TAPPecker: TAP Logic Inference and Violation Detection in Heterogeneous Smart Home Systems

Qixiao Lin
Beihang University
China
linqx529@buaa.edu.cn

Jian Mao*
Beihang University
Tianmushan Laboratory
Hangzhou Innovation Institute
Zhongguancun Laboratory
China
maojian@buaa.edu.cn

Ziwen Liu
Beihang University
China
liuziwen@buaa.edu.cn

Zhenkai Liang
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

***Abstract*—In IoT environments—particularly within smart home systems—Trigger-Action Programming (TAP) serves as the primary mechanism for specifying automation rules. While TAP enables flexible and user-friendly automation, unintended interactions among TAP rules that violate security or privacy policies can lead to serious security consequences. A common assumption in prior research is that automation rules are readily available for analysis—that is, the TAP logic can be directly accessed. However, many real-world IoT platforms, such as Xiaomi and HomeKit, do not expose their internal automation rule sets. Moreover, existing logic extraction techniques primarily focus on the relationships between individual events, failing to capture the complex semantics of multi-condition TAP rules. The growing heterogeneity of smart home systems complicates the challenge of ensuring consistency between automation logic execution and system security objectives across such diverse "black-box" systems.**

In this paper, we present TAPPecker, an approach that leverages self-adaptation and evolutionary strategies to automatically infer TAP rules from system events in heterogeneous smart home environments. We analyze the inferred rules to detect potential security violations, with a specific focus on temporal aspects. We prototype TAPPecker and develop a hybrid testbed capable of generating realistic smart-home event logs, enabling comprehensive, multi-scenario testing. Our experimental results demonstrate that TAPPecker improves inference accuracy by 40.85% over existing approaches, while generating more expressive TAP logic and uncovering previously undetected security violations. Notably, our system revealed two time-related policy violations within official TAP rule sets that had not been previously reported.

***Index Terms*—Smart Home; Trigger-Action Programming; Logic Inference; Model Checking**

I. Introduction

Smart home, as a typical IoT application, allows users to implement automation in commodity appliances and devices, enhancing the convenience of their daily lives. In such scenarios, devices interact with each other through Trigger-Action Programming (TAP) [8], [19], [22], [31], [43], [51], which uses *if-then* rules to specify the logic of the expected action (e.g., turning on a light) based on a triggered event (e.g., a motion sensor activated). Automation programming in IoT platforms allows users to specify complex TAP logic among diverse devices (e.g., electronic appliances, environmental sensors, alarms).

Multiple TAP rules in a smart home environment may interact with each other, leading to undesirable situations and potential security and privacy violations [6], [7], [17], [21], [45]. Researchers have proposed various solutions to scrutinize IoT device interactions within a set of TAP rules. Soteria [6] and IoTGuard [7] prevent unsafe events by analyzing rule violations through behavioral state models, employing model checking and runtime execution monitoring. IoTMon [17] approaches the problem by analyzing physical interaction chains among devices, evaluating risk levels and identifying potentially hazardous device interactions.

The above solutions assume that the TAP logic is *readily accessible*. Smart homes increasingly integrate devices from different vendors, introducing challenges in analyzing TAP logic across heterogeneous platforms (e.g., SmartThings [40], HomeKit [2], Home Assistant [3] and IFTTT [24]). In real-world heterogeneous smart-home systems, TAP logic from different platforms often remains close-source. For instance, TAP logic in platforms like Xiaomi and HomeKit is not directly accessible. TAP logic forms the foundation of smart home security mechanisms, including property violation checking [6], [9], [17], [51], security/safety policy enforcement [7], [49], anomaly detection [21], [30] and provenance [32], [46]. The *inaccessibility* of TAP logic in such heterogeneous smart home environments raises a essential questions concerning the identification and verification of unsafe TAP logic: *how to identify and represent the executed TAP logic in a unified way*.

Intuition. TAP logic execution in a smart-home system begins with a device or service state change, triggering subsequent state changes in other devices or services. Such state changes can be monitored and collected at a local network's centralized device (e.g., a home gateway or a hub) — the convergence point for all inter-device traffic. When we treat each state change as an event, we can observe a sequential (but not

* Jian Mao is the corresponding author.

necessarily contiguous) pattern of trigger and action events in the logs. These event sequences preserve the implicit logic of TAP in the system, making the event log an accessible and valuable resource for analyzing TAP logic interactions.

Traditional approaches based on static analysis [21], [52] and data mining [10], [27] profile system behaviors by examining/verifying relationships between pairs of individual devices/events. Their representation of "relation" fails to express complex TAP rules involving multiple conditions connected by logical operators such as "or" or "and". This limits accurate description and inference of *multi-condition* TAP logic. Existing deep learning approaches [39], [47] determine whether an event occurs in an appropriate context. But they cannot extract the underlying inter-event relationships as TAP logic. **To address these limitations, we intend to represent complex TAP logic conditions in a unified way and extract them from event logs.** We adopt boolean polynomial expressions as a unified representation as they offer higher expressiveness. While enumerating and validating all possible logic combinations against ground truth (i.e., event logs) offers a direct solution, it demands intensive computation. Taking inspiration from evolutionary principles, we propose an adaptive approach that continuously evolves candidates based on their adaptability to the ground truth, achieving precise alignment with observed behaviors.

In this paper, we propose TAPPecker, a self-adaptive TAP logic inference mechanism to infer the TAP logic of smart-home systems. Given the action of devices, TAPPecker can adaptively evolve candidate environmental conditions to find the most suitable one with high effectiveness. The inferred TAP logic enables various smart home security analyses, including anomaly detection, provenance analysis, rule interaction analysis, and security violation identification. We demonstrate this capability by identifying potential violations through model checking, using an automaton model built from the inferred TAP logic. Inspired by existing security checking methods [33], [48], [49], [51] which consider the temporal factor, we propose new templates for time-related security policies from multiple perspectives, including the longest/shortest duration, within/outside the specified time, etc.

To validate our approach, we implement a hybrid testbed to generate smart home event logs. The platform allows us to configure TAP rules from heterogeneous environments. It collects event logs that imply user-specific TAP logic and physical correlations in real scenarios, eliminating the need for laborious setup. This platform supports us in collecting comprehensive data and conducting multi-perspective experiments.

Contribution. In summary, our contributions are as follows:

- We propose a self-adaptive approach for TAP logic inference in heterogeneous smart-home environments. By representing TAP logic as the boolean polynomial expression in a unified way, our approach evolves and validates possible TAP logic based on their fitness based on their fitness to observed event logs from real IoT environments.
- We summarize a comprehensive set of security policy templates incorporating temporal factors (longest/shortest duration, within/outside specified time periods). Based on the inferred results, we develop an integrated model-checking solution to detect safety/security policy violations.
- We prototype and evaluate TAPPecker. We develop a hybrid testbed to generate datasets for smart home systems. Based on our collected data, evaluation results show TAPPecker improves TAP logic inference accuracy by 40.85% compared to existing approaches, achieving complete consistency with configured TAP logic.

II. Background and Problem Statement

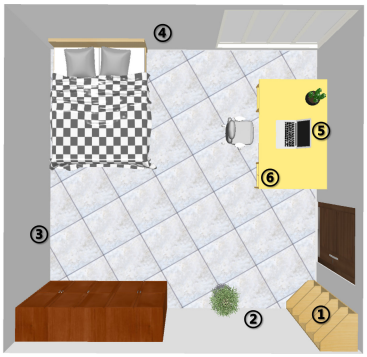
Automation forms a core functionality in smart home systems. Smart home platforms [2], [3], [36], [40] enable users to install and specify automation program for device integration. These platforms increasingly integrate with third-party trigger-action platforms [24], [34], [50] to connect physical devices with digital processes.

Automation rules across different platforms follow the *Trigger-Action Programming* (TAP) paradigm [19], [23], [38], [43], [44], [51]. A TAP statement follows the format "**IF** $[trigger]$ **THEN** $[action]$", where $trigger$ represents a device or service state change (e.g., "door opens") and $action$ specifies a device actuation (e.g., "turn on the welcome light") or service operation (e.g., "send an email notification").

A. A Motivating Example: Turning off the Guard System Unexpectedly in a Heterogeneous Smart Home System

Unexpected combined TAP logic in heterogeneous smart home systems may violate safety and security properties in the IoT environment, causing privacy leaking, property loss, and even physical damage. Consider a smart home scenario with devices from different brands (Figure 1a) and multiple trigger-action rules (Figure 1b). The bedroom setup includes a Samsung smart socket and several Xiaomi devices. The Xiaomi Gateway features a Security Guard mode that must remain active. In this mode, the gateway pushes alert notifications to the Mijia app and emits continuous warning sounds when configured devices detect anomalies. Home Assistant serves as the integration platform connecting Samsung's SmartThings and Xiaomi's Mi Cloud. To facilitate home automation management, we define a *home mode* (Mode) and configure multiple trigger-action rules in Xiaomi's Mijia App and the Home Assistant platform, as shown in Figure 1b.

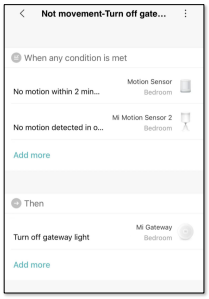
The user may misconfigure the "turn off appliances" actions in the rule R7, which will turn off the gateway light, the smart socket, and the security guard mode. Sequential execution of automation rules may induce an unsafe and insecure environment. For example, when a user is at home (the contact sensor is closed) but inactive (two motion sensors are clear), rules R2, R4, and R7 will be triggered sequentially, ultimately disabling the gateway's security guard mode and leaving the environment unprotected.



(a) The bedroom environment with heterogeneous devices.

Platform	#	IF	THEN
Mijia	R1	**MS1** detected or **MS2** detected or single-click **B**	turn on **L**
	R2	**MS1** clear and **MS2** clear or double-click **B**	turn off **L**
Home Assistant	R3	**CS** closed and **L** on	change **Mode** to "home"
	R4	**CS** closed and **L** off	change **Mode** to "sleep"
	R5	**CS** open	change **Mode** to "away"
	R6	hold **B**	turn on **S**
	R7	**Mode** changes to "sleep" or "away"	turn off all appliances

(b) Multiple TAP rules configured in the environment.



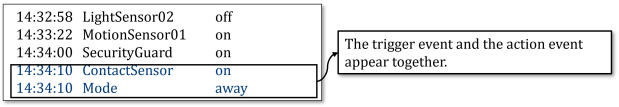

```
- id: '1651069340131'
  alias: change-mode-to-away
  description: ''
  trigger:
  - type: opened
    platform: device
    device_id: f00ce68a9aeca36034c926b2342d3e0b
    entity_id: binary_sensor.e4aaec44d616_contact
    domain: binary_sensor
  condition: []
  action:
  - service: input_select.select_option
    target:
      entity_id: input_select.mode
    data:
      option: away
  mode: single
```

(c) Trigger-action rule representation from the Mijia App (left) and the Home Assistant platform (right).

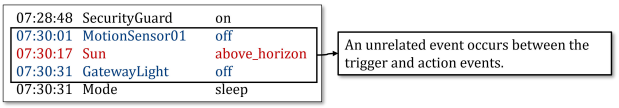
Fig. 1: A smart home scenario with heterogeneous devices and multiple TAP rules.

Prior research on IoT safety/security property violations focuses primarily on Samsung's SmartThings or IFTTT platforms. These studies assume TAP logic can be extracted through static analysis of SmartThings' SmartApps and IFTTT applets. However, heterogeneous smart home systems involve devices from multiple manufacturers. This diversity complicates TAP logic extraction. Taking the above smart home scenario as an example, while Home Assistant automation can be extracted from the `automation.yaml` file, automation customized on the Mijia app cannot be automatically collected.

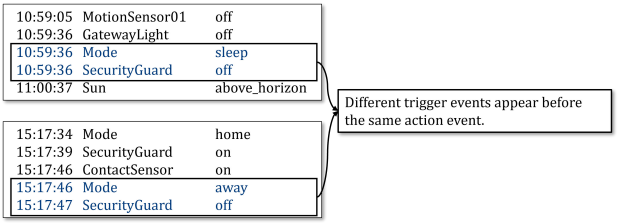
Furthermore, TAP logic representation varies across platforms. Figure 1c illustrates different trigger-action rule formats used in the Mijia App and the Home Assistant platform. This necessitates a unified model for cross-platform TAP logic representation.



(a) A log snippet related to the execution of the rule R5.



(b) A log snippet related to the execution of the rule R2.



(c) Two log snippets related to the execution of the rule R7.

Fig. 2: Some log snippets collected from the example smart home scenario.

B. Problem Analysis

Most existing research on TAP logic extraction focuses on the SmartThings platform, relying on static source code analysis. We aim at a general heterogeneous smart home system, where users deploy devices from multiple vendors and manage them through various proprietary platforms. This makes it difficult to fully/easily obtain automation/apps source code, rendering the TAP dependency analysis from previous approaches incomplete. The key challenge in safety/security checking for general smart home scenarios is **how to identify executed TAP logic and represent them in a unified format.**

The execution of TAP logic begins when a state change occurs in a device or service. This change then triggers an automatic state change in another device or service. Event logs in smart home systems from a centralized devices (e.g., a home gateway or a hub) record these state changes and implicitly indicate the executed TAP logic while providing information about its content. TAP logic execution follows a deterministic pattern—action events typically follow trigger events in the log, though not with absolute consistency.

We observe that the execution of TAP logic begins when a state change occurs in a device or service. This change subsequently triggers an automatic state change in another device or service. Event logs in smart home systems record these state changes and implicitly indicate the executed TAP logic while providing essential information about its content. TAP logic execution follows a deterministic pattern: action events typically follow trigger events in the log.

For example, Figure 2a shows a log snippet during rule R5 execution in our smart home scenario. The snippet highlights both the trigger event and action event in blue, revealing that the trigger event ("ContactSensor open") appears strictly

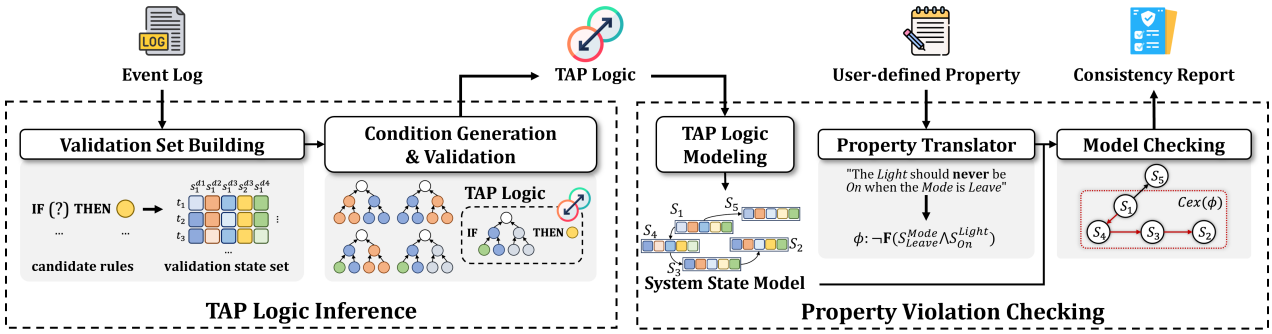


Fig. 3: Workflow of TAPPecker.

before the action event ("Mode changes to away"). By analyzing the whole event log, we can calculate the frequency of "the trigger event occurs before the action event" as 1. This observation naturally suggests a statistical approach for TAP logic inference. We can identify dependent event pairs by calculating how frequently certain events precede target events in the logs.

When analyzing the logs corresponding to other TAP logic, we conclude that the inference approach should meet two key requirements.

- **Eliminate the impact of irrelevant events**: Unrelated events often appear between trigger and action events in TAP logic. Figure 2b shows a log snippet of rule R2 execution with an unrelated event (highlighted in red) occurring between the rule's trigger and action events (highlighted in blue)..
- **Handle TAP logic with multiple conditions**: Users may configure complex TAP logic with multiple conditions connected by logic operators "or" and "and". This complexity means one trigger event may not always appear before action events with absolute consistency. Figure 2c illustrates the log snippets of the rule R7 triggered by different events (highlighted in blue). The trigger events ("Mode changes to sleep" and "Mode changes to away") precede the action event ("Turn off appliances") with frequencies of 0.77 and 0.23, respectively. It is noteworthy that complex logic holds a significant presence (44.1%) within current publicly available datasets, highlighting the need to extract complex multi-condition TAP logic in "black-box" smart home systems.

Statistic-based approaches in prior work [18], [21], [52] focus only on simple event relationships. These approaches assume one event immediately triggers another, failing to meet our requirements or describe complex multi-condition TAP logic. Complex trigger relationships require a more expressive representation to capture how multiple events collectively contribute to triggering an action. We address this limitation by adopting boolean polynomial expressions as a unified representation for describing relationships among multiple events.

Definition of TAP Logic Representation. We define a TAP logic as "**IF** $[trigger]$ **THEN** $[action]$." For a TAP logic whose $[action]$ is a, its $[trigger]$ is defined as ϵ_a, which may be a single environment condition or multiple sub-conditions connected by $\wedge$ or $\vee$. A single condition is a logic expression representing that the status of a *device* should *satisfy* the specific *value*. We use "equal to $(=)$," "less than $(<)$" and "greater than $(>)$" to specify the satisfaction relationship. The trigger of a TAP logic ϵ_a is a boolean polynomial of different single conditions combined by logical connectors. In this way, a TAP logic "**IF** the motion sensor M is ON and the temperature sensor T is larger than 20, **THEN** turn on the air condition AC" will be represented as "**IF** $M = ON \& T > 20$ **THEN** turn on AC"

Statistic-based approaches generate candidate relations and verify them against observations using various statistical metrics, such as frequency, p-value [21] and cross-correlation [52]. The enumeration method inherently suffers from high computational complexity, limiting its scalability. Our goal is to try possible combinations of operations and identify the relationship that best fits the observation. Intuitively, this process is similar to a process of evolving programs to fit specific tasks requirements. Inspired by evolutionary principles, we formulate TAP logic inference in "black-box" smart home systems as a process of generating and evaluating candidates to find the most suitable one based on observations (i.e., the event log from the real IoT environment). Through evolutionary operations, each new generation outperforms its predecessors, making our inference process more efficient than enumeration methods.

III. Design

Figure 3 illustrates the workflow of TAPPecker with its two main components: the TAP Logic Inference Module and the Property Violation Checking Module. The TAP Logic Inference Module extracts validation sets from event logs, generates and evolves possible conditions for target device actions, and outputs TAP logic in our defined format. This

module guarantees that any TAP logic will be successfully inferred as long as it has been previously triggered and recorded in the system event logs. The Property Violation Checking Module constructs an automaton from the inferred TAP logic to formally model the smart-home system. It then converts user-defined policies to LTL formulas using predefined templates and check consistency between inferred TAP logic and security/safety policies.

A. TAP Logic Inference

The TAP Logic Inference module takes event logs as input, extract input-output pairs of TAP logic as its validation sets, traverses all actuators in the system, and identifies their possible transitions as inferred TAP logic. For a target action a, we determine both the existence of TAP logic performing action a and its condition ϵ_a through the following process.

For a smart home system with n devices, we define a device set $\mathcal{D}$ as

$$\mathcal{D} = \{D^i | i = 1, 2, ..., n\}, \tag{1}$$

where D^i represents the i-th device. Each device has a specific state set. For device D^i with m_i states, its state set[1] $\int^i$ is defined as

$$\int^i = \{s_k^i | k = 1, 2, ..., m_i\}. \tag{2}$$

Time-related conditions (e.g., duration constraints and counters) are critical components in the TAP logic. To model these factors, we also consider the timer and the duration as numeric devices. We generate candidate condition combinations from device set $\mathcal{D}$ (excluding the target device) for the target action a and validate these against the facts observed from event logs. Following evolutionary principles in the problem analysis, we adopt Genetic Programming [28] as the main technique for TAP logic inference. We propose a self-adaptive technique to generate, mutate, and validate candidate conditions for state transitions.

1) Condition Generation and Evolution: Polynomial expressions typically use tree-based representations. Similarly, we represent TAP logic conditions as trees, with different conditions serving as terminal nodes and logic connectors ("and" and "or") as tree nodes. Each condition node consists of three components: a device node, a logic operator node, and a value node. Tree-based structures inherently facilitate evolution and evaluation of mathematical expressions. This advantage naturally extends to our TAP logic conditions, enabling efficient evolution and validation of candidate conditions for device actions.

We generate, evolve, and validate different population generations as illustrated in Algorithm 1. In our context, a "population" refers to a set of candidate conditions (i.e., a set of "individuals") for a specific state transition, structured as the tree-based structure. The algorithm execute N iterations of genetic processes, where each iteration begins with an initial population of randomly generated conditions (Lines 14-17) and iteratively produces subsequent generations. For each generation, the algorithm performs three key operations:

- **Selection.** It evaluates each condition's fitness using validation set V(Line 3-11). Fitness serves as a metric for evaluating candidate conditions. We calculate the fitness scores by examining each record in the validation set V, whose construction process will be introduced later. For each record, we substitute the device states in the candidate condition with their specific values and determine whether the resulting logical output matches the actual observation. The alignment between predicted and

[1]For example, the state set of a motion sensor M is $\{ON, OFF\}$.

Algorithm 1: Generating, evolving and validating the candidate conditions for a specific action a of the target device

```
1  Input: Validation set V and the candidate device set D
2  Output: The most possible condition ε_a
3  Function valiPop(population[gen], V):
4      foreach tree ∈ population[gen] do
5          s[tree] ← 0
6          foreach v ∈ V do
7              apply the state value in v to tree and calculate
                the logic result as r
8              if r = v[output] then
9                  s[tree] + +
10     sort s in ascending order
11     return s[0]
12 condSet ← NULL
13 while i ≤ N do
14     gen ← 0
15     population[gen] ← NULL
16     while len(population[gen]) ≤ popSize do
17         append randomTree(depth, D, prob_t) to
            population[gen]
18     score ← valiPop(population[gen], V)
19     while gen ≤ M and score > 0 do
           // Selection
20         assign parents as the top N trees in
            population[gen] that fits V
21         gen + +
22         population[gen] ← NULL
23         while len(population[gen]) ≤ popSize do
24             randomly choose two trees tree_1 and tree_2
                from parents
               // Crossover
25             newTree ← crossover(tree_1, tree_2, prob_c)
               // Mutation
26             newTree ← mutate(newTree, D, prob_m)
27             append newTree to population[gen]
28         score = valiPop(population[gen], V)
29     store the first tree of population[gen] and its fitness
        score in condSet
30 assign bestCond as the best fitting condition in condSet
31 if the fitness level of bestCond meets the threshold then
32     return bestCond
33 else
34     return NULL
```

We present details of Function `randomTree`, `crossover` and `mutate` in Appendix A.

Algorithm 2: Building validation set for a specific action from the event log.

Input: Event log E, target action a (including the target device - D^{tar}, its states before and after performing the action - s^{tar}_{start} and s^{tar}_{end}) and its candidate device set $\mathcal{D}$
Output: Validation set V
$V \leftarrow \emptyset$
// v is a dictionary, whose keys are the candidate device from $\mathcal{D}$ and the *output*
$v \leftarrow dict()$
foreach $device \in \mathcal{D}$ **do**
  $v[device] \leftarrow NULL$
$v[output] \leftarrow 0$
// $lastState$ stores the state of the target device D^{tar}
$lastState \leftarrow NULL$
foreach $event \in E$ **do**
  // $event$ is a triple (T, D, S), indicating that at the time T, the state of the device D changes to S
  if $D = D^{tar}$ **then**
    if $lastState = s^{tar}_{start}$ **and** $S = s^{tar}_{end}$ **then**
      $v[output] \leftarrow 1$
    else
      $v[output] \leftarrow 0$
    append v to V
    $lastState \leftarrow S$
  else if $D \in \mathcal{D}$ **then**
    $v[D] \leftarrow S$

observed outcomes forms the basis of the fitness score calculation. It then selects n top-performing conditions as parents for the next generation based on these scores (Line 20).

- **Crossover.** With the crossover probability $prob_c$, it randomly selects two parent conditions ($tree_1$ and $tree_2$) and creates a new condition by replacing a subtree in $tree_1$ with a random subtree from $tree_2$ (Lines 24-25).
- **Mutation.** Each node in the newly generated condition undergoes mutation with the mutation probability $prob_m$ (Line 26). Mutation operations include generating new subtrees and modifying logic operators or condition node values.

Each iteration terminates when either the generation count reaches the maximum limit M or a candidate condition in the population perfectly matches the validation set V (Line 19). From the conditions validated across N iterations, we select the best-fitting condition as the inferred condition ϵ_a for the target action. If the inferred condition's fitness falls below a pre-defined threshold, we conclude that no valid TAP logic exists for the target action a.

2) Validation Set Building: Our approach closely parallels approaches that derive mathematical expressions from observed facts. To evaluate the fitness of mathematical expressions, we need validation sets consisting of "input-output" pairs. In these pairs, inputs are parameter values in the expression, while outputs are corresponding expression results. For example, validation pairs for the function `X*(Y+5)` might include `(X = 1, Y = 2, output = 7)` and `(X = 3, Y = 4, output = 27)`.

Similarly, we prepare validation sets for state transitions from event logs. For TAP logic, inputs are candidate device states, and outputs indicate whether transitions occur under given conditions (1 for occurrence and 0 for non-occurrence, respectively). For instance, validation pairs for the TAP logic "IF M is ON and T is larger than 20, THEN turn on AC" might include `(M = ON, T = 25, output = 1)`, `(M = OFF, T = 23, output = 0)`, and `(M = ON, T = 15, output = 0)`. We represent the event log as a sequence $E = \{e_1, e_2, ..., e_N\}$, where each event e_i is a triple (T, D, S), indicating that at the time T, the state of the device D changes to S. Given the target state transition (the target device D^{tar} and its start and end states s^{tar}_{start} and s^{tar}_{end}) and its candidate device set $\mathcal{D}$, we construct validation sets by recording target device state changes and corresponding candidate device states, as shown in Algorithm 2.

B. Property Violation Checking

1) TAP Logic Modeling: We model the smart home system as a finite state machine (FSM) $\mathcal{M} = (\mathcal{S}, \Sigma, \mathcal{I})$. $\mathcal{S}$ represents the (finite) set of states. For a system with n devices, each state $S \in \mathcal{S}$ is represented as an n-dimensional vector $S = (s^{d_1}, s^{d_2}, ..., s^{d_n})$, where each dimension represents the state of a corresponding device. $\mathcal{I} \subseteq \mathcal{S}$ defines the set of initial states, and $\Sigma \subseteq \mathcal{S} \times \mathcal{S}$ represents the set of state-transition functions.

Each inferred TAP logic rule "**IF** ϵ_a **THEN** a" is modeled as a state-transition function $\delta(S, S') \in \Sigma$ in $\mathcal{M}$. Given that the action a changes the state of the target device D^{tar} to s^{tar}_j, the transition function is defined as:

$$S' = \begin{cases} \text{Assign}^{tar}_j(S), & \text{if } S \text{ satisfies } \epsilon_a \text{ and } S[tar] \neq s^{tar}_j \\ S, & \text{otherwise,} \end{cases} \tag{3}$$

where $\text{Assign}^{tar}_j(S)$ is a function to update the state of device D^{tar} to s_j in state S according to the action a.

For formal verification, we employ NuSMV [11], [12], a state-of-the-art model checker. NuSMV's specification language directly supports FSM modeling, enabling straightforward translation of our state-transition automata into its input format.

2) Policy Translator: Model checking requires formal specification of expected system behaviors and properties using formal logic such as linear temporal logic (LTL) [29] or computation tree logic (CTL) [13]. While these logics effectively specify time-related properties using state variables and temporal operators, they remain too abstract for users without formal methods expertise. We address this by automatically generating LTL formulas from pre-defined templates. In this way, security professionals can define policies and present them to end-users in natural language. Users select the rules that best fit their deployment, and TAPPecker automatically converts the chosen policies into corresponding LTL formulas.

TABLE I: Safety/Security policy templates. **G**, **F**, **X** and **U** are LTL operators representing "always **G**lobally", "eventually in the **F**uture", "ne**X**t" and "**U**ntil". $[device]$ and $[state]$ are user inputs, selected from candidate device list and the corresponding state set.

Policy Type	Input Template	LTL Formula
Single Device	The $[device^d]$ should **always** be $[state_i]$	$\mathbf{G}(s_i^d)$
	The $[device^d]$ should **never** be $[state_i]$	$\neg\mathbf{F}(s_i^d)$
Multiple Devices	The $[device^1]$, ... , the $[device^d]$ should **always** be $[state_1]$, ... , $[state_i]$ together	$\mathbf{G}(s_1^1 \wedge ... \wedge s_i^d)$
	The $[device^1]$, ... , the $[device^d]$ should **never** be $[state_1]$, ... , $[state_i]$ together	$\neg\mathbf{F}(s_1^1 \wedge ... \wedge s_i^d)$
	The $[device^p]$ should **always** be $[state_q]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}((s_1^1 \wedge ... \wedge s_i^d) \rightarrow \mathbf{X}s_q^p)$
	The $[device^p]$ should **only** be $[state_q]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}(\mathbf{X}s_q^p \rightarrow (s_1^1 \wedge ... \wedge s_i^d))$
	The $[device^p]$ should **never** be $[state_q]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\neg\mathbf{F}(s_1^1 \wedge ... \wedge s_i^d \wedge s_q^p)$
	The $[device^p]$ should **eventually** be $[state_q]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}((s_1^1 \wedge ... \wedge s_i^d) \rightarrow \mathbf{F}s_q^p)$
	The $[device^p]$ should **always** be $[state_q]$ **within** $[t]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}((s_1^1 \wedge ... \wedge s_i^d) \rightarrow \mathbf{X}(time \leq t\ \mathbf{U}\ s_q^p))$
	The $[device^p]$ should **always** be $[state_q]$ and **last for** $[t]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}((s_1^1 \wedge ... \wedge s_i^d) \rightarrow \mathbf{X}(s_q^p\ \mathbf{U}\ time > t))$
	The $[device^p]$ should **only** be $[state_q]$ **within** $[t]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}(\mathbf{X}(time \leq t\ \mathbf{U}\ s_q^p) \rightarrow (s_1^1 \wedge ... \wedge s_i^d))$
	The $[device^p]$ should **only** be $[state_q]$ and **last for** $[t]$ **when** the $[device_1]$ is $[state_1]$, ... , and the $[device^d]$ is $[state_i]$	$\mathbf{G}(\mathbf{X}(s_q^p\ \mathbf{U}\ time > t) \rightarrow (s_1^1 \wedge ... \wedge s_i^d))$

Time-related conditions in the TAP logic require timing-dependent security policies. Building on approaches that incorporate temporal elements in security checking [33], [49], [51], we developed templates for security policies covering multiple perspectives: longest/shortest duration, within/outside specified times, and others. Table I presents these policy categories with their corresponding LTL formula templates. For example, a conditional policy "The $Light$ should **never** be On **when** the $Mode$ is $Leave$" will be translated as the LTL formula $\neg\mathbf{F}(s_{Leave}^{Mode} \wedge s_{On}^{Light})$.

3) Model Checking: Automatic execution of TAP logic can cause device state changes that violate user-defined safety or security policies, leading unexpected and unsafe situations. Consider a safety/security policy ϕ and a device state-change trace $\mathscr{T} = (\mathscr{S}_1, \mathscr{S}_2, ..., \mathscr{S}_T)$, where $\mathscr{S}_t$ represents all device states at a given moment. State transitions from $\mathscr{S}_t$ to $\mathscr{S}_{t+1}$ follow inferred TAP logic. Chained execution of TAP logic violates the specific safety/security policy when the trace $\mathscr{T}$ fails to satisfy the constraint ϕ.

Using the home system automata $\mathcal{M}$ and translated LTL formula ϕ as constraints, we employ the NuSMV symbolic model checker to verify constraint satisfaction and generate counter-examples $Cex(\phi)$ for constraint violations.

IV. Evaluation

In this section, we evaluate TAPPecker on these aspects:

- **RQ1**: Compared to the state of the art, how effective is TAPPecker in inferring TAP logic? How consistent is the inferred TAP logic with the ground truth? (Section IV-B)
- **RQ2**: Will the inferred TAP logic be applicable for checking property violations? (Section IV-C)
- **RQ3**: How sufficient are the new-defined security policy templates to detect time-related security violations? (Section IV-C)
- **RQ4**: What is the time overhead of TAPPecker? (Section IV-D)

A. Dataset

Dataset Generation. There are several sensor datasets collected from the physical environment for Human Activity Recognition (such as [14], [15]). However, they come from non-"smart" environments and contain only sensor data without implicit user-specific TAP logic, making them unsuitable for evaluating smart home research, including TAPPecker. Smart home researchers typically address this gap by building physical testbeds with various devices (including sensors and actuators) and configuring specific rules. They then collect event logs while participants perform daily activities over extended periods (minimum one month). These self-collected datasets rarely become public, and recreating them requires significant time investment. This challenge motivated our development of a hybrid smart home testbed for user behavior simulation and dataset generation.

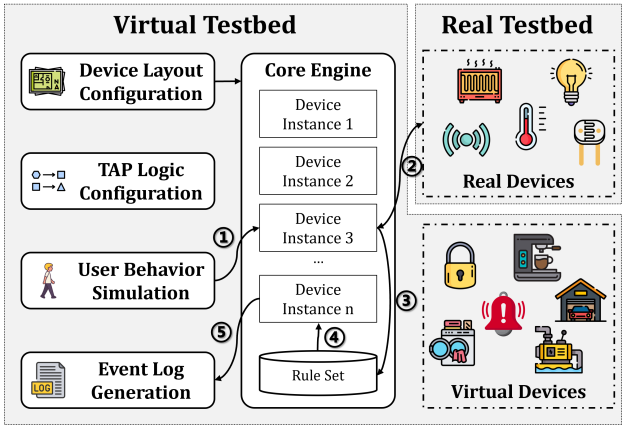


Fig. 4: The architecture of our hybrid testbed for data generation.

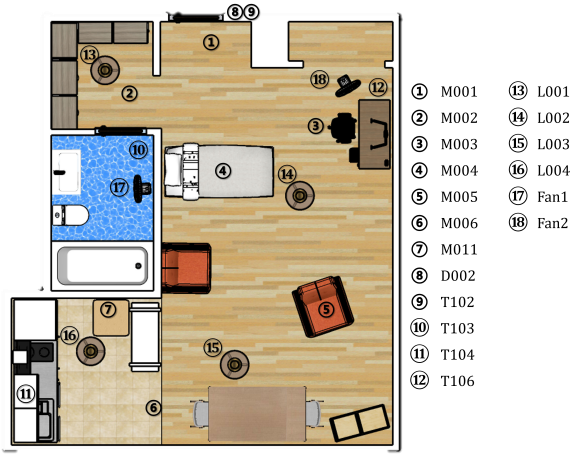


Fig. 5: The home layout with different devices for **Set S1**.

#	IF	Then
1	**M002** ON	**L001** ON
2	**M002** OFF	**L001** OFF
3	**M003** ON or **M004** ON	**L002** ON
4	**M003** OFF and **M004** OFF	**L002** OFF
5	**M005** ON or **M006** ON	**L003** ON
6	**M005** OFF and **M006** OFF	**L003** OFF
7	**M011** ON	**L004** ON
8	**M011** OFF	**L004** OFF
9	**Time** is 11pm	**Mode** changes to "Sleep"
10	**Time** is 7am	**Mode** changes to "Home"
11	**D002** Open and **Mode** is "Home" or "Sleep"	**Mode** changes to "Leave"
12	**D002** Open and **Mode** is "Leave"	**Mode** changes to "Home"
13	**Mode** is "Sleep"	Turn off **all appliances**
14	**T103** >= 26 ℃	**Fan1** ON
15	**T103** < 26 ℃	**Fan1** OFF
16	Indoor temp. (**T106**) is 3 ℃ higher than outdoor (**T002**)	**Fan2** ON
17	Indoor temp. (**T106**) is not 3 ℃ higher than outdoor (**T002**)	**Fan2** OFF

Fig. 6: The TAP rules we configured in the testbed for **Set S1**.

Our goal is to collect event logs implying user-specific TAP logic and physical correlations in realistic scenarios. To avoid physical device limitations and complex setup procedures, we build a virtual platform where users deploy virtual devices based on specific smart home layouts. This hybrid platform uses sensor datasets from existing research as input to represent user behaviors. It also supports TAP logic configuration and connects with real smart home platforms to simulate user-specific TAP logic and physical correlations.

The architecture of our hybrid testbed is illustrated in Figure 4. The testbed simulates/replays user physical behaviors in a smart home environment by combining real and virtual components. It takes physical sensor event logs as input, representing user behaviors in smart home environments. We configure reasonable TAP rules based on the physical environment layout. The rules are obtained from official public repositories (e.g., official SmartApps from the SmartThings GitHub repository [41] and official IFTTT applets from IFTTT market [25], [42]) and open-source datasets from previous research [5], [6], [26]. The testbed traverses the input event log (① in Figure 4), sends state-change events to the related virtual/real devices (②), triggers TAP logic (③), generates action events (④), and inserts them into the original log (⑤). The final output is an event log containing both user behavior events and rule-triggered events, reflecting the configured TAP logic and physical correlations. We integrate our hybrid testbed with the open-source Home Assistant platform to control and monitor IoT devices, where we collect the event logs in a centralized manner.

Dataset Configuration - Different Scenarios. We use the hybrid testbed to generate different datasets (i.e., event logs) from different smart-home scenarios. We collect different home layouts and their sensors' logs from previous user behavior recognition studies [1], [4], [14]. Based on the home layout, we configure reasonable TAP rules in the testbed and collect the corresponding event logs. We generate four datasets (**Set S1**, **Set S2**, **Set S3** and **Set S4**) based on the real sensors' logs. We explain how we generate **Set S1** as follows and illustrate the details of other datasets in Appendix B.

For **Set S1**, we use the sensors' event log of the HH130 apartment from WSU CASAS datasets [4], [14], whose layout is illustrated in Figure 5. There are three kinds of physical sensors in this home scenario: 7 Motion Sensors (M), 1 Door Sensor (D), and 4 Temperature Sensors (T). We add different virtual actuators to enable home automation: 4 Virtual Lights (L), 2 Virtual Fans (Fan), and a Virtual Security System.

The rule sets we configured in this smart home scenario are illustrated in Figure 6. The principles of our rule configuration are as follows. All the reference SmartApps are from the public official (vetted) SmartThings GitHub repository [41].

- We virtualize the light devices and use the motion sensors to trigger them, which is the same as the "Light Follows Me" SmartApp. The near sensors will trigger the same light for the complex rules. (Rule 1-8)
- We virtualize the home mode, which has the "Home",

"Leave" and 'Sleep" modes. Following the "Scheduled Mode Change" SmartApp, we use the living experience to set up related rules. (Rule 9-12)
- We include a rule that will turn off all appliances when the home mode changes to "Sleep", consistent with the intention of the "Good Night House" SmartApp. However, the user may configure the security system as one of the appliances. (Rule 13)
- We virtualize the fan devices and use the temperature sensors to trigger them, obeying the "It's Too Hot" and "It's Too Cold" SmartApps. Specifically, we set up a complex rule to compare the indoor and outdoor temperatures based on the logic from the "Whole House Fan" SmartApp. (Rule 14-17)

We generate the fuzzing sensors' log for the home layout in the HH130 and HH125 apartments in the CASAS dataset. We enumerate different state permutations of all sensors in the environment as the sensors' event sequence to trigger the configured rules without any omissions. The datasets generated based on the fuzzing sensors' log in the HH130 and HH125 apartments are **Set S5** and **Set S6**.

Dataset Configuration - Different Number of Rules. We collect official SmartApps from the SmartThings GitHub repository [41] and openHAB rules from users' GitHub repositories [5], [16]. We only include SmartApps that contain explicit TAP logic and exclude those that serve as authentication proxies to other IoT platforms. In total, we collect 84 SmartApps (114 rules) and 21 openHAB rules.

We deploy different numbers of SmartThings rules (ranging from 10 to 110, with increments of 10) on the hybrid testbed. For each deployment, we use the CASAS HH130 Apartment dataset as the user behavior log to generate event logs, named **Set L10**, **Set L20**, ..., **Set L110**, respectively. These datasets imply large-scale homogeneous rules. We then deploy all SmartApps and openHAB rules together on the hybrid testbed to generate a dataset with heterogeneous rules (**Set L135**).

B. Effectiveness on TAP Logic Inference

Choice of iteration and generation numbers. TAPPecker has two significant parameters: the number of iterations in one inference N and the number of maximum generations in one iteration M. We summarize different kinds of inferred results by TAPPecker with different iteration and generation configurations on **Set S1** in Table II. According to the comparison with the ground truth, we categorize our inferred results into three kinds:

- **Accurate**: The inferred condition of the action is the same as the ground truth (e.g., **IF** $M002 = ON$ **THEN** turn off $L001$).
- **Consistent**: The inferred condition is consistent with the ground truth, but it is incomplete, or it has redundant sub-conditions (e.g., **IF** $T103 \geq 25.96$ **THEN** turn off $Fan1$).
- **Inconsistent**: There is no related state transition from the ground truth (e.g., **IF** $D002 = Open \vee Mode = Home$ **THEN** turn off $SecuritySystem$). It should be noted that we have filtered out the candidate TAP logic, which violates the logs based on their fitness level. The concept of "inconsistent" is when the result differs from the ground truth but is not "incorrect" as all the inferred results comply with the logs.

TABLE II: Number of results in different categories inferred by TAPPecker with different iteration (i.e., *N*) and generation (i.e., *M*) configurations. N_T represents **t**otal number of inferred results. N_A, N_C and N_I represent the number of **a**ccurate, **c**onsistent and **i**nconsistent inferred results, respectively.

N	**1**	**5**	**10**	**5**	**2**	**1**	**5**	**10**
M	**100**	**100**	**100**	**200**	**500**	**1000**	**1000**	**1000**
N_T	18	18	**17**	16	17	17	17	18
N_A	5	7	**10**	8	7	7	6	7
N_C	8	7	**4**	6	6	7	7	6
N_I	5	4	**3**	2	4	3	4	5

Although the number of inferred results by different configurations is similar, the "accurate" results are increasing with more iterations and fewer generations. With more iterations, TAPPecker has more chances to start with different seeds. If we only apply one iteration with fewer generations, the candidate condition may not evolve adequately, leading to incomplete conditions. For example, TAPPecker with one iteration of 100 generations only infers the condition of "turn off $L002$" as $M004 = ON$. On the other hand, more generations may result in an "over-fit" problem: the inferred condition may be specific to a certain environment state, causing redundant conditions. For example, TAPPecker with ten iterations of 1000 generations infers the condition of "turn off $L002$" as $M003 = ON \vee M004 = ON \vee (L004 = ON \wedge D002 = Close \wedge SecuritySystem = OFF)$. Therefore, we choose ten iterations of 100 generations as the implementation choice for these simulated datasets.

Baseline Choice and Implementation. To evaluate the effectiveness of TAPPecker, we choose and implement the statistic-based approaches which also takes the event log as the input to infer/verify TAP logic. They generate hypothetical correlations between two events based on the heuristic knowledge (e.g., physical correlations from previous researches [18], [21], common TAP logic experiences, and device layouts) and perform hypothesis testing to verify hypothetical correlations using event logs. Given a hypothetical correlation from Event A to Event B, we perform forward searching on the event log to generate and check the testing cases. Specifically, we traverse event logs to find all Event A as the testing cases and check whether when Event A happens, Event B will happen in a certain time window. The checking part follows HAWatcher [21]'s procedure. We treat the testing cases as the sequence of independent Bernoulli trials and use the one-tail test [20] to evaluate each hypothetical correlation's correctness. In addition, we compare TAPPecker with Trace2TAP [52] using their GitHub repository implementation. Trace2TAP selects

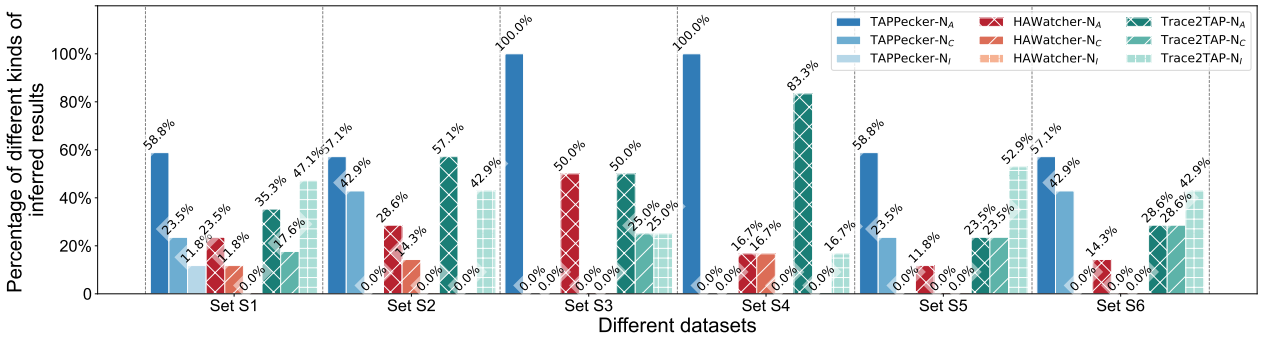


Fig. 7: Comparison results of TAPPecker, HAWatcher and Trace2TAP on datasets generated from different scenarios.

candidate triggers and conditions for target actions through statistical features (cross-correlation and conditional entropy) and validates them using symbolic reasoning and SAT-solving.

Inference Result Comparison and Analysis. We compare the inferred and the ground-truth conditions of different rules as shown in Figure 7. The inferred conditions contain the results of TAPPecker, HAWatcher and Trace2TAP. For the ground truth, we manually extract the conditions from the rule set.

We are taking the inferred results of **Set S1** as the examples (Table V in Appendix C), we analyze the performance of TAPPecker and the baseline based on the complexity of the rules.

- **Simple rules** refer to the rules with one trigger. We have five simple rules (Rules 1, 2, 7, 8, 13) in the configured rule set and extract five related transitions. All the inferred conditions of these rules by TAPPecker and the statistic-based approaches are accurate from the inferred results. We find that these approaches are effective in inferring simple rules.
- **Complex rules with discrete-state conditions** refer to the rules with multiple triggers related to devices with discrete states (e.g., motion sensors with ON and OFF states). The configured rule set has six complex rules (Rules 3-6, 11, 12). TAPPecker is more powerful, for it can infer the conditions the statistic-based approaches cannot obtain. For instance, TAPPecker infers the accurate condition $M003 = OFF \wedge M004 = OFF$ for the action "turn off $L002$". Such complex condition cannot be distinguished by HAWatcher's hypothetical testing or Trace2TAP's correlation analysis.
- **Complex rules with continuous-value conditions** refers to the rules with triggers composed of devices with continuous values (e.g., temperature sensors with numeric values). We extract four related rules (Rules 14-17). From the inferred results of the corresponding rules, we find that the statistic-based approach cannot infer these conditions, while TAPPecker performs well in this scenario. On the one hand, the "correlation" representation of HAWatcher is coarse-grained for the numeric devices. It treats them as binary devices by assigning the value into "High"/"Low" states, making them incapable of handling complex conditions with numeric devices. On the other hand, Trace2TAP cannot select the correct candidates when they are numeric devices. Thus, they cannot generate correct TAP rules. Meanwhile, TAPPecker provides a broader and more consistent condition range (i.e., $T103 \geq 25.96$). In addition, TAPPecker can learn more complex conditions (e.g., numerical calculation and comparison). Although it cannot infer the exact condition $(T106 - T102) \geq 3$ for the action "turn on $Fan2$", its inferred result implies the relationship between the values of $T106$ and $T102$.
- **User behavior patterns** indicate that the user's routine may cause implicit dependencies among different devices, which cannot be extracted from the rule set. For example, TAPPecker infers the rule "**IF** $D002 = Open \vee Mode = Home$ **THEN** turn on $SecuritySystem$". This is because after the security system is turned off mistakenly, the user may notice it and turn it on again in the real scenario. Such behavior happens when the user is at home ($Mode = Home$), or the user will leave home and perform the security checking ($D002 = Open$). Also, considering the user behavior, the sub-condition $M001 = ON$ for the action "set $Mode$ as $Home$" is reasonable. From the home layout, we can find that $M001$ is located nearby $D002$. So TAPPecker captures the actual event sequence that the door is open, the motion sensor ($M001$) detects movement, and the home mode changes to "Home".

Specifically, the trigger and action of the rule 11 never happen in the event log, so it is natural that TAPPecker does not infer the corresponding condition, nor by the statistic-based approach.

Since TAPPecker represents the condition as the boolean polynomial logic expression, the inferred results show that TAPPecker effectively mines the implied TAP logic from the execute traces and outperforms the statistic-based approaches by the TAP logic representation and the inferred results.

The Stability of TAPPecker. We run TAPPecker 50 times on different large-scale datasets. Figure 13 in Appendix D illustrates the number of different kinds of TAP logic inferred by TAPPecker from 50 runs on **Set L110**. Figure 8 presents the

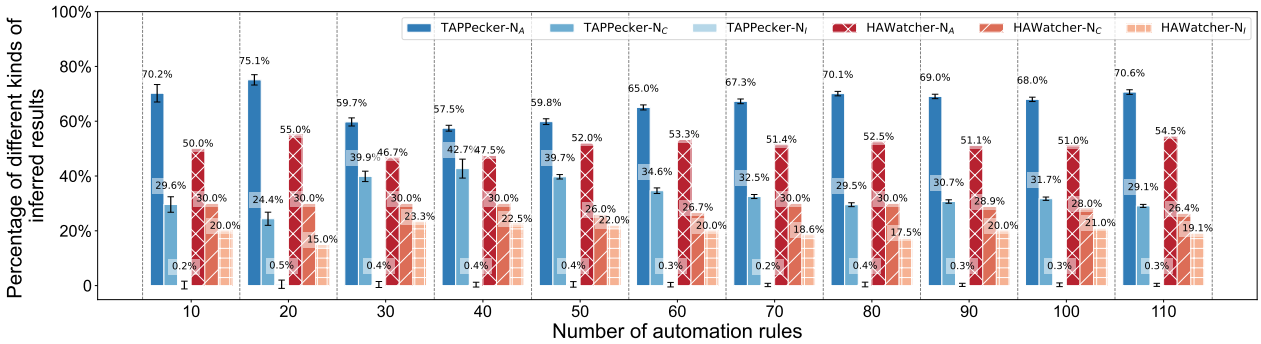


Fig. 8: Comparison results of TAPPecker and HAWatcher performed on logs implying homogeneous rules (**Set L10-Set L110**).

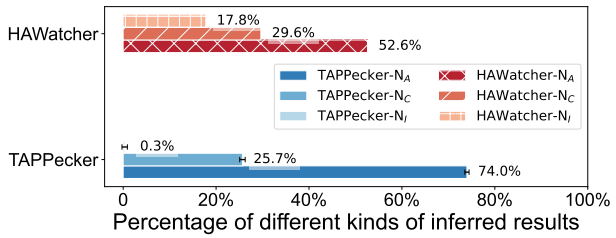


Fig. 9: Comparison results of TAPPecker and HAWatcher performed on logs implying heterogeneous rules (**Set L135**).

average number and standard deviation of different TAP logic types inferred from 50 runs on homogeneous rule datasets (**Set L10-Set L110**). The results demonstrate that different initial seeds produce nearly identical inferred TAP logic, confirming that TAPPecker is highly stable.

Large-scale Comparison. We perform rule inference using TAPPecker and HAWatcher on the generated log. Figure 8 shows the comparative results of rule inference after configuring different numbers of SmartApps (**Set L10-Set L110**), while Figure 9 shows the comparative results in a heterogeneous experimental environment (**Set L135**) between TAPPecker and HAWatcher.

TAPPecker exhibits consistent inference results without any inconsistencies. The majority of the inferred results are accurate. In cases where TAPPecker produces inaccurate but consistent inference results, they mainly involve numeric comparisons. While the comparison relationships remain consistent, the specific numerical values may not match exactly with the original SmartApp.

The validation results of HAWatcher reveal numerous inconsistencies, which highlight two key limitations of HAWatcher. First, HAWatcher can only categorize continuous numerical values into two states, namely "High" and "Low," for numerical devices. It cannot accurately infer rules encompassing the states of numerical devices. Second, HAWatcher heavily relies on the completeness of the log data to determine the existence of device correlations. If a particular device correlation appears infrequently in the log, HAWatcher may not recognize its existence.

> **Answer to RQ1:** Compared to the state of the art, the accuracy of our method has increased by 40.85%, and there are no inconsistencies with the configured TAP logic in the inference results, owing to our fine-grained TAP representation. Meanwhile, TAPPecker ensures to effectively detect and infer the TAP logic whenever it occurs in the log data, regardless of its frequency or occurrence.

C. Application of Property Violation Checking

Sufficiency of the inferred results. We further perform the Property Violation Checking module based on the inferred results. According to the simulation scenario of **Set 1**, we define two security/safety policies. We describe how they will fit the scenario and the configured rules as follows:

- **Policy 1**: The security system should always be on ($\mathbf{G}(s_{ON}^{SecuritySystem})$). It is a common security policy from the previous approach [7]. However, we intentionally misconfigured the security system as one of the appliances in Rule 13, so it will be turned off when the home mode changes to "Sleep", violating this policy.
- **Policy 2**: All four lights should never be turned on when the home mode is Leave ($\neg\mathbf{F}(s_{Leave}^{Mode} \wedge s_{ON}^{L001})$)[2]. We set up this energy-saving policy. The "Leave" mode indicates no human movement in the environment. Also, we do not include other devices that may trigger motion sensors (e.g., robot vacuum) in this scenario. So even without mode constraints in related rules (i.e., Rule 1, 3, 5, 7), the system will not violate this policy.

We transform them into five LTL formulas. The outcome of property violation checking shows that the automaton modeled based on our inferred results does violate **Policy 1**, which aligns with our expectations. Meanwhile, we cannot discover this policy violation if we use the inferred results from statistic-based approaches because they cannot infer the

[2]Each of the four lights has a corresponding LTL formula. Here, we omit the remaining three.

TABLE III: Violation results performed on the market set compared with previous approaches.

	A1	A2	A3[a]	TAPPecker
Violations	**10**	**10**	**12**	**12**
against time-related policies	0	0	2	2
against general policies	10	10	10	10

[a]A1 - IoTGuard; A2 - TAPInspector; A3 - AutoTap.

related TAP logic, proving the necessity of an accurate TAP logic inference approach.

Answer to RQ2: The TAP logic inferred by TAPPecker is applicable for model checking. Since the logic inferred by TAPPecker is consistent with the configured logic, the results of model checking using the inferred results are the same as the original results.

Correctness of the checking tool. Based on the inferred TAP logic from the large-scale dataset (**Set L110**), we compare the correctness of our Violation Checking module and other model checking approaches, including IoTGuard [7], TAPInspector [48] and AutoTap [51]. We define 25 safety/security policies and translate them into 35 LTL formulas, extending from the previous approaches [7], [48], [51]. The checking results are summarized in Table III, and the details are shown in Table VI in Appendix E. We find two unique time-related violations that IoTGuard and TAPInspector dismiss because their policy syntax cannot represent time-related policies. IoTGuard cannot find two more violations because these policies are not included in their considerations. According to our experiment, the templates provided by AutoTap effectively identify time-related violations. However, these violations were not reported due to the lack of defined related properties and the absence of a large-scale evaluation of the public rule sets.

Answer to RQ3: Using the security policy templates we define, we can identify new time-related security violations in large-scale TAP logic sets that have not been considered or reported in previous approaches.

D. Timing Overhead

We measure the time to infer the rules using TAPPecker with various configurations in **Set S1**. The results in Table IV show that TAPPecker has an increasing timing overhead when both iterations (i.e., N) and generations (i.e., M) are increased. The results also show that having more generations incurs a huge timing overhead for rule inference. The chosen configuration (i.e., ten iterations of 100 generations) for the simulation dataset is close to the median of the timing overhead among all configurations.

TABLE IV: The average timing overhead to infer each rule of one iteration with different iteration (i.e., N) and generation (i.e., M) configurations.

N	M	**Average Runtime (seconds)**
1	100	2.46
5	100	3.14
10	**100**	4.28
5	200	7.25
2	500	26.19
1	1000	27.75
5	1000	45.66
10	1000	44.30

Answer to RQ4: The time cost of inferring a rule with TAPPecker is at the second level and increases with the enlargement of iterations and generations. TAPPecker adopts an evolutionary approach to avoid assuming and testing all possible candidates, which improves time efficiency.

V. Discussion

Integration with Downstream Security Analysis Approaches. TAPPecker reconstructs the inaccessible TAP logic of closed-source smart-home platforms. Its output provides complete and accurate TAP logic for downstream security analyses. Our evaluation demonstrates that the inferred TAP logic serves as valid input to other model checking approaches (e.g., IoTGuard [7], TAPInspector [48], and AutoTap [51]) without compromising correctness. Since TAPPecker's TAP logic representation is semantically equivalent to the input required by other security analysis approaches, it can also support anomaly detection, provenance analysis, and access control methods.

Physical Channel Interaction Identification. TAPPecker captures correlations induced by physical channels, provided that the corresponding interactions are reflected in the logs. This capability enables IoTSafe [18] and IoTSeer [37] to build upon TAPPecker's output for finer-grained physical modeling while avoiding their costly preprocessing, such as grid search.

Log Completeness. TAPPecker can effectively detect and infer TAP logic whenever such logic appears in the log data, regardless of its frequency, which other log-based approaches cannot achieve. However, TAPPecker cannot discover TAP rules that have never been triggered in the log. Log completeness is therefore a prerequisite for TAPPecker's accuracy. We will consider incorporating techniques such as dynamic fuzzing to systematically trigger all TAP logic during log collection in future work.

Policy Specification. Incomplete policies also impact violation identification results. In our experiment (shown in Appendix E), we find that IoTGuard cannot detect two violations because the defined policies are incomplete. To address this

issue, TAPPecker compiles a comprehensive set of security policies from existing model checking approaches [7], [18], [37], [48], [49], [51]. Each policy is presented to users in natural language and as an LTL formula. Users may adopt the entire set or select policies relevant to their homes. Enforcing all policies is stricter and may generate false alarms. While this may not align with user expectations, such alerts still provide valuable insights. Users can also customize policies, which TAPPecker automatically translates into LTL representations.

VI. Related Work

Dependency Extraction. The vast majority of previous research extract automation rules (i.e., intra-app dependency) by source code analysis. SAINT [5] translates IoT app source code into an intermediate representation and performs taint analysis to track information flow from sensitive sources to sink. Soteria [6] also performs the IR translation for the IoT apps and extracts the state machines of apps. IoTSan [35] translates apps written in Groovy (the programming language of SmartThings apps) into Promela as part of the whole system model. IoTMon [17] performs an intra-app interaction analysis using static program analysis to extract necessary application information for building intra-app interactions. Aiming at IFTTT applets, iRuler [45] parses them to extract trigger-action rules and transform them into a self-defined rule representation. HomeGuard [9] proposes to symbolically execute the app, exploring its execution paths to guarantee complete and precise rules.

In addition to intra-dependency, previous research has also worked on capturing the direct interactions among IoT apps. IoTSan [35] identifies each app's input/output events and connects them as an app dependency graph. iRuler [45] infers inter-rule information flows using Natural Language Processing (NLP) and constructs an information flow graph. IoTGuard [7] instruments IoT apps to monitor and collect app information at runtime and use a dynamic model to represent the runtime execution behaviors. HomeGuard [9] defines formats of different Cross-App Interference threats and employs a constraint solver to evaluate the relation between rules.

Furthermore, researchers pay more attention to implicit interactions (i.e., physical effects and user behaviors). IoTMon [17] identifies physical channels of IoT apps by analyzing application descriptions with NLP techniques and generates inter-app interaction chains by connecting intra-app interactions through physical and system. IoTSafe [18] proposes a physical interaction discovery approach by dynamic testing to capture real physical interactions among IoT devices. HAWatcher [21] generates hypothetical correlations from smart app, physical, and user activity channels and verifies them with event logs. Trace2TAP [52] identifies top variable candidates most suitable for trigger and condition propositions based on cross-correlation and conditional entropy. It then formulates a symbolic constraint using these identified variables and applies the constraint to a solver to generate all constraint-satisfying TAP rules.

The main difference between these existing dependency discovery approaches and TAPPecker is that they all assume that the source code of smart home apps/applets is available and even instrumentable. However, as the motivating example illustrates, a heterogeneous smart home system does not meet the basic assumption. TAPPecker provides an effective solution for inferring dependencies from the executed traces (i.e., event logs) without heuristic knowledge. Meanwhile, statistical approaches [21], [52] that synthesize/validate TAP logic from event logs fail to handle multi-condition TAP logic due to their simple rule representations and ineffective statistical features.

Property Violation Checking. Several methods evaluate the execution of IoT apps against user-defined safety/security policies in the IoT environment. In general, based on their chosen model-checking tools, they represent IoT apps and policies in a specific way and perform model-checking on them. For example, IoTSan [35] uses the Promela model to represent IoT apps and policies and checks them with the Spin model checker. iRuler [45] represents IoT apps by rewriting logic and uses Rewriting modulo SMT to check violations. Because IoTGuard [7] connects devices' transition as a dynamic model, it applies reachability analysis to check whether the model reaches an unexpected/wrong situation. SOATERIA [6], AutoTap [51] and TAPInspector [48] model the IoT apps into the automaton/state model, translate policies to formulas in linear temporal logic (LTL) and use the NuSMV tool to perform model checking. Yu et al. [49] present TAPFixer, an automatic framework for detecting and repairing vulnerabilities in home automation systems. TAPFixer employs a novel negated-property reasoning algorithm to generate patches that eliminate rule interaction vulnerabilities in both logical and physical spaces, enhancing the security and reliability of home automation systems.

VII. Conclusion

In this paper, we propose TAPPecker, a self-adaptive technique to infer complex TAP logic from event logs. We demonstrate its capabilities to infer TAP logic through the interaction of heterogeneous smart home devices in the smart home system. Compared to the statistical-based inference approaches, we have shown that TAPPecker provides substantial improvements to infer complex TAP logic in a runtime environment. We also propose a model-checking mechanism to formally identify safety or security property violations based on the inferred TAP logic. TAPPecker provides the capabilities to identify violated policies during runtime without knowing the predefined trigger-action rules.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (No. 62172027, No. U24B20117) and the Zhejiang Provincial Natural Science Foundation of China (No.LZ23F020013).

References

[1] Hande Alemdar, Halil Ertan, Ozlem Durmaz Incel, and Cem Ersoy. Aras human activity datasets in multiple homes with multiple residents. In *2013 7th International Conference on Pervasive Computing Technologies for Healthcare and Workshops*, pages 232–235. IEEE, 2013.

[2] Apple. Home. https://www.apple.com/ios/home/.

[3] Home Assistant. https://www.home-assistant.io/.

[4] CASAS. Smart home data sets. http://casas.wsu.edu/datasets/.

[5] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1687–1704, 2018.

[6] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, 2018.

[7] Z Berkay Celik, Gang Tan, and Patrick D McDaniel. Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In *NDSS*, 2019.

[8] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Data privacy in trigger-action systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 501–518. IEEE, 2021.

[9] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-app interference threats in smart homes: Categorization, detection and handling. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 411–423. IEEE, 2020.

[10] Jiwon Choi, Hayoung Jeoung, Jihun Kim, Youngjoo Ko, Wonup Jung, Hanjun Kim, and Jong Kim. Detecting and identifying faulty iot devices in smart home with context extraction. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 610–621. IEEE, 2018.

[11] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International conference on computer aided verification*, pages 359–364. Springer, 2002.

[12] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer, 1999.

[13] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*, pages 52–71. Springer, 1981.

[14] Diane J Cook, Aaron S Crandall, Brian L Thomas, and Narayanan C Krishnan. Casas: A smart home in a box. *Computer*, 46(7):62–69, 2012.

[15] Diane J Cook, Michael Youngblood, Edwin O Heierman, Karthik Gopalratnam, Sira Rao, Andrey Litvin, and Farhan Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003.(PerCom 2003).*, pages 521–524. IEEE, 2003.

[16] Thomas Dietrich. openhab-config. https://github.com/ThomDietrich/openhab-config/tree/master/rules.

[17] 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*, pages 832–846, 2018.

[18] 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)*, 2021.

[19] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[20] Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pages 66–70. Springer, 1992.

[21] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Hawatche:semantics-aware anomaly detection for appified smart homes. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4223–4240, 2021.

[22] 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*, 14(10):2607–2622, 2019.

[23] Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 215–225, 2015.

[24] IFTTT. Every thing works better together. https://ifttt.com/.

[25] IFTTT. Smartthings integrations. https://ifttt.com/smartthings.

[26] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*, volume 2, pages 2–2. San Diego, 2017.

[27] Palanivel A Kodeswaran, Ravi Kokku, Sayandeep Sen, and Mudhakar Srivatsa. Idea: A system for efficient failure management in smart iot environments. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 43–56, 2016.

[28] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4:87–112, 1994.

[29] Leslie Lamport. "sometime" is sometimes "not never" on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–185, 1980.

[30] Hai Lin, Chenglong Li, J Lang, Zhiliang Wang, Linna Fan, and Chenxin Duan. Cp-iot: A cross-platform monitoring system for smart home. In *Network and Distributed System Security (NDSS) Symposium*, 2024.

[31] Kulani Mahadewa, Yanjun Zhang, Guangdong Bai, Lei Bu, Zhiqiang Zuo, Dileepa Fernando, Zhenkai Liang, and Jin Song Dong. Identifying privacy weaknesses from multi-party trigger-action integration platforms. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 2–15, 2021.

[32] Jian Mao, Qixiao Lin, Shishi Zhu, Liran Ma, and Jianwei Liu. Smarttracer: Anomaly-driven provenance analysis based on device correlation in smart home systems. *IEEE Internet of Things Journal*, 11(4):5731–5744, 2023.

[33] M Hammad Mazhar, Li Li, Endadul Hoque, and Omar Chowdhury. Maverick: An app-independent and platform-agnostic approach to enforce policies in iot systems at runtime. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 73–84, 2023.

[34] Microsoft. Power automate. https://powerautomate.microsoft.com/.

[35] 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*, pages 191–203, 2018.

[36] openHAB. https://www.openhab.org/.

[37] 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*, pages 2415–2428, 2022.

[38] Amir Rahmati, Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Ifttt vs. zapier: A comparative study of trigger-action programming frameworks. *arXiv preprint arXiv:1709.02788*, 2017.

[39] Phillip Rieger, Marco Chilese, Reham Mohamed, Markus Miettinen, Hossein Fereidooni, and Ahmad-Reza Sadeghi. Argus:context-based detection of stealthy iot infiltration attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4301–4318, 2023.

[40] SmartThings. One simple home system. a world of possibilities. https://www.smartthings.com/.

[41] SmartThings. Smartthings official app repository. https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/smartapps.

[42] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1501–1510, 2017.

[43] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 803–812, 2014.

[44] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3227–3231, 2016.

[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*, pages 1439–1453, 2019.

[46] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*, 2018.

[47] Jingyu Xiao, Zhiyao Xu, Qingsong Zou, Qing Li, Dan Zhao, Dong Fang, Ruoyu Li, Wenxin Tang, Kang Li, Xudong Zuo, et al. Make your home safe: Time-aware unsupervised user behavior anomaly detection in smart homes via loss-guided mask. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3551–3562, 2024.

[48] 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.

[49] Yinbo Yu, Yuanqi Xu, Kepu Huang, and Jiajia Liu. Tapfixer: Automatic detection and repair of home automation vulnerabilities based on negated-property reasoning. In *33th USENIX security symposium (USENIX Security 24)*, 2024.

[50] Zapier. Automation that move you forward. https://zapier.com/.

[51] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, 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.

[52] Lefan Zhang, Weijia He, Olivia Morkved, Valerie Zhao, Michael L Littman, Shan Lu, and Blase Ur. Trace2tap: Synthesizing trigger-action programs from traces of behavior. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):1–26, 2020.

APPENDIX A
ALGORITHM DETAILS

Function `randomTree`, `crossover` and `mutate` in Algorithm 1 is defined in Algorithm 3, 4 and 5.

Algorithm 3: Function `randomTree` in Algorithm 1

Function `randomTree` $(depth, \mathcal{D}, prob_t)$ **:**
- generate a random number n in $[0, 1]$
- **if** $n < prob_t$ **and** $depth > 0$ **then**
  - `// tree node`
  - randomly choose a logic connector from $\{and, or\}$ as f
  - $child_l, child_r \leftarrow randomTree(depth-1, \mathcal{D}, prob_t)$
  - form $f, child_l, child_r$ as a tree $newTree$
  - **return** $newTree$
- **else**
  - `// condition node`
  - randomly choose a device from $\mathcal{D}$
  - randomly generate the corresponding logic operator and value
  - form the logic operator, device and value as a tree $newTree$
  - **return** $newTree$

APPENDIX B
DETAILS OF OTHER DATASETS

We explain how we generate **Set S2**, **Set S3**, **Set S4** as follows.

For **Set S2**, we use the sensors' event log of the HH125 apartment from WSU CASAS datasets [4], [14], whose layout is illustrated in Figure 10a. There are 3 kinds of physical sensors in this home scenario: 8 Motion Sensors (M), 2 Door Sensor (D) and 6 Temperature Sensors (T). To enable the home automation, we add different virtual actuators: 4 Virtual Lights (L) and 3 Virtual Fans (Fan). The rule sets we configured in this smart home scenario are illustrated in Figure 10b.

For **Set S3**, we use the sensors' event log of House A from the ARAS dataset [1], whose layout is illustrated in Figure 11a. There are 6 kinds of physical sensors in this home scenario: 3 Contact Sensors (CO), 3 Force Sensors (FO), 6 Photocells (PH), 6 Distance sensors (DI/SO), 1 temperature sensor (TE) and 1 IR receiver (IR). To enable the home automation, we add different virtual actuators: 2 Virtual Lights (L) and 1 Virtual Fans (Fan). The rule sets we configured in this smart home scenario are illustrated in Figure 11b.

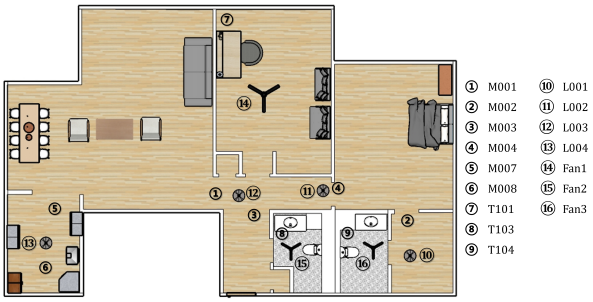
For **Set S4**, we use the sensors' event log of House B from the ARAS dataset [1], whose layout is illustrated in Figure 12a.

Algorithm 4: Function `crossover` in Algorithm 1

Function `crossover` $(tree_1, tree_2, prob_c)$ **:**
- generate a random number n in $[0, 1]$
- **if** $n < prob_c$ **and** *depth¿0* **then**
  - `// do not corssover`
  - **return** $tree_2$
- **else**
  - `// crossover`
  - $newTree \leftarrow tree_1$
  - **if** $tree_1$ *and* $tree_2$ *both have children* **then**
    - **foreach** $child_1 \in tree_1.children$ **do**
      - randomly choose one child of $tree_2$ as $child_2$
      - $newTree.children \leftarrow crossover(child_1, child_2, prob_c)$
  - **return** $newTree$

Algorithm 5: Function `mutate` in Algorithm 1

Function `mutate` $(tree, \mathcal{D}_{cand}, prob_m)$ **:**
- generate a random number n in $[0, 1]$
- **if** $n < prob_m$ **then**
  - `// mutate`
  - **if** $tree$ *has children* **then**
    - `// for the tree node, generate a new subtree`
    - **return** $randomTree(depth, \mathcal{D}_{cand}, prob_t)$
  - **else**
    - `// for the terminal node, change its logic operator or value`
    - randomly change the logic operator or the value of $tree$
    - **return** $tree$
- **else**
  - $newTree \leftarrow tree$
  - **if** $tree$ *has children* **then**
    - **foreach** $child \in tree.children$ **do**
      - $newTree.child \leftarrow mutate(child, \mathcal{D}_{cand}, prob_m)$
  - **return** $newTree$

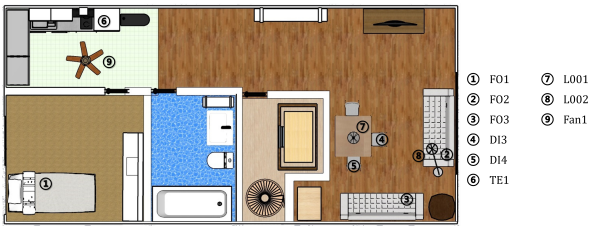


(a) The home layout with different devices for simulation.

#	IF	Then
1	**M002** ON	**L001** ON
2	**M002** OFF	**L001** OFF
3	**M004** ON	**L002** ON
4	**M004** OFF	**L002** OFF
5	**M001** ON or **M003** ON	**L003** ON
6	**M001** OFF and **M003** OFF	**L003** OFF
7	**M007** ON or **M008** ON	**L004** ON
8	**M007** OFF and **M008** OFF	**L004** OFF
9	**T101** >= 32 ℃	**Fan1** ON
10	**T101** < 31 ℃	**Fan1** OFF
11	**T103** >= 25 ℃	**Fan2** ON
12	**T103** < 24 ℃	**Fan2** OFF
13	**T104** >= 25 ℃	**Fan3** ON
14	**T104** < 24 ℃	**Fan3** OFF

(b) The TAP rules we configured in the testbed for simulation.

Fig. 10: Dataset configuration for **Set S2**.



(a) The home layout with different devices for simulation.

#	IF	Then
1	**DI3** ON or **DI4** ON	**L001** ON
2	**DI3** OFF and **DI4** OFF	**L001** OFF
3	**FO2** ON	**L002** ON
4	**FO2** OFF	**L002** OFF
5	**TE1** ON	**Fan1** ON
6	**TE1** OFF	**Fan1** OFF
7	**FO1** ON and **FO3** ON	**Mode** changes to "Sleep"
8	**FO1** OFF or **FO3** OFF	**Mode** changes to "Home"

(b) The TAP rules we configured in the testbed for simulation.

Fig. 11: Dataset configuration for **Set S3**.

There are 5 kinds of physical sensors in this home scenario: 6 Contact Sensors (CO), 3 Force Sensors (FO), 2 Photocells (PH), 5 Pressure Mats (PR) and 4 Distance sensors (DI/SO). To enable the home automation, we add 5 Virtual Lights (L). The rule sets we configured in this smart home scenario are illustrated in Figure 12b.

(a) The home layout with different devices for simulation.

#	IF	Then
1	**FO1** ON or **FO2** ON	**L001** ON
2	**FO1** OFF and **FO2** OFF	**L001** OFF
3	**PR5** ON	**L002** ON
4	**PR5** OFF	**L002** OFF
5	**PR1** ON or **PR2** ON	**L003** ON
6	**PR1** OFF and **PR2** OFF	**L003** OFF
7	**CO1** ON or **CO2** ON	**L004** ON
8	**CO1** OFF and **CO2** OFF	**L004** OFF
9	**CO4** ON or **CO5** ON	**L005** ON
10	**CO4** OFF and **CO5** OFF	**L005** OFF
11	**PR3** ON and **PR4** ON	**Mode** changes to "Sleep"
12	**PR3** OFF or **PR4** OFF	**Mode** changes to "Home"

(b) The TAP rules we configured in the testbed for simulation.

Fig. 12: Dataset configuration for **Set S4**.

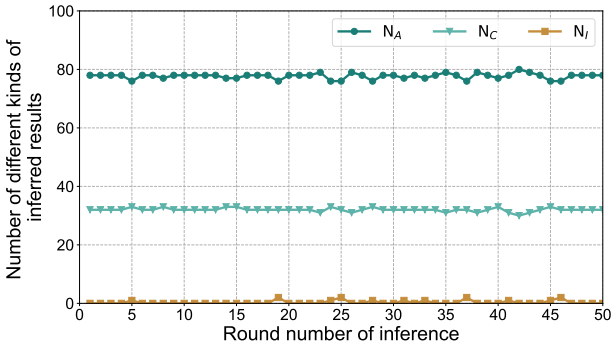


Fig. 13: Number of different kinds of rules inferred by TAPPecker from 50 runs on the log implying 110 rules (**Set L110**).

Appendix C
Detailed comparison between our inferred results and ground-truth

Table V illustrates the inferred results of TAPPecker and baselines and the ground-truth we extract manually from the rule set for **Set S1**.

Appendix D
Stability Experiment Result

Figure 13 illustrates the number of different kinds of TAP logic inferred by TAPPecker from 50 runs on **Set L110**.

TABLE V: The comparison of the ground-truth and inferred TAP logic on **Set S1**.

Action	Condition			
	Ground-truth	**TAPPecker**	**HAWatcher**	**Trace2TAP**
Turn on *L001*	*M002 = ON*	*M002 = ON*	*M002 = ON*	*M002 = ON*
Turn off *L001*	*M002 = OFF*	*M002 = OFF*	*M002 = OFF*	*M002 = OFF*
Turn on *L002*	*M003 = ON* $\vee$ *M004 = ON*	*M003 = ON* $\vee$ *M004 = ON*	-	*M003 = ON* $\vee$ *M004 = ON*
Turn off *L002*	*M004 = OFF* $\wedge$ *M003 = OFF*	*M004 = OFF* $\wedge$ *M003 = OFF*	-	*M003 = OFF*
Turn on *L003*	*M005 = ON* $\vee$ *M006 = ON*	*M005 = ON* $\vee$ *M006 = ON*	*M005 = ON* *M006 = ON*	*M005 = ON* $\vee$ *M006 = ON*
Turn off *L003*	*M005 = OFF* $\wedge$ *M006 = OFF*	*M005 = OFF* $\wedge$ *M006 = OFF*	*M005 = OFF* *M006 = OFF*	*M005 = OFF*
Turn on *L004*	*M011 = ON*	*M011 = OFF*	*M011 = OFF*	*M011 = OFF*
Turn off *L004*	*M011 = OFF*	*M011 = OFF*	*M011 = OFF*	*M011 = OFF*
Turn on *Fan1*	*T103* ≥ 26	*T103* ≥ 25.96	-	-
Turn off *Fan1*	*T103* < 26	*T103* ≤ 25.3	-	-
Turn on *Fan2*	*T106 - T102* ≥ 3	(*T102* $\leq 27.28 \wedge$ *T106* $\geq 28.50) \vee$ (*T106* ≥ 30.97)	-	-
Turn off *Fan2*	*T106 - T102* < 3	(*T102* $\geq 27.3 \wedge$ *T106* $\leq 30.62) \vee$ (*T102* $\geq 26.64 \wedge$ *T106* $\leq 25.49) \vee$ (*T106* ≤ 28.72)	-	-
Turn on *SecuritySystem*	-	*D002 = Open* $\vee$ *Mode = Home*	-	-
Turn off *SecuritySystem*	*Mode = Sleep*	*Mode = Sleep*	-	-
Change *Mode* from *Home* to *Leave*	*D002 = Open*	*D002 = Open*	-	-
Change *Mode* from *Leave* to *Home*	*D002 = Open*	*M001 = ON* $\wedge$ *D002 = Close*	-	*D002 = Open*
Change *Mode* from *Sleep* to *Leave*	*D002 = Open*	-	-	-
Change *Mode* from *Home* to *Sleep*	-	*M001 = OFF* $\vee$ *L002 = ON* $\vee$ *M004 = ON*	-	-

APPENDIX E
SAFETY/SECURITY POLICIES

Table VI shows how TAP logic from the market set combined with each other violate safety/security policies.

TABLE VI: TAP logic combinations of the market set and their violated policies. "-" represents that the specific policy is violated because there are no related TAP logic. "●" and "○" indicate that this violation can/cannot be identified by the corresponding approach, respectively. "◐" indicates that the policy is not defined in the corresponding approach, but can be represented by its policy syntax and then be identified as violation.

TAP logic (source SmarApp)	Violated Policy	A1	A2	A3	A4[a]
$Outlet.On \xrightarrow{Time=45min} Outlet.Off$ ("Curling Iron")	The curling iron should be on for a maximum of 30 minutes.	○	○	●	●
$Switch.On \xrightarrow{Time=20min} Switch.Off$ ("Power Allowance")	The electric iron should be on for a maximum of 15 minutes.	○	○	●	●
$GarDoor.Open \xrightarrow{PS=Off} GarDoor.Close$ ("ridiculously-automated-garage-door") $GarDoor.Close \xrightarrow{App=Touched} GarDoor.Open$ ("garage-door-opener")	The garage door should only be open when people are at home, and it should always be closed when people leave home.	◐	●	●	●
$Valve.On \xrightarrow{WS=Wet} Valve.Off$ ("close-the-valve") $Valve.Off \xrightarrow{SpTime=On} Valve.On$ ("sprayer-controller-2")	The valve should always be closed when water sensor is wet and when the water level threshold specified by a user is reached.	◐	●	●	●
$Mode.Home \xrightarrow{App=Touched} Mode.Sleep$ ("good-night-house") $MP.Pause \xrightarrow{MButton=Pushed} MP.Play$ ("sonos-remote-control")	The devices (e.g., music player) should never be open or turned on when the user is not at home or sleeping.	●	●	●	●
$Mode.Home \xrightarrow{PS=Off} Mode.Away$ ("bon-voyage") $AC.Off \xrightarrow{InTemp>35} AC.On$ ("its-too-hot")	Some device functionality (e.g. turning on AC and heater) should never be used when the user is not at home or before a time specified by a user.	●	●	●	●
$Heater.Off \xrightarrow{InTemp<10} Heater.On$ ("its-too-cold") $AC.Off \xrightarrow{InTemp>35} AC.On$ ("its-too-hot")	The AC and heater should never not be on at the same time.	●	●	●	●
-	When there is smoke, the lights should always be on if it is night, and the door should always be unlocked.	●	●	●	●
-	The alarm should only sound when there is smoke or CO; and when an unexpected motion, tampering, and entering occurs.	●	●	●	●
-	The HVACs, fans, heaters should always be off when the temperature values are out of the threshold specified by the user.	●	●	●	●
-	The AC should always be on when a user is within a specified distance of the house or at a time specified by the user.	●	●	●	●
-	The windows should never not be open when the heater is on.	●	●	●	●

[a] A1 - IoTGuard; A2 - TAPInspector; A3 - AutoTap; A4 - TAPPecker.