiv G(\neg(\bigwedge_{i=1}^n state_i \wedge \neg state)) \quad (18)$$

Applying the negation propagation of G LTL logic  $G(\neg\psi) \equiv \neg F(\psi)$  to (18), we have that

$$G(\bigwedge_{i=1}^n state_i \Rightarrow state) \equiv \neg F(\bigwedge_{i=1}^n state_i \wedge \neg state) \quad (19)$$

Theorem 4 is proved.  $\square$

## E Categorized and Prioritized Correctness Properties

Scenario-based correctness property categorization ensures the safety of different devices operating in the same automation scenario as described in Section 6.4. To address property conflicts, TAPFixer develops prioritized correctness properties. It first sorts according to pre-proposition priority and then according to post-proposition priority. TAPFixer defines the pre-proposition priority based on automation scenarios as shown in Table 13 and post-proposition priority based on device capabilities as shown in Table 14. It requires one-pass manual efforts and many of them are reusable across different scenarios since home device types and usage scenarios are limited. Prioritized properties that share the same device capability are listed in ascending priority order in Table 15. The grouping of scenario-based G1-G7 and prioritized G8-G23 are also shown in Table 15.

TAPFixer defines state-based properties to continuously prevent exceptions from occurring (e.g., P.10, P.34, P.44, P.53). The safety-sensitive properties do not specify a latency (e.g., P.10, P.29, P.31, P.42) because it is expected that these safety measures will remain effective until the risk is eliminated. For instance, the security camera in P.10 is expected to work at all times while the user is away, rather than only for a limited period. Whereas other properties can be designed to be satisfied periodically (e.g., P.34, P.44), allowing for permitted latencies to be specified. If there is a tardy attribute in the pre-proposition of a property, it takes a period for the satisfiability of the pre-proposition to change from true to false (e.g., CO<sub>2</sub> drops



Table 13: Sorting descriptions of the pre-proposition priority.

Scenarios in the pre-proposition of the correctness property	Pre-proposition priority
General	user.not_present > user.present, smoke.detected = CO.detected > weather.raining > CO <sub>2</sub> -related = humidity-related
Temperature-related	user.not_present > heater.on = AC.on > the temperature is below / rises above a predefined value

Table 14: Sorting descriptions of the post-proposition priority.

Device Capabilities	Post-proposition Priority
light.switch	light.on = light.off
door.lock	door.lock > door.unlock
security_camera.switch	camera.on > camera.off
switch.switch	switch.off > switch.on
AC.mode	AC.heating_mode = AC.cooling_mode
heater.switch	heater.off > heater.on
coffee_machine.switch	coffee_machine.off > coffee_machine.on
electric_blanket.switch	electric_blanket.off > electric_blanket.on
alarm.state	alarm.activated > alarm.unactivated
ventilating_fan.switch	ventilating_fan.off > ventilating_fan.on
oven.switch	oven.off > oven.on
gas_water_heater.switch	gas_water_heater.off > gas_water_heater.on
gas_valve.switch	gas_valve.off > gas_valve.on
water_valve.switch	water_valve.on > gas_valve.off
sprinkler.switch	sprinkler.on = sprinkler.off
window.switch	window.open = window.close

below the defined threshold in P.34). We define that the latency longer than it is not permitted. Properties with latencies follow state-based property in TAPFixer for violation detection. Take Fig. 1(a) and P.34 with a latency (“fan should be on for at least 10min”) for example, TAPFixer defines a variable “fan.timer” to record the remaining operating time of the fan (see §4.2) and translates P.34 into the LTL form:  $G(\text{CO}_2 > \text{a predefined value} \wedge (\text{fan.timer} \geq \text{fan.config\_latency}-600) \Rightarrow \text{fan.on})$ , which follows the LTL template of the state-based property in Table 1.

Table 15: Categorized and prioritized correctness properties.

Property	Description	Scenario-based category types	Priority-based category types
P.1	IF the user arrives home, the light should be turned on.	G1	G8
P.2	IF the user is not at home / not nearby-home, the light should be turned off.	G2	G8
P.3	WHEN the user is not at home / not nearby-home, the light should be off.	G2	G8
P.4	IF the user arrives home, the garage door should be opened.	G1	G9
P.5	IF the user leaves home, the garage door should be closed.	G2	G9
P.6	WHEN the user leaves home, the garage door should be closed.	G2	G9
P.7	IF the user is not at home / not nearby-home, the door should be locked.	G2	G9
P.8	WHEN the user is not at home / not nearby-home, the door should be locked.	G2	G9
P.9	IF the user is not at home / not nearby-home, the security camera should be turned on.	G2	G10
P.10	WHEN the user is not at home / not nearby-home, the security camera should be on.	G2	G10
P.11	IF the door opens while the user is not at home / not nearby-home, the security camera should take pictures.	G2	G10
P.12	IF the user is not at home / not nearby-home, the switch should be turned off.	G2	G11
P.13	WHEN the user is not at home / not nearby-home, the switch should be off.	G2	G11
P.14	IF the temperature is below a predefined value and someone is at home, the AC should be in heating mode.	G3	G12
P.15	IF the temperature rises above a predefined value, the AC should be in cooling mode.	G3	G12
P.16	WHEN the heater is on, the AC should be off.	G3	G12
P.17	IF the user is not at home / not nearby-home, the AC should be turned off.	G2	G12
P.18	WHEN the user is not at home / not nearby-home, the AC should be off.	G2	G12
P.19	IF the temperature is below a predefined value while someone is at home, the heater should be turned on.	G3	G13
P.20	IF the temperature rises above a predefined value, the heater should be turned off.	G3	G13
P.21	WHEN the AC is on, the heater should be off.	G3	G13
P.22	IF the user is not at home / not nearby-home, the heater should be turned off.	G2	G13
P.23	WHEN the user is not at home / not nearby-home, the heater should be off.	G2	G13
P.24	IF the user is not at home / not nearby-home, the coffee machine should be turned off.	G2	G14
P.25	WHEN the user is not at home / not nearby-home, the coffee machine should be off.	G2	G14
P.26	IF the user is not at home / not nearby-home, the electric blanket should be turned off.	G2	G15
P.27	WHEN the user is not at home / not nearby-home, the electric blanket should be off.	G2	G15
P.28	IF the smoke is detected, the alarm should be activated.	G4	G16
P.29	WHEN there is smoke, the alarm should be activated.	G4	G16
P.30	IF CO is detected, the alarm should be activated.	G5	G16
P.31	WHEN CO is detected, the alarm should be activated.	G5	G16
P.32	IF humidity is greater than a predefined value, the ventilating fan should be turned on.	G6	G17
P.33	IF CO <sub>2</sub> is greater than a predefined value, the ventilating fan should be turned on.	G4	G17
P.34	WHEN CO <sub>2</sub> remains greater than a predefined value, the ventilating fan should be on for at least the permitted time.	G4	G17
P.35	IF the user is not at home / not nearby-home, the oven should be turned off.	G2	G18
P.36	WHEN the user is not at home / not nearby-home, the oven should be off.	G2	G18
P.37	IF CO is detected, the natural gas hot water heater should be turned off.	G5	G19
P.38	WHEN CO is detected, the natural gas hot water heater should be off.	G5	G19
P.39	IF CO is detected, the gas valve should shut off.	G5	G20
P.40	WHEN CO is detected, the gas valve should be closed.	G5	G20
P.41	IF the smoke is detected, the water valve should be turned on.	G4	G20
P.42	WHEN there is smoke, the water valve should be on.	G4	G20
P.43	IF the soil moisture sensor is below a predefined value, the sprinkler system should be turned on.	G6	G21
P.44	When the soil moisture sensor is below a predefined value, the sprinkler system should be on for at least the permitted time.	G6	G21
P.45	IF the weather is raining, the sprinkler should be turned off.	G7	G21
P.46	WHEN the weather is raining, the sprinkler should be off.	G7	G21
P.47	IF CO <sub>2</sub> is greater than a predefined value, the window should be opened.	G5	G22
P.48	IF the weather is raining, the window should be closed.	G7	G22
P.49	WHEN the weather is raining, the window should be closed.	G7	G22
P.50	IF the smoke is detected, the window should be opened.	G4	G23
P.51	WHEN there is smoke, the window should be opened.	G4	G23
P.52	IF CO is detected, the window should be opened.	G5	G23
P.53	WHEN CO is detected, the window should be opened.	G5	G23