This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



# SafeChain: Securing Trigger-Action Programming from Attack Chains

Kai-Hsiang Hsu, Yu-Hsi Chiang, Hsu-Chun Hsiao, *Member, IEEE*

**Abstract**—The proliferation of Internet of Things (IoT) is reshaping our lifestyle. With IoT sensors and devices communicating with each other via the Internet, people can customize *automation rules* to meet their needs. Unless carefully defined, however, such rules can easily become points of security failure as the number of devices and complexity of rules increase. Device owners may end up unintentionally providing access or revealing private information to unauthorized entities due to complex chain reactions among devices. Prior work on trigger-action programming either focuses on conflict resolution or usability issues, or fails to accurately and efficiently detect such *attack chains*. This paper explores security vulnerabilities when users have the freedom to customize automation rules using trigger-action programming. We define two broad classes of attack—*privilege escalation* and *privacy leakage* —and present a practical model-checking-based system called SafeChain that detects hidden attack chains exploiting the combination of rules. Built upon existing model-checking techniques, SafeChain identifies attack chains by modeling the IoT ecosystem as a Finite State Machine. To improve practicability, SafeChain avoids the need to accurately model an environment by frequently re-checking the automation rules given the current states, and employs rule-aware optimizations to further reduce overhead. Our comparative analysis shows that SafeChain can efficiently and accurately identify attack chains, and our prototype implementation of SafeChain can verify 100 rules in less than one second with no false positives.

**Index Terms**—Trigger-Action Attack Chains, Privilege Escalation, Information Leakage, Model Checking, Internet of Things

## 1 Introduction

We now live in an era with smart technologies that utilize connected devices and sensors to automatically adapt, enhance performance based on prior experience, and use reasoning to modify the next behavior [35]. According to a recent survey [6], around 8.4 billion networked devices are expected to be in use by 2017 and the projected number escalates to more than 20 billion by 2020. This speculation indicates that these Internet of Things (IoT) are already beginning to reshape our daily lifestyles seamlessly.

Similar to prior advancement in technology, IoT will bring convenience to our daily lives, at the cost of security and privacy. As IoT devices are tightly entangled with the physical world, an adversary in cyberspace can threaten human users' safety and privacy in the physical world via IoT devices. The lack of appropriate security mechanisms in IoT has already been highlighted in recent news, ranging from cyber incidents (e.g., hacking smart fridges to send spam emails [5], compromising smart meters to reduce power bills [16], and hijacking toys to leak information [8]) to detrimental cyber-physical threats (e.g., exploiting cardiac devices to induce inappropriate pacing or shocks [3], injecting a worm on IoT devices using ZigBee communication to launch a massive city-wide light disruptions [40], and compromising IoT devices to disrupt the power grid's normal operations [41]). As more and more vulnerabilities are discovered, relying on vendors to patch IoT devices in a timely manner is insufficient. Additional defenses must be in place to limit the impact on vulnerable devices.

An interesting feature of IoT is supporting customized interaction among devices using end-user programming, such as trigger-action programming [43]. This often takes the form of "if trigger, then action" and allows users to specify a *trigger* that represents a condition and the corresponding *action* to be taken whenever that trigger event occurs. Once defined, such trigger-action rules can be automatically applied without user involvement. As the number of connected devices multiplies[1], the complexity of interactions among them will also increase with customized automation. The increasingly complex interdependencies between devices can easily allow for various attacks, because an adversary controlling one IoT device can now expand influence to more devices through such interdependencies. Unfortunately, attacks leveraging trigger-action rules are difficult to detect manually, as device owners may unintentionally provide access or reveal private information to unauthorized entities due to complex chain reactions [46].

This work presents an automated *prevention* system called SafeChain which identifies exploitable trigger-action *attack chains*. SafeChain can thus work in conjunction with methods that support postmortem attack reconstruction from logs [46], methods that identify errors in individual rules [37], and methods that resolve conflicts between rules [30], [31], [32], [33].

We first formulate two classes of attack that exploit trigger-action rules. The first is *privilege escalation*, in which an adversary gains control of more devices than it initially has via automation rules. For instance, given the rule "if someone is home, turn on the light", an attacker who compromises the occupancy sensor can also affect the status of the lightbulb. The other attack class is *privacy leakage*, in which an adversary learns more information about the devices than it initially has via automation rules. For example, given the rule "if someone is home, turn on the light", an attacker who observes the state of the lighting device (e.g., the light is publicly observable or hacked) can infer the status of

• *K.-H. Hsu, Y.-H. Chiang and H.-C. Hsiao are with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, 106, Taiwan.*

*Manuscript received x x, 2018; revised x x, 2018.*

1. The number of connected IoT devices per household are anticipated to rise to 50 by 2020 [9].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



the occupancy sensor. In other words, turning the lighting device on and off with respect to the occupancy of the home leaks information to the adversary. The attacker can also leverage the combination of multiple rules to create a chain effect.

To efficiently and accurately identify the two attack classes, we present SAFECHAIN, a practical system built upon model-checking techniques and enhanced by domain specific optimizations. SAFECHAIN models the IoT ecosystem as a Finite State Machine (FSM)[2] such that finding an attack can be reduced to a reachability problem in the FSM. Both static and dynamic analysis techniques have been used in prior work to verify IoT automation. Static analysis [42] is often more efficient but comes with higher false positives as no runtime information is provided. On the other hand, prior work [30], [47] that similarly utilizes model-checking tools lacks a clear and detailed specification, or mostly focuses on resolving conflicts and making sure individual rules match user intent. Dynamic analysis, either FSM or symbolic execution [31], often suffers from scalability problems and needs a reliable method of modeling the execution environment. Therefore, we only consider dynamic analysis to be practical unless the following challenges have been overcome.

**Challenge 1: Environment modeling in FSM.** As model checking verifies properties against a given "model," an inaccurate model may miss detection or create false alarms. Accurately modeling environment variables (e.g., trajectory of a user and temperature) is nevertheless challenging because it requires extensive knowledge about physical environments. Instead of aiming to create an accurate environment model (e.g., using differential equations and control theory), SAFECHAIN relaxes the requirement by frequently re-calibrating a simple environment model based on the current state and the extrapolated near-future state. SAFECHAIN then re-checks (e.g., every 1s or when the current state changes) the automation rules given the updated model.

**Challenge 2: The state explosion problem in model checking.** The number of states in FSM grows exponentially with attributes. Given hundreds of rules (and device attributes), how can we accurately and efficiently detect vulnerable rules? In addition, to support frequent rechecking as stated in the first challenge, the verification should be able to run as close to real-time as possible. SAFECHAIN employs two *rule-aware optimization* techniques to reduce redundant checks and to run significantly faster than using an off-the-shelf model checker.

Our comparative analysis shows that SAFECHAIN can efficiently and accurately identify attack chains. Our prototype implementation of SAFECHAIN can efficiently verify up to 300 automation rules within one second, outperforming the baseline without any optimization, which can take more than 15 minutes. The experimental results also show that SAFECHAIN has no false positives under appropriate assumptions.

**Contributions.** This paper makes the following contributions:

- We analyze the attack chains found in a real-world dataset, investigate two attack classes (i.e., privilege escalation and privacy leakage), and formulate them as checkable properties on FSMs.
- We design and implement SAFECHAIN, a lightweight system to detect the two attack classes.
- We evaluate SAFECHAIN using a large-scale dataset and compare with prior work. We show that SAFECHAIN can verify 300 rules in less than 1s, which is up to 1,000 times faster than the baseline approach, with no false negatives.

2. A state is a value assignment of all device variables, rules are modeled as state transitions, and exploitable devices are modeled as arbitrary change of the variables of these devices at any time.

## 2 BACKGROUND

Before explaining how to identify vulnerable trigger-action rules using model checking, we introduce the related background knowledge and terminologies.

### 2.1 Trigger-Action Rules and IFTTT

Networking capabilities allow IoT devices to communicate and share information with each other. For example, an occupancy sensor can control the lighting or heating system in a smart home when it detects motion in the space, making daily life more convenient and reducing unnecessary power consumption. To support such custom automation, users can utilize trigger-action programming to specify triggering circumstances to execute actions. The general format of a trigger-action rule is as follows:

IF *trigger*, THEN *action*.

For example, the *trigger* of the previous instance is when the occupancy sensor detects something and the *action* is to turn on the light or activate the air conditioner. Thanks to its simplicity and straightforwardness, novice users can use such programming to customize their IoT device behavior [43]. These rules can also be machine generated [20], [35] or learned [38], [43] from user intent.

IFTTT [11] is one of the leading services and platforms to help users define custom automation on their IoT devices. With more than one million registered users, IFTTT has connected more than 400 devices and online services, and in 2015, more than 19 million rules have been created and 600 million rules have been executed monthly [18]. Other platforms providing similar services include Samsung SmartThings [17], Zapier [19] and Microsoft Flow [12].

### 2.2 Model Checking

Model checking is a method to formally verify finite-state systems. A model (i.e., an abstract representation) of a system is automatically and exhaustively checked to determine if it complies with specified properties. The desired property of a system is usually expressed in logic languages, such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL).

The characteristics of exhaustive checking from model checking is especially suitable for security validation, because every hidden threat can be found with no false alarm. Several off-the-shelf model-checking tools are provided to help verify systems, such as NuSMV. Once the users model their systems as finite state machines and express properties in supported logic languages, the model checker can help determine if there are any violations.

## 3 CASE STUDY

Before formally defining the problem, we show it is possible to launch an attack using several harmless automation rules and describe a home scenario as a working example throughout the paper.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



TABLE 1: Examples of chained recipes

| Chain | Recipe 1 | Recipe 2 | Type |
|---|---|---|---|
| C1 | Convert an e-mail to event in Google Calendar | Send recurring Square Cash payments with Google Calendar & Gmail | privilege |
| C2 | Disconnect from home Wi-Fi, start recording on Manything | When Manything detects motion, scare the intruder. | privilege |
| C3 | Turn off sprinklers when I arrive home | If irrigation stopped, then blink lights | privacy |
| C4 | When your nest thermostat is set to away then put your water heater in vacation mode | If water heater enters vacation mode, then turn off the lights | privacy |

## 3.1 Chained Recipes

Table 1 shows interesting examples of chained recipes we found that can lead to attacks. The recipes are all chosen from the publicly available information on the IFTTT website in April, 2018.[3] For simplicity, we only consider chains of length two.

The first two examples show privilege escalation attacks. C1 exhibits how untrusted inputs flow through recipes to a trusted action. The first recipe of C1 enables almost anyone to create calendar events by sending mails to the owner, and the second recipe of C1 creates recurring payments if an event added to Google Calendar matches a given format. Therefore, an attacker can receive payments from the victim by simply sending a crafted email. C2 shows another example in which the recipes are chained implicitly. The first recipe of C2 will turn on the security camera (i.e., Manything) when the user leaves home, while the second recipe of C2 will turn on lights, sounds or speakers in order to scare an intruder. This implies that even if the user is home and the camera is turned off, an attacker can jam the Wi-Fi to control the house's lighting, sound, or speaker system. We consider them as chained recipes since the second trigger fires only after the camera is turned on. Though the attack might not cause any real damage, it can still be annoying.

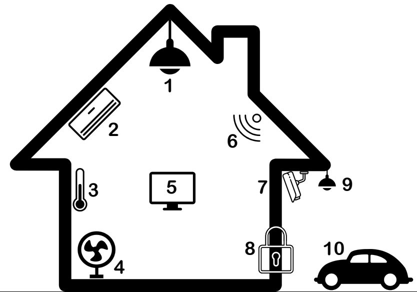
The last two examples illustrate privacy leakage attacks, and we assume that knowing whether one is home is sensitive, and that lightbulbs can be easily observed or compromised [40]. With C3, an attacker can learn that the owner returns home if the observable light blinks. Similarly, in C4, the light indicates whether the water heater is in vacation mode or not, and the water heater's mode is determined by a nest thermostat. Thus, the thermostat mode (away or home) is leaked by the light in C4.

## 3.2 A Working Example

Figure 1 illustrates a simplified smart home scenario consisting of ten devices and 12 rules. Smart home in reality can be even more complex (and thus harder to analyze), as the number of connected IoT devices per household are anticipated to rise to 50 by 2020 [9], and a real dataset provided by a smart home owner (see Appendix B for details) contains 85 IoT devices connected through nearly 70 automation rules. The table at the bottom of Figure 1 summarizes the devices and their possible statuses, and Table 2 lists the automation rules used in this example. Note that rules may be created by multiple users to accommodate individual needs. A small user study (see Appendix C for details) showed that a household of three will have a high chance to adopt all rules used in this example.

These rules are specified in the format of trigger-action programming and can work in an automation service like IFTTT.

3. Because there is no indication whether a rule is actively used by a user, we assume a user may sign up for any subset of rules.




| # | Device | Attribute | Possible statuses |
|---|---|---|---|
| 1 | Light bulb | `light2` | **ON**, **OFF** |
| 2 | Air conditioner | `ac` | **ON**, **OFF** |
| 3 | Thermometer | `temperature` | **0,1,...,100** |
| 4 | Fan | `fan` | **ON**, **OFF** |
| 5 | Smart TV | `tv` | **ON**, **OFF** |
| 6 | Occupancy sensor | `occupancy` | **TRUE** **FALSE** |
| 7 | Surveillance camera | `camera` | **ON**, **OFF** |
| 8 | Smart lock | `lock` | **LOCKED** **UNLOCKED** |
| 9 | Light bulb | `light1` | **ON**, **OFF** |
| 10 | Smart car | `location` | **0,1,...,127** |

Fig. 1: A Smart Home Scenario

TABLE 2: Example Rules

| Rule | Trigger | Action |
|---|---|---|
| R1 | User is near home<br>`location` = **0** | Turn on the outside lightbulb<br>Turn on the surveillance camera<br>`light1` ← **ON**, `camera` ← **ON** |
| R2 | The surveillance camera is turned on<br>`camera` = **ON** | Unlock the smart lock<br>`lock` ← **UNLOCKED** |
| R3 | User has been driven out<br>`location` ≠ **0** | Lock the smart lock<br>`lock` ← **LOCKED** |
| R4 | The smart lock is locked<br>`lock` = **LOCKED** | Turn off the outside lightbulb<br>`light1` ← **OFF** |
| R5 | The outside lightbulb is off<br>`light1` = **OFF** | Turn off the surveillance camera<br>`camera` ← **OFF** |
| R6 | Occupancy sensor detects someone<br>`occupancy` = **TRUE** | Switch on the smart TV<br>`tv` ← **ON** |
| R7 | Occupancy sensor detects nobody<br>`occupancy` = **FALSE** | Switch off the smart TV<br>`tv` ← **OFF** |
| R8 | The smart TV is on<br>`tv` = **ON** | Turn on the inside lightbulb<br>`light2` ← **ON** |
| R9 | The smart TV is off<br>`tv` = **OFF** | Turn off the inside lightbulb<br>`light2` ← **OFF** |
| R10 | Temperature is a little high<br>`temperature` ≥ **28** | Turn on the fan<br>`fan` ← **ON** |
| R11 | Temperature is high<br>`temperature` ≥ **32** | Turn on the air conditioner<br>`ac` ← **ON** |
| R12 | Temperature is low<br>`temperature` ≤ **25** | Turn off the fan<br>Turn off the air conditioner<br>`fan` ← **OFF**, `ac` ← **OFF** |

In this example, the homeowner's intention is to record people entering or leaving the house. Thus, the surveillance camera will be turned on before the door is unlocked (R1-R2) and off after the door is locked (R3-R5). The rules from R6 to R12 are designed for energy efficiency, so the appliances in the living room will be switched on and off with respect to different conditions of weather and human presence.

Unfortunately, not every IoT device is equally secure; some might have vulnerabilities that have not been patched. By compromising a vulnerable device, an attacker may be able to control or observe other devices due to inter-device dependencies. In the example in Table 2, suppose the GPS sensor in the smart car [7] and the lightbulb [13] are compromised, and their states are controlled by an attacker. The attacker can infer whether the owner is home (i.e., by knowing the status of the occupancy sensor), and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



stealthily break in the house (i.e., by unlocking the smart lock while the surveillance camera is off) by leveraging one or a chain of automation rules as we will explain in §4.1.

From the above examples, most of the recipes may look seemingly harmless if observed individually, but can become harmful when chained together. As the number of devices and recipes are likely to increase in the near future, it is harder for humans to debug unsafe chains, especially when recipes are created at different times or by different people.

## 4 Problem Definition

The goal of this work is to create an automated system that can efficiently detect trigger-action *attack chains* and suggest fixes to users. Here, we define an attack chain as a set of rules such that the action of one rule in the set satisfies the trigger condition of another rule in the set.

Consider a smart space[4] consisting of IoT *devices*, user-defined trigger-action *rules*, and a *service provider*. These devices can interact with each other explicitly through automation rules. Such automation is implemented using a trusted service provider that executes rules whose trigger conditions are satisfied by the current device state.

In the rest of this section, we define the threat model, system model, and desired properties in detail.

### 4.1 Threat Model

We consider a realistic attacker who has compromised a set of vulnerable devices at the beginning; for example, via malicious apps, known exploits, or proximity-based attacks [46]. The attacker can read and write attributes of the compromised devices at any time.

Because we are developing a defense system, we consider a strong adversary that knows all the rules created by users[5]. By successfully mitigating such a strong adversary, we can also mitigate weaker adversaries who know partial information, as discussed in §7.

The attacker's goal is to exploit IoT automation and perform unauthorized actions *(privilege escalation)* or unauthorized reads *(privacy leakage)*.

**Privilege escalation.** In a privilege escalation attack, the attacker attempts to make the IoT system transition into an insecure state (e.g., the door is unlocked when cameras are off), which can never be reached if the devices are operating as expected. To do this, the attacker actively manipulates the attributes of the compromised devices, thereby triggering changes of other devices via automation rules. Sometimes the attacker may also need to manipulate the device attributes in a specific sequence and at a specific time.

In the example in §3.2, the attacker can manipulate the state of the smart lock and surveillance camera, thus break in stealthily without being recorded, even though the attacker has no direct control of the two devices. To achieve this, the adversary forces the GPS sensor to incorrectly report (e.g., by generating a stronger, fake GPS signal or hacking the backend service [15]) that the car is home. The service provider is then misled to apply rules R1 and R2 to turn on the outside lightbulb and surveillance camera and unlock the smart lock, respectively. After that, the adversary forges the status of the lightbulb to trigger rule R5, which turns off the surveillance camera.

**Privacy leakage.** In a privacy leakage attack, the attacker attempts to deduce private information from publicly observable data and the attributes of compromised devices. In addition to passive observation, the attacker can also actively manipulate compromised devices and observe the resulting changes.

In the example in §3.2, the attacker can infer whether the owner is home by monitoring the state of the vulnerable lightbulb inside. This is because when the occupancy sensor detects a human presence, rules R6 and R8 will be triggered, thus turning on the smart TV and the inside lightbulb. On the other hand, if the occupancy sensor detects no one, rules R7 and R9 will be applied, and both the smart TV and inside lightbulb will both be off. Thus, the adversary can infer the state of the occupancy sensor through monitoring the state of the inside lightbulb.

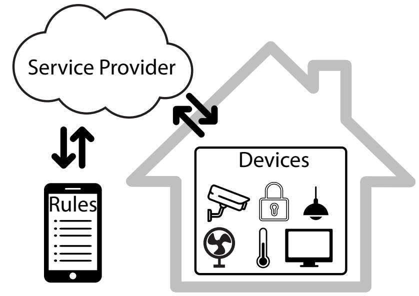



Fig. 2: System Model

### 4.2 System Model

Figure 2 shows our system model. More details are described as follows.

**Devices.** A device's state can be represented using a set of attributes, which can be accessed via APIs. For example, a thermometer can have a `temperature` attribute, which is set to the value perceived by its temperature sensor; a lightbulb or surveillance camera can have a `switch` attribute that represents whether its functionality is enabled or disabled.

Note that these attributes may be affected by the time-varying *environment* via the devices' sensors. They may also affect the environment via the devices' actuators.

**Trigger-action rules.** Users can enable automation between devices by adding customized trigger-action rules. Users need to specify a trigger and the corresponding action when creating a customized rule.

**A service provider.** We consider IoT automation implemented using a trusted service provider[6] (e.g., IFTTT, Zapier, or Samsung SmartThings). The service provider offers an interface for users to add or remove customized rules and can interact with devices

4. Such a smart space can be a home, building, office, factory, etc. For ease of demonstration, we will use smart homes (as illustrated in §3.2) in our examples throughout the paper.

5. Many IFTTT users publish their rules online, and most service providers (e.g., IFTTT, Zapier, or Samsung SmartThings) provide a public dataset of automation rules so that users can quickly create customized automation.

6. Securing the service provider is an orthogonal problem to our work, but we emphasize that the service provider has incentives and resources to employ better security measures than individual IoT devices. Many IoT vendors focus on providing novel functionalities rather than security, and IoT devices rarely adopt strong security measures due to limited resources. On the other hand, once the service provider is compromised, the attacker can directly control all devices to launch powerful attacks, which will ruin the service provider's reputation.

through their APIs. We assume that the service provider polls devices periodically; that is, queries every device's status to check for satisfying rules and to apply corresponding actions. We also assume that the service provider can resolve conflicting rules and avoid ambiguities, e.g., by enforcing an order of precedence on rules [34], [39].

### 4.3 Desired Properties

**Low false rates.** The system should be able to accurately identify privilege escalation and privacy leakage attacks. A false positive occurs when the system falsely reports an attack, which may annoy users or affect normal functionality of IoT devices. A false negative occurs when the system fails to identify an attack, providing users with a false sense of security.

**Timely detection.** The system should be able to scale to hundreds or even thousands of devices and rules, and detect potential attacks in a timely manner. Timely detection ensures that users have sufficient time to fix problematic rules before they are exploited. Designing a scalable algorithm is challenging because the number of possible state combinations grows exponentially with devices and rules.

**Low interference with intended functionalities.** One simple way to prevent attackers from exploiting automation rules is to ask users for permission before executing every rule. However, this simple fix contradicts the purpose of IoT automation, which is to make users' lives easier. Hence, the mitigation method should avoid interfering with users' intended functionalities and should not place a burden on users.

## 5 SafeChain

The core idea of SafeChain is to model the IoT ecosystem as a Finite State Machine (FSM), such that finding an attack can be reduced to a property-checking problem on the FSM, which can be solved using existing model checkers. To improve its practicability, SafeChain deploys novel techniques to overcome the research challenges described in §1.

After highlighting the core insights and the system overview (§5.1), we explain how SafeChain models the problem of identifying automation rule exploitation as a reachability problem on a finite state machine (§5.2), such that we can adopt model-checking techniques to ensure accurate detection of privilege escalation and privacy leakage (§5.3). As model checking tends to be a slow process, we propose several rule-aware optimization techniques to achieve timely detection (§5.4). Finally, we explore how to mitigate the identified attacks with low interference (§5.5).

### 5.1 High-Level System Overview

As shown in Figure 3, SafeChain takes in rules, devices, vulnerability databases, and security policies, and reports identified attacks to users. Users can interact with SafeChain using an application interface to check detailed attack traces and apply fixes.

We envision that SafeChain can work as an extension to an existing IoT automation service provider or as a standalone system, and the inputs can be supplied by the service provider, IoT vendors, crowdsourcing sites, users, etc. Experts can help provide default security policies, and users can revise them according to their needs.

Because the adversary's goal is to control an uncompromised device or leak data from an uncompromised device via automation rules, it is reasonable to consider cases where some devices are vulnerable and some are not. (The attacker succeeds immediately if all devices are compromised.) Several approaches exist to determine the vulnerable devices. For example, vulnerability databases provide lists of vulnerable devices[7]. Alternatively, users can manually select devices that require high level of protection, in which case our system assumes that all the other devices are vulnerable and evaluates whether the manually-selected devices are attackable.

SafeChain consists of four major components:

1) **Modeling.** To model the interaction between devices and rules, we build a FSM in which the device statuses and automation rules correspond to the states and transitions, respectively. In addition, to model a volatile environment (Challenge 1), we use a short-lived window to predict the changes of each sensor variable and renew the prediction after the previous one expires.
2) **Verification.** To verify the system, it is crucial to define the attacker model in the FSM. Given the security policies, the verification component translates these policies into the FSM properties, which can be checked using a model checker. Once the short-lived windows are due, our system re-verifies the model again with the new windows.
3) **Optimization.** To scale to a large number of devices and rules (Challenge 2), we propose two rule-aware optimization techniques to shrink the size of the FSM by pruning redundant states and grouping equivalent states. Since the optimization is done before actually transforming into FSM, with high-level semantics preserved, our approach incurs less overhead compared to general optimization techniques implemented in common model checkers.
4) **Mitigation.** The mitigation component greedily selects a small set of rules whose removal can disable the identified attacks. To avoid violating intended functionalities, this component also shows the identified attack traces and suggests fixes to the users.

As we will show in §6, SafeChain can verify hundreds of rules in just a few seconds, and thus is capable of frequent rechecks based on the latest rule set and sensor-attribute values. Note that frequent rechecks are needed to accommodate the short-lived window for the modeling environment.

### 5.2 Modeling

We will first describe how we encode each concept separately, and then explain in detail how SafeChain models the whole smart space to simulate the possible interactions. The same modeling is used for detecting both privilege escalation and privacy leak.

**Devices.** Each device is symbolized by using a set of attributes to represent its equipped sensors, actuators or internal states. We use $A$ to denote this attribute set of all devices, which are classified into two disjoint groups: read-only attributes $A_R$ and read-write attributes $A_{RW}$. The former corresponds to sensors, which provide APIs only for obtaining values, and the latter

7. Several public datasets have been established to consider the consumer device vulnerabilities. For instance, National Vulnerability Database [14] and AndroidVulnerabilities [2] have accumulated a variety of vulnerabilities with respect to different devices and smart phones.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



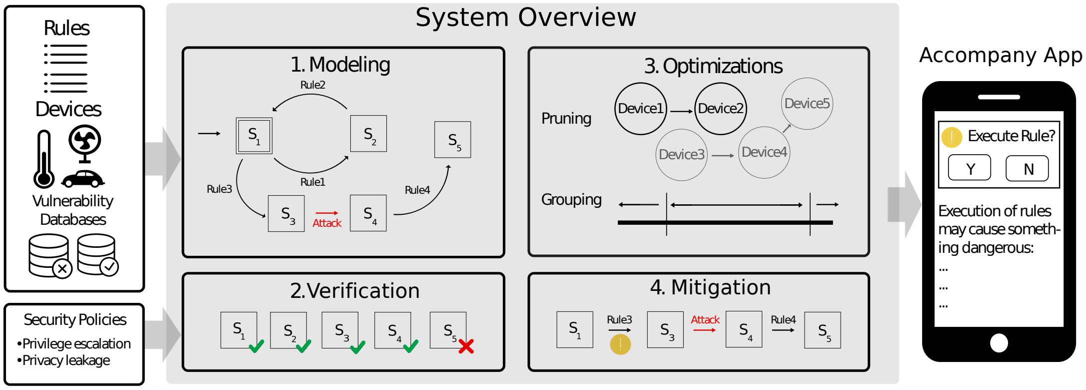



Fig. 3: System overview. "S" stands for states.

corresponds to actuators, which provide APIs for obtaining and setting values. Possible values of each attribute $a_i \in A$ are specified in its API specifications and we use $possible(a_i)$ to denote this set.

**Automation rules.** A rule, represented by $r_i \equiv (bool_i, assign_i)$, can be specified by users or generated by machines. $bool_i$ is a Boolean-valued function defined over the device attribute space, and $r_i$ is triggered when $bool_i$ is evaluated to be **TRUE**. $assign_i$ defines the action of the rule. This function maps device attributes to values.

For simplicity, we assume that all the satisfied rules (i.e., $bool_i(s)$ = **TRUE**) are executed concurrently. We also assume that if there is a conflict between rules, some resolution techniques are applied, such as user preferences, to ensure that the service provider knows which rules to execute. Two rules $r_i$ and $r_j$ are said to be conflicting if they assign different values to the same variable (i.e., $assign_i(a_k) \neq assign_j(a_k)$).

**Environment.** The values of sensor attributes will change as the environment changes and corresponding sensors perceive the differences. At one extreme, we could try to model the environment accurately using knowledge and formula in physics. However, since the smart space can be a home, office or factory, they may differ greatly in their environments. It is hard to use one or few formulas to cover all conditions. At the other extreme, we could set every sensor attribute to be constraint-free and check every combination of their values at any time, but this may cause excessive false alarms that annoy users because some combinations may never happen in the real world.

To strike a middle ground between the two extremes, we propose a practical approach to handle environmental changes in SAFECHAIN by focusing on possible changes in the near future. Specifically, for each sensor attribute $a_i \in A_R$ of a secure device, we try to predict a window $window(a_i) \subseteq possible(a_i)$ in which this attribute will reside during a period of time, and after the period has elapsed, we repeat the prediction process and verify the rules again. For example, the temperature is expected to be between 23 and 33 Celsius degrees tomorrow, based on the weather forecast. To ensure that no attacks will occur tomorrow, there is no need to check values outside this window (unless the thermometer is assumed to be hacked). A similar assumption can be made for the GPS sensor in a smart car, because the car movement obeys the laws of physics and cannot move faster than a certain speed. For sensors without known constraints on their attribute values (e.g., an `occupancy` sensor), we enumerate all possible conditions in the future, i.e. $window(a_i) = possible(a_i)$.

In addition, to alleviate the impact of inaccurate prediction, SAFECHAIN will monitor the sensing data and immediately recheck and re-predict if the prediction is violated. In our implementation, the prediction window is derived from a fixed width for each sensor attribute and can be adjusted with respect to users' tolerance to false alarms. It can also be improved by using techniques such as machine learning.

**Adversary.** Given the vulnerability databases, SAFECHAIN uses $A_{VUL} \subseteq A$ to denote the attribute set of vulnerable devices. Any attribute in $A_{VUL}$ can be monitored or modified by an attacker at any time.

An attacker can be either active or passive. A passive attacker gathers information about compromised devices over time and tries to infer information about secure devices, while an active attacker reports bogus information to trigger automation rules. We use a special attribute `attack`, which can be either **ACTIVE** or **PASSIVE** at a time, to represent the chosen strategy.

The whole smart space is then modeled as a finite state machine, which is a tuple $FSM \equiv (S, \longrightarrow, I)$ where $S$ is a set of states, $\longrightarrow \subseteq S \times S$ is a transition relation, and $I \subseteq S$ is a set of initial states.

A state $s \in S$ is an $N$-tuple $(a_1, a_2, \ldots a_N)$, where $a_i \in A$ are the attributes of all installed devices. We use the notation $s(a_i)$ to represent the value of $a_i$ in $s$. Set $S$ consists of all the possible states in the smart space while set $I$ contains only one state representing the current status. The next possible state $s'$ from state $s$ can be affected by automation rules, environment, and the adversary simultaneously. Formally for state $s$, the transition relation $(s, s') \in \longrightarrow$ exists if for any $a_i \in A$,

$$s'(a_i) \left\{ \begin{array}{ll} \in possible(a_i) & \text{if } a_i \in A_{VUL} \text{ and } s'(\texttt{attack}) = \textbf{ACTIVE} \\ \in window(a_i) & \text{if } a_i \in A_R \\ = assign_k(a_i) & \text{if } bool_k(s) = \textbf{TRUE} \text{ for some rule } r_k \\ = s(a_i) & \text{otherwise.} \end{array} \right\}.$$

The first condition corresponds to the case when the attacker actively controls the devices so that the vulnerable attributes can be set to any possible value. The second condition limits the environment attributes to remain in our predicted values, and the third defines the effects of automation rules. In addition to what is discussed, the status of devices will remain unchanged.

### 5.3 Verification

In this subsection, we explain the formats of security policies for privilege escalation and privacy leakage, and how such policies are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



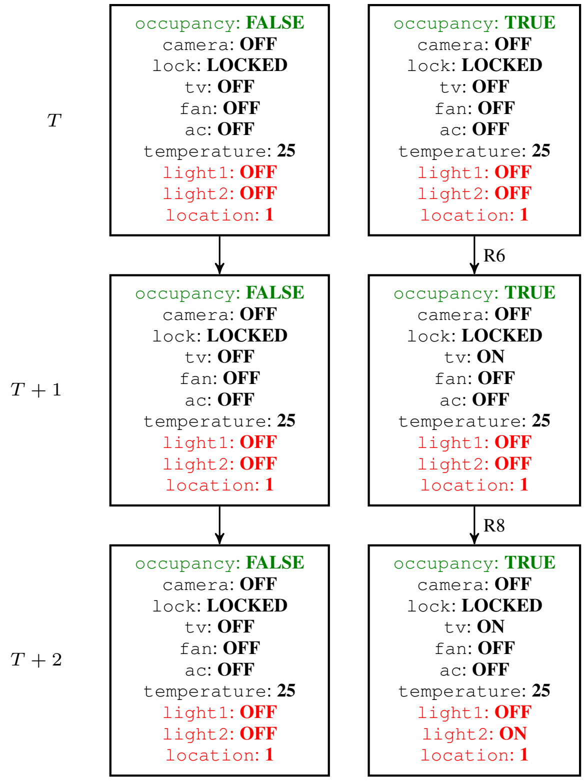
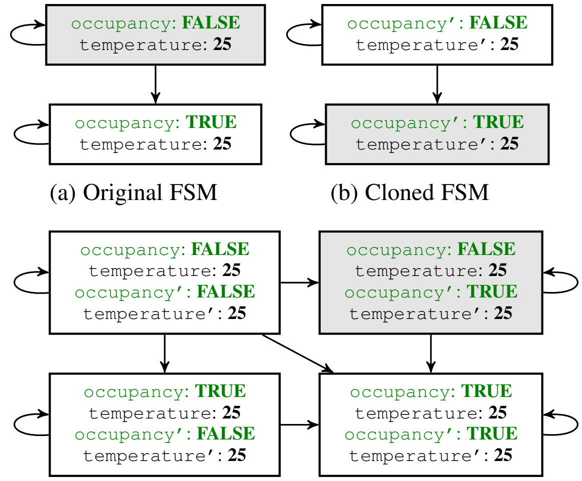



Fig. 4: Example of privacy leakage. Different PRIVATE values (green) lead to different PUBLIC values (red) after some transitions.




(c) Product FSM

Fig. 5: Example of a product machine. The initial states are shaded. On the product FSM, two parallel traces of the original FSM can be viewed as a single trace, which makes our privacy policy expressible in LTL/CTL.

translated into FSM's properties that can be verified using model checking tools. We envision that (1) our system has a set of pre-installed general security policies written by experts, and (2) users can modify or create their own policies, and share their policies to a public dataset. We note that even if sophisticated users can write their own security policies, it may still be challenging for them to manually check all possible interactions between devices.

**Privilege escalation.** A security policy defines the expected behaviors of devices, and can be represented by conditions that either must be or must not be satisfied. They are stated in the form of "$device_1$ is (not) $state_1$ and/or $device_2$ is (not) $state_2$ and/or ...". To improve usability, users only need to select the values of $device_i$, $state_i$ as well as the logical connective (and, or, not), and SafeChain will then convert it into LTL or CTL. For example, $AG(\texttt{lock} = \textbf{LOCKED} \vee \texttt{camera} = \textbf{ON}) = AG\neg(\texttt{lock} = \textbf{UNLOCKED} \wedge \texttt{camera} = \textbf{OFF})$ in CTL expresses that the policy "smart lock is unlocked but surveillance camera is off" should not happen.

**Privacy leakage.** The privacy policy defines the level of confidentiality for each device, which can be expressed by assigning a label (PRIVATE, PUBLIC, or OTHER) to each attribute.

A device attribute labeled as PRIVATE indicates that the attribute contains confidential information and should be protected from an attacker's observation, while the PUBLIC label indicates that an attacker may end up observing the attribute through hacking or some local observations (e.g., lights can be observed from outside at night). We propose the OTHER label for those attributes that are neither confidential (PRIVATE) nor vulnerable (PUBLIC) from the user's perspectives (i.e., user does not care about the information leakage). Although the OTHER class complicates the analysis process and cannot be handled by prior methods [25], [27], [42], it allows us to accurately model real-world settings.

In the motivating example, the attribute `occupancy`, which indicates whether someone is home, is considered private. However, as shown in Figure 4, attackers can infer the actual value of this PRIVATE attribute (colored in green) at time $T$ by observing the PUBLIC attributes (colored in red) at time $T+2$, because different values of the PRIVATE attribute trigger different automation rules and eventually lead to different values of the PUBLIC attributes. That is, if the value of `light2` is **ON**, the value of `occupancy` should be **TRUE**. Otherwise, it should be **FALSE**.

Inspired by this observation, we want to ensure that any two states that are only different in PRIVATE attributes in the same environmental changes should stay indistinguishable from the adversary's perspective at any future time. That is, the values of the PUBLIC attributes should stay the same between these two traces at any moment. If not, the adversary can tell the two traces apart and infer the values of PRIVATE attributes. The environmental changes should be equivalent in both traces because attackers can only observe the change in real life.

Formally, SafeChain defines security against privacy leakage as follows:

$$\begin{aligned} &\forall s_0, s_0' \in S, t > 0 \\ &\begin{cases} s_0 =_{PUBLIC,OTHER} s_0' \\ s_t(a_i) = s_t'(a_i) & \forall a_i \in A_R \end{cases} \\ &\implies s_t =_{PUBLIC} s_t', \end{aligned}$$

where $=_{PUBLIC}$ and $=_{PUBLIC,OTHER}$ stand for the equivalence over PUBLIC only and PUBLIC and OTHER attributes, respectively.

Hence, to check whether a PRIVATE attribute can be leaked, we can modify the PRIVATE attribute and see if the trace is altered from the attacker's perspective. Nonetheless, these kinds of properties cannot be expressed in ordinary LTL or CTL because it requires pairwise comparison between two traces of the finite state machine.

To overcome this challenge, we construct a product machine as follows. We first build an almost equivalent copy of the original FSM, and their differences reside in the PRIVATE values of the initial states. For example, suppose the original FSM is Figure 5a,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



then the copy one named CLONE would be Figure 5b. (Shaded states are the initial states.) Formally, $CLONE \equiv (S, \longrightarrow, I')$, and $I' = \{s' | s' =_{L,O} s, \forall s' \in S, s \in I\}$. Then, the states of product machine $PM$ are Cartesian products of the states of the two state machines, as in Figure 5c. The initial state is the state that consists of the initial states of original and cloned FSM. The transitions can be seen as the union of the transitions from both machines. That is, $PM \equiv (S \times S, \longrightarrow', I \times I')$, where $\longrightarrow' = \{((s_i, s_j), (s'_i, s'_j)) | (s_i, s_j) \in \longrightarrow \wedge (s'_i, s'_j) \in \longrightarrow\}$.

We also add constraints to ensure that the original and cloned machines undergo the same environmental changes. To this end, the problem is reduced to whether the product machine can arrive at a state where its two internal states are different in the PUBLIC attributes, which is a reachability problem and can be expressed in ordinary LTL or CTL. Although building a product machine increases the state space[8], our evaluation shows that the verification can still be efficiently performed with our optimization techniques.

Our product FSM construction is similar to the self-composition approach proposed by Gilles et al. [27]. However, it is unclear whether their program-execution-based non-interference definition can model the event-driven scenarios in trigger-action programming. In addition, SAFECHAIN avoids the additional cost of transforming a program to a state machine because the security properties of trigger-action programming can be directly defined over a transition system instead of a program.

### 5.4 Optimization

To enable timely verification, SAFECHAIN's optimization component aims to simplify the input (a model of a smart space) by removing redundant attribute values, rules, and devices. As the input to the model checker is reduced, the verification process can be accelerated.

We explain in detail how SAFECHAIN reduces the size of the problem by *grouping* attribute values that always cause the same effects, and by *pruning* devices and rules irrelevant to the security policy.

Generally speaking, our practice shares the same spirit with common techniques to address state explosion (e.g. grouping equivalences). However, thanks to the simple structure of trigger-action programs, we can effectively address the state explosion problem in model checking using static code analysis. We first obtain high-level program semantics using static analysis, and then efficiently construct an abstraction of the state-transition system without spurious counterexamples; thus, no further refinement is needed [23]. In addition, as our application scenarios consider non-technical users who specify their own rules and security policies, we further consider optimizations for simple security polices (in contrast to complex ones with temporal qualifiers).

**Grouping.** In trigger-action programming, two attribute values can be considered *equivalent* if they always trigger the same actions, regardless of the other attributes. Hence, we can reduce the number of states needed to be considered by grouping equivalent values into subgroups (or meta-values) and then rewrite the automation rules using the new meta-values. Algorithm 1 shows the pseudo-code. First, we collect all constraints from the rules (Lines 2-9) and the policy (Lines 10-13). Then, the acquired constraints are sorted for each attribute, which is used to classify possible

8. If the original state machine is $O(2^N)$, then the product machine will become $O(2^N) * O(2^N) = O(4^N)$, where $N$ is the number of attributes.

TABLE 3: Grouping of variables. Given a set of rules, values of variables are grouped together if they trigger same actions, reducing the number of states to be considered in FSM.

| Attributes | Possible values | Constraints | Grouped values |
|---|---|---|---|
| light1 | **ON, OFF** | light1 = **OFF**<br>light1 ← **OFF** | **ON, OFF** |
| camera | **ON, OFF** | camera = **ON** | **ON, OFF** |
| location | **0, ..., 10*** | location = **0**<br>location ≠ **0** | **0,OTHERS** |
| lock | **LOCKED**<br>**UNLOCKED** | lock = **LOCKED** | **LOCKED**<br>**UNLOCKED** |
| occupancy | **TRUE, FALSE** | occupancy = **TRUE**<br>occupancy = **FALSE** | **TRUE, FALSE** |
| tv | **ON, OFF** | tv = **ON**<br>tv = **OFF** | **ON, OFF** |
| light2 | **ON, OFF** | light2 ← **ON**<br>light2 ← **OFF** | **ON, OFF** |
| fan | **ON, OFF** | | **ALL** |
| ac | **ON, OFF** | | **ALL** |
| temperature | **23, ..., 33*** | temperature ≥ **28**<br>temperature ≥ **32**<br>temperature ≤ **25** | **26..27,28..31**<br>**32..33**<br>**23..25** |

* Short-lived windows for location and temperature are [**0, 10**] and [**23, 33**] respectively.

TABLE 4: Rule rewriting. Rules in Table 2 are rewritten w.r.t. the grouped values in Table 3.

| Rule | Trigger | Action |
|---|---|---|
| R10 | Temperature is a little high<br>temperature = **28..31**<br>∨temperature = **32..33** | Turn on the fan<br>fan ← **ALL** |
| R11 | Temperature is high<br>temperature = **32..33** | Turn on the air conditioner<br>ac ← **ALL** |
| R12 | Temperature is low<br>temperature = **23..25** | Turn off the fan<br>Turn off the air conditioner<br>fan ← **ALL**, ac ← **ALL** |

values into meta-values (Lines 14-18). Here, we build a map to convert between meta-values and original values. Finally, we translate rules to equivalencies by looking up the map (Lines 19-24). Overall, sorting the acquired constraints for each attribute takes most of the time, and the number of possible constraints is proportional to the number of rules. For a single attribute, the maximum number of constraints is $O(kM)$, where $k$ is the maximum number of constraints in a rule and $M$ is the number of rules. Hence, the time complexity is $O(kMN \log kM)$, where $N$ is the number of attributes.

Table 3 and Table 4 show the meta-values and rewritten rules after the grouping method is applied to the example in §3.2, respectively. The temperature values $\leq 25$ are considered equivalent because they all trigger rule R12. The possible values of the attribute temperature are divided into four subgroups, thereby the number of states is reduced to four. In rule R10, the trigger $\texttt{temperature} \geq \mathbf{28}$ is converted to $\texttt{temperature} = \mathbf{28..31} \vee \texttt{temperature} = \mathbf{32..33}$ and action $\texttt{fan} \leftarrow \mathbf{ON}$ is the same as $\texttt{fan} \leftarrow \mathbf{ALL}$ because its value does not affect the execution of any rules.

Note that constraints in policies should also be taken into consideration as they will also affect the grouping results.

**Pruning.** While grouping reduces the number of possible values of an attribute, pruning aims to reduce the number of attributes needed to be considered because not every device can influence the state of another device. For instance, to check this CTL specification $AG\neg(lock = \mathbf{UNLOCKED} \wedge camera = \mathbf{OFF})$ in the example in §3.2, we only need to consider devices in the front door because all the devices inside the house have no impact (via the automation rule) on the smart lock and the surveillance

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



**Algorithm 1:** Grouping

```
Data:
A is a list of attributes
R is a list of rules
P is the policy to be verified
Result: A^Grouped, R^Grouped
begin
    constraints = Dictionary(List);
    for r_i ∈ R do
        trigger ← GetTrigger(r_i);
        for constraint ∈ trigger do
            a_i ← GetTarget(constraint);
            constraints[a_i].append(constraint);
        end for
    end for
    for constraint ∈ P do
        a_i ← GetTarget(constraint);
        constraints[a_i].append(constraint);
    end for
    for a_i ∈ A do
        values ← Sorted(constraints[a_i]);
        a_i^Grouped ← GroupAttribute(a_i, values);
        A^Grouped.append(a_i^Grouped);
    end for
    for r_i ∈ R do
        bool_i ← GetEqualTrigger(A^Grouped, r_i);
        assign_i ← GetEqualAction(A^Grouped, r_i);
        r_i^Grouped ← MakeRule(bool_i, assign_i);
        R^Grouped.append(r_i^Grouped);
    end for
end
```

camera.

Algorithm 2 shows the pseudo-code for pruning. To prune irrelevant devices, we represent the relationship between devices by building a dependency graph (Lines 2-14), in which each vertex represents an attribute, and a directed edge from one vertex to another represents one attribute being able to affect the other. After building a dependency graph, we back-trace from interesting attributes (that are involved in the security policy) to find relevant devices (Lines 15-26). Building the dependency graph takes most of the time and requires adding edges between trigger attributes and action attributes. The worst-case scenario is that both the trigger and action contain all attributes for each rule. Thus, the time complexity is $O(MN^2)$, where $M$ and $N$ are the number of rules and attributes, respectively.

Figure 6 illustrates the dependency graph of the example in §3.2. For each rule, we check the dependency between the involved devices. For example, in rule R1, the attributes `light1` and `camera` are affected by the attribute `location`. Thus, on the dependency graph, we add two directed edges from `location` to `light1` and `camera` respectively, and label these two edges with rule R1 to track their relationship. During backtracking, we start from the interesting attributes (e.g., `lock` and `camera` in the policy $AG\neg(\texttt{lock} = \textbf{UNLOCKED} \wedge \texttt{camera} = \textbf{OFF}))$ and find all connected vertices (i.e., relevant attributes) and connected edges (i.e., relevant rules).

In privacy leakage, the vulnerable devices observable by attackers are interesting attributes, such as `light1`, `light2` and `location` in the example in §3.2. We further accelerate the verification of privacy leakage by filtering out PUBLIC attributes that are unreachable from any PRIVATE attribute on the dependency graph, because there is no leak if the PRIVATE attributes have no impact on the PUBLIC attributes. If any PUBLIC attribute can be reached, we back-trace from those reachable PUBLIC vertices. Hence, we can perform the verification with only three attributes after back-tracing from `light2` in the example in §3.2.

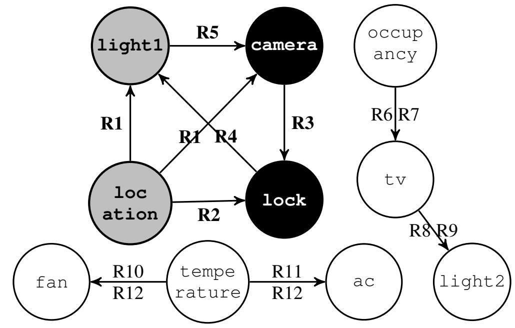



Fig. 6: Dependency graph for privilege escalation. The vertices represent attributes and the edges represent rules. Only attributes which involved in the security policy (in black) and their dependencies (in grey) need to be considered in the verification process.

**Algorithm 2:** Pruning

```
   Data:
   A is a list of attributes
   R is a list of rules
   P is the policy to be verified
   Result: A^Related, R^Related
1  begin
2      graph ← Graph();
3      for a_i ∈ A do
4          graph.addNode(a_i);
5      end for
6      for r_i ∈ R do
7          bool_attributes ← GetTriggerAttributes(r_i);
8          assign_attributes ← GetActionAttributes(r_i);
9          for a_j ∈ bool_attributes do
10             for a_k ∈ assign_attributes do
11                 graph.addEdge(a_j, a_k, r_i);
12             end for
13         end for
14     end for
15     unexplored_nodes ← GetAssociatedAttributes(Policy);
16     while unexplored_nodes ≠ ∅ do
17         a ← unexplored_nodes.pop();
18         A^Related.add(a);
19         for neighbor ∈ graph.neighbors(a) do
20             if neighbor ∉ A^Related then
21                 unexplored_nodes.add(neighbor);
22             end if
23             r ← graph.getEdgeMark(a, neighbor);
24             R^Related.add(r);
25         end for
26     end while
27 end
```

## 5.5 Mitigation

Ideally, if one could fix every vulnerable device immediately, the attacker would be unable to increase the attack surface by exploiting automation rules. However, patching in a timely manner is challenging, and devices may also have zero-day vulnerabilities. Hence, in this work, we discuss mitigations that can be achieved by updating automation rules, which include removing or modifying exploitable ones with users' consent. The question then becomes which to remove or how to modify them.

A straightforward approach is to put every rule involved in the identified attacks into a watchlist, and request confirmation from the user whenever a watchlisted rule is about to be executed. Although this can indeed prevent attacks, it undermines the convenience of this system.

To reduce the level of inconvenience, we would like to add as few rules to the watchlist as possible. For each attack chain, it is sufficient to stop the attack by blocking at least one rule in the chain. Also, one rule can appear in multiple attack chains. Therefore, one possible approach is to first determine all the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



TABLE 5: Comparison between SAFECHAIN and related work. O: supported; X: not supported; △: inaccurate; ◇: can be extended.

| Properties | SAFECHAIN | Milijana et al. [42] | SIFT [31] | Salus [30] |
|---|---|---|---|---|
| Consider attacks | O | O | X* | X* |
| Detect privacy leakage | O | △ | X | X |
| Detect privilege escalation | O | X | ◇ | ◇ |
| Detect integrity attack | ◇ | △ | X | X |
| Consider device states | O | X | O | O |
| Analysis technique | Model checking | Information flow | Symbolic execution | Model checking |
| Verification Speed | Fast | Very fast | Slow | Slow |

* focuses on the reliability problem (i.e. policy validation or conflict detection) in trigger-action programming.

TABLE 6: Rules for case study. R3 will raise a false alarm in static analysis because the volume is always greater than or equal to 50 at runtime and will never trigger R3.

| Rule | Trigger | Action |
|---|---|---|
| R1 | User is at home (*private*) `location` = **HOME** | Set the volume (*restricted*) `volume` ← **50** |
| R2 | User is outside (*private*) `location` ≠ **HOME** | Set the volume (*restricted*) `volume` ← **100** |
| R3 | Any phone call missed with low volume (*restricted*) `missed_call` = **TRUE** ∧ `volume` ≤ **0** | Set band to vibrate (*public*) `band_vibration` ← **TRUE** |
| R4 | Any phone call missed with low volume (*restricted*) `missed_call` = **TRUE** ∧ `volume` ≤ **50** | Blink the band LED (*public*) `band_led` ← **BLINK** |

attack chains, then apply some optimization techniques or greedy strategies to minimize the blocked rules.

Nonetheless, the question remains of how to effectively figure out all the attack traces, because model-checking tools like NuSMV report only one counterexample at a time. It is also unclear what the selection algorithm would be like. We will leave these problems for future research. On the other hand, Salus [30] proposed a solution that first parameterizes trigger conditions, and then finds a feasible configuration using model checking. However, such an approach also suffers from usability problems, because Salus needs user permission to accept or reject each set of feasible parameters one by one, and there may be multiple sets of feasible parameters.

## 6 EVALUATION AND IMPLEMENTATION

We evaluate SAFECHAIN's security and performance using analytical comparisons (§6.1) and experiments (§6.3-6.4) on a prototype implementation described in §6.2.

### 6.1 Security comparison

We compare SAFECHAIN with closely related work [30], [31], [42] and the results are summarized in Table 5. The integrity attack is the opposite of the privacy leakage attack; devices with attributes labeled PRIVATE should not be affected by vulnerable devices.

To identify potential integrity and privacy violations in automation rules, Milijana et al. [42] define information flow lattices and check if information can flow from a more restricted trigger to a less restricted action. However, as their work focuses on static analysis on automation rules and does not consider the actual attribute values, their method cannot detect whether the system will enter an unauthorized state in the future (privilege escalation). Also, static analysis may produce false positives when detecting privacy leakage, since the actual values of devices are omitted.

As shown in Table 6, R3 will raise a false alarm because the information is propagated from restricted to public, when in reality, it will never be triggered since the volume will always be higher than 0. Another example is R4. Whether it will leak information depends on the parameters. If the second condition is set to `volume` ≤ **100**, then the attacker cannot derive information about the volume value by monitoring the status of the LED. Hence, the location of this user will not be disclosed. We label both actions as public, because information can be leaked not only from publicly-observable devices but also from vulnerable devices, whose information can be directly accessed by the attacker. For example, some smart bands have been reported with vulnerabilities [24]. Once attackers break into a device, its information is considered public.

SIFT [31] and Salus [30] are designed to help users debug trigger-action rules by verifying whether the devices' interaction (through automation rules) meets users' expectations. Since they focus on reliability rather than security, their techniques cannot be directly applied to detect attacks exploiting automation, but may be extended to check privilege escalation to some extent. Using symbolic execution for automated analysis, SIFT [31] starts by transforming automation rules into IF statements in C#, and wrap all the rules in a while loop. Each policy specified by the users is then encoded as an assertion and will be checked by Pex, an automated whitebox testing tool for .NET. However, they only unroll the while loop for a fixed number of steps ($k$), and thus may have false negatives for violations occurring after $k$ steps. The subsequent work, Salus [30], adopts model-checking techniques as we do. However, it is unclear whether Salus can handle the growing complexity in IoT, since their experiments show that the time needed for verification increases exponentially after irrelevant devices are installed. Thus, with respect to performance, Salus can be seen as a baseline approach without optimizations.

On the contrary, SAFECHAIN is designed to detect privilege escalation and privacy leakage. It can be easily extended to detect integrity attacks by observing whether the values of the PRIVATE attributes change when the values of the PUBLIC attributes are altered. In addition, SAFECHAIN takes advantage of formal model checking, which provides stronger guarantees of reducing false negatives, and supports rule-aware optimizations to accelerate verification.

### 6.2 Implementation

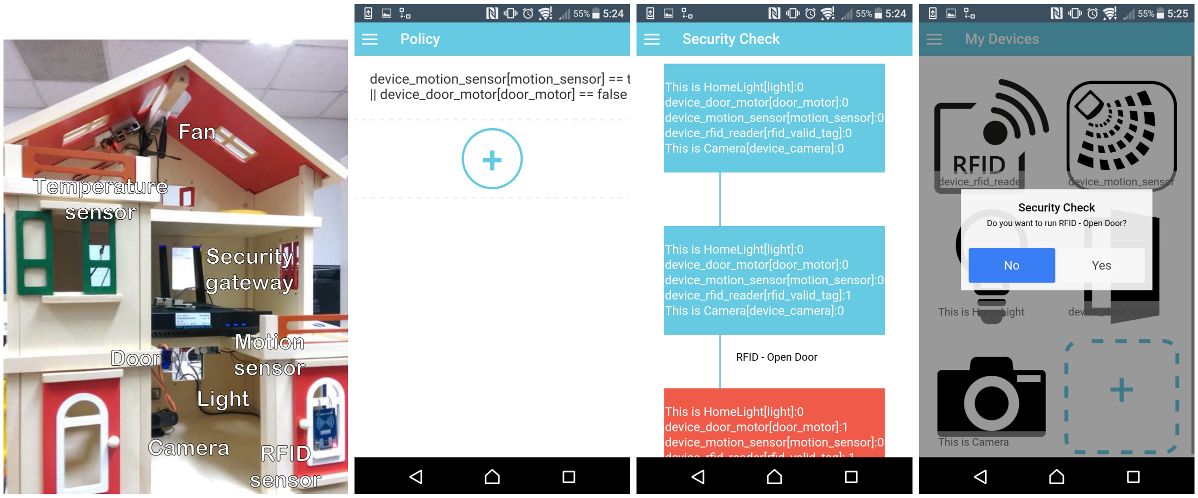
To demonstrate the practicality of our lightweight system, we built a smart home testbed and an Android application as shown in Figure 7. The major components, an IoT gateway and SAFECHAIN, reside in a Raspberry Pi 2B board.

We implement an IoT gateway to provide similar functionalities as existing service providers. Our system is built on top of the Kura framework [4], which is a Java/OSGi-based platform for building IoT gateways. We add several customized control messages to enable communication between devices and protocols like MQTT and CoAP. To simulate a smart space, we implement several IoT devices, such as fan, camera, temperature sensor and smart door using Arduino Yun, Raspberry Pi and Banana Pi boards. After devices are connected to the IoT gateway, they can share messages with each other.

Users can utilize our Android application to monitor, control and manage these devices. Similar to existing home automation services, trigger-action rules can also be added using Boolean function and assignments to enable automation. Our system will

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security






(a) A toy house (b) Security policies (c) Check result (d) Security alarms

Fig. 7: Proof-of-concept implementation

query devices' status regularly, apply satisfied automation rules and update their status in the application. To check which automation rules should be applied, we use the Javaluator library [10], which is a powerful infix expression evaluator for Java.

As discussed in §6.3, the security policies shown in Figure 7b can be set up by experts, modified by users, or selected from a public database, and SAFECHAIN will convert them into LTL or CTL for verification. The results are shown in our application and alarms will be popped out to request user confirmations when vulnerable automation rules are going to be executed, as shown in Figures 7c and 7d.

**User interaction model.** We describe how users interact with our system in a step-by-step manner. We assume a standalone app that pulls information about devices and automation rules from an existing automation platform (e.g., IFTTT). A similar procedure can be described when our system is integrated with an existing automation platform.

1) A user installs our app, which comes with default security policies maintained by experts. The user can also modify or write customized policies, as shown in Fig. 7b.
2) Our app runs in the background and periodically retrieves the most recent set of automation rules from the linked automation platform, and performs security checks according the specified security policies.
3) When the app detects a violated security policy, a pop-up warning is displayed, recommending the user to review the rule set. When the user clicks the warning or opens our app, the user can see detailed information about the attack trace of the violation, as shown in Fig. 7c.
4) If the vulnerable rule is triggered before the user fixes it, an alert will be displayed to request the user's explicit consent to execute the corresponding action, as shown in Fig. 7d. This feature may require permissions to modify the data stored in the automation platform, such as temporarily disabling the action of the vulnerable rule.

## 6.3 Performance Evaluation

Based on the implementation, we conducted a large-scale performance evaluation using a real dataset as described below. In each run of the experiment, we randomly sampled $N$ out of the 4,161 rules we encoded and generate security polices with respect to these sampled rules to simulate different use scenarios. SAFECHAIN can work with most model checkers; to compare with previous work [30], we use the open-sourced model checker, NuSMV, as our backend tool.

**Dataset.** We selected 42 IoT-related channels from a real dataset containing 313 channels and 295,155 IFTTT rules [44]. We manually modeled these selected channels and obtained 190 attributes and 4,161 rules. The selected channels are listed in Appendix A and the encoded data are available online [1]. The total state space is roughly $2^{650}$.

**Privilege escalation.** Recall that security concerns will be transformed into LTL or CTL to check whether the automation rules will be exploited. To test the worst-case performance of SAFECHAIN, we randomly build an always-TRUE specification, such that every reachable state should be visited for verification: $G(a_i = \mathbf{v} \vee a_i \neq \mathbf{v})$, where $a_i \in A$ and $\mathbf{v} \in possible(a_i)$ are both randomly chosen. We randomly select a vulnerable attribute to simulate attacks, and each data point on the figure represents the average of 200 experiment runs. We set a timeout of 30 minutes (1,800 seconds) for each run.

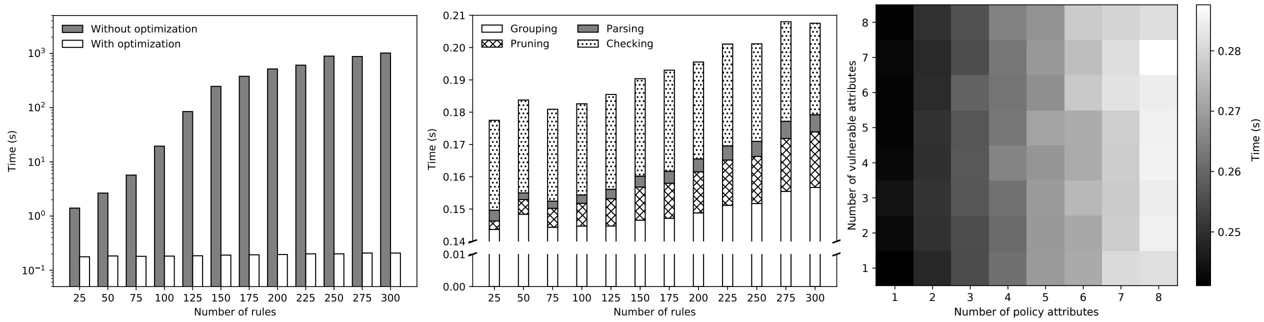
Figure 8a shows the verification time (in a logarithmic scale) with and without our optimization techniques. Without optimization, the time needed for verification increases exponentially with the number of rules. With optimization, SAFECHAIN can verify 300 rules in less than one second. Due to space limitations, we omit similar results of using CTL specifications.

Figure 8b shows the processing time breakdown of optimization. Pruning and grouping are the time needed to perform respective optimizations. Checking time represents how long NuSMV takes to return its results. Parsing time represents the time needed to generate input files for NuSMV and analyze its outputs. The time needed for grouping and pruning only slightly increases as the number of rules increases.

We now evaluate the effects of different numbers of security policy attributes and vulnerable attributes, and the result is shown in the heatmap in Figure 8c. In the heatmap, the execution time is represented by colors; a darker color represents a shorter

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security






(a) Performance.
300 rules can be verified within less than one second with optimization.

(b) Optimization overhead.
The time needed for grouping and pruning only slightly increases as the number of rules increases.

(c) Effects of involved attributes.
Verification time increases slightly as the number of policy attributes increases, while the number of vulnerable attributes has almost no effect.

Fig. 8: Performance evaluation for privilege escalation in LTL

execution time. We fix the number of automation rules to 500 and randomly select one attribute as vulnerable. As shown, increasing vulnerable attributes does not greatly affect SAFECHAIN's performance, because vulnerable attributes irrelevant to policy attributes are pruned. Also, as the number of policy attributes increases, the verification time increases slightly. The results suggest that SAFECHAIN is fast enough to support complex security policies.

**Privacy leakage.** Recall that users can specify what kinds of information are deemed confidential and check whether they will be leaked to any vulnerable attributes. To evaluate the worst-case performance, our evaluation uses $INVAR(a_i^{FSM} = a_i^{CLONE})$ specification for verification, where $a_i^{FSM}$ and $a_i^{CLONE}$ represent the PUBLIC attributes $a_i \in A$ in the original and cloned machines, respectively. The keyword $INVAR$ is a specialized keyword provided by NuSMV to check invariance conditions. Each time the experiment is repeated, we randomly mark an attribute as PRIVATE and another attribute as PUBLIC.

Figure 9a shows the results with and without our optimization. Figure 9b shows the comparison between our optimization and Bounded Model Checking [21] implemented by NuSMV, and Figure 9c summarizes how much time is spent during each step. As shown in these figures, time needed for verifying privacy leakage grows much faster than privilege escalation because we use two finite state machines to achieve pairwise comparisons. Nonetheless, these results prove that SAFECHAIN is still efficient enough to detect attacks in time.

Figure 9d shows SAFECHAIN's performance with respect to different numbers of marked attributes. The execution time is visualized by colors, and a darker color represents a shorter execution time. The experiment setup is the same as privilege escalation. We fix the number of automation rules to 500, randomly choose different numbers of PRIVATE and PUBLIC attributes, and take the average over 1,000 experiments.

The figure shows that the increase of only PUBLIC or PRIVATE attributes has little impact on the performance of SAFECHAIN, because those irrelevant attributes are pruned through optimizations. On the other hand, as the number of both PRIVATE and PUBLIC attributes increases, the probability of dependency between attributes boosts, and thus fewer attributes can be pruned. Further, since we built a product machine for pairwise comparisons of traces, the more attributes left, the longer time needed for verification. In this case, SAFECHAIN can still verify 500 automation rules within five seconds and outperform the baseline without optimizations.

### 6.4 Accuracy Evaluation

We evaluate attack detection accuracy of the verification part only. That is, we assume the devices considered vulnerable are indeed compromised, and the environment models are accurate. Also, for simplicity, we let $window(a_i) = possible(a_i)$.

**Dataset.** Our synthetic dataset shares the same channels and attributes with those used in §6.3. In the dataset, the original triggers and actions are removed, and one trigger and one action are added for each attribute. The trigger is in the form of $a_i = X$ and the action is $a_i \leftarrow Y$. Both $X$ and $Y$ are configurable parameters.

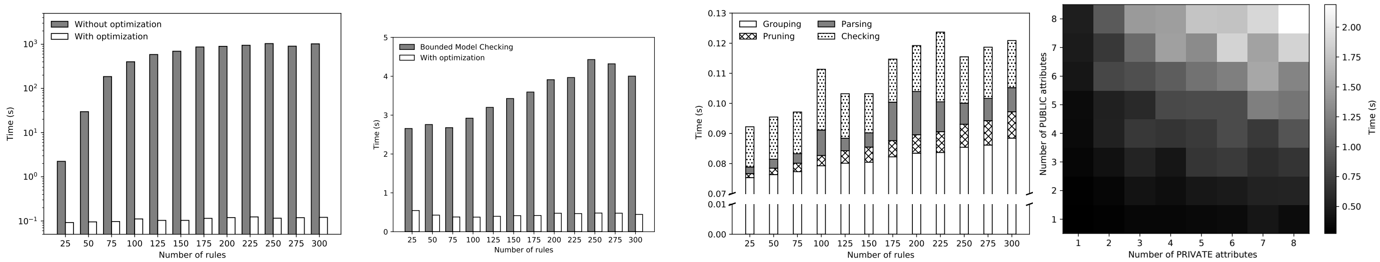
For each attack class, we conduct two experiments. These two experiments are designed to ensure that the ground truth is known, such that we can correctly identify false positives and false negatives.

The first one is to evaluate the false negative rate of our system. We first construct an attack chain $R_1, R_2, \ldots R_l$ of length $l$ randomly, where $l$ is an integer chosen between 2 and 8. If the action of $R_i$ is "$a_i \leftarrow X$", then the trigger of $R_{i+1}$ will be "$a_i = X$". We also add another 50 rules, of which the triggers and actions are randomly chosen.

The second experiment compares SAFECHAIN with previous work [42] that uses static analysis to show that our system can avoid false positives in static analysis. The attack chain is similar to that of the one in the first experiment. The only difference is that, if action of $R_i$ is "$a_i \leftarrow X$", then the trigger of $R_{i+1}$ will be "$a_i = Y$" and "$X \neq Y$." We expect previous work to consider such a chain to be exploitable, since they statically label triggers and actions without considering user-supplied arguments or attribute values. Note that in this experiment we did not add additional rules to avoid accidentally forming another attack chain.

We run each experiment 1,000 times, and the results and detail configurations (e.g., security policies) are described below.

**Privilege Escalation** We mark the attributes associated with the trigger of $R_1$ as vulnerable, and the security policy is the negation of $R_l$'s action. Rules that are not on the chain are added with care, so that the last action can only be triggered through the chain.




(a) Performance. 300 rules can be verified within less than one second with optimization.

(b) Performance (BMC). With optimization, SAFECHAIN is 3-4x faster than BMC.

(c) Optimization overhead. The time grows slightly faster compared to 8b due to the use of a product machine.

(d) Different numbers of marked attributes. The time increases only when the numbers of PUBLIC and PRIVATE variables both increase.

Fig. 9: Performance evaluation for privacy leakage

The results are as expected. SAFECHAIN reported the attack traces for all the test cases in the first experiment and verified all test cases as secure in the second experiment.

**Privacy Leakage.** We label the attribute associated with the trigger of $R_1$ as PRIVATE, and the attribute associated with the action of $R_l$ as PUBLIC. Other attributes are all labeled OTHER.

For the first experiment, SAFECHAIN detects the attack chain in all 1,000 test cases, which implies that there is no false negative in our system. For every test case in the second experiment, our system also reported no attack chain.

The experimental result demonstrates that SAFECHAIN has no false negative as long as we have an accurate model. It also shows that our approach outperforms static analysis in specific cases, as we take runtime values into account.

## 7 Discussion and Limitations

In this section, we discuss several research directions to further improve SAFECHAIN.

**Adversary with partial information.** To achieve a high level of security, SAFECHAIN assumes a strong attacker who knows all of the automation rules specified by users, and thus can also defend against a weaker attacker who knows only a subset of the rules. An interesting future direction is to gauge an automation system's level of security based on how much information the attacker needs to launch a successful attack; it can be considered secure if the attacker is required to know more than a threshold number (e.g., 100) of automation rules.

**Relaxing privacy notions.** In this paper, we consider a relatively strict definition of privacy leakage: any two states which are indistinguishable by attackers should stay indistinguishable in the future. By satisfying this definition, one can prevent high-valued attributes from leaking any information to low-valued variables. This definition is relatively strict because it does not quantify the amount of information leakage and thus cannot differentiate a 1-bit leak from a 100-bit leak, despite the fact that the latter is worse than the former. One interesting research direction is to consider an analogy of an anonymity set, and quantify the level of privacy based on the number of indistinguishable traces on the finite state machine. We leave it as future work to develop and evaluate such relaxed definitions of privacy.

**Environment modeling.** In addition to explicit dependencies introduced by automation rules, there are implicit dependencies enabled by proximity and environmental changes, which can also chain automation rules together. For example, the rule "if the temperature is too low, turn on the heater" seems to be unrelated to the rule "if the temperature is high enough, open the window". However, there might be a hidden relation, "turning on the heater will increase the temperature", which is not explicitly specified in the automation rules, but links the above two rules together. Another example is that switching on bulbs can trigger light sensors in the same room but not in other rooms. Lacking information about such implicit dependencies may cause false alarms and undetected attacks due to missing transitions in the finite state machine.

Building an accurate environment model is challenging even with extensive domain knowledge. SAFECHAIN tries to mitigate this by focusing on the most likely scenarios in the near future and frequently re-calibrating based on the current environment state. Moreover, while it is impossible to fully model an environment, SAFECHAIN can benefit from additional information that helps reconstruct missing transitions. For example, additional information (e.g., the location of each device, and implicit relationships between attributes) can be provided by the users or automatically discovered by machine learning.

**False positives and false negatives.** Because each counterexample reported by the model checker is indeed a feasible attack trace with respect to the model, our scheme should have no false positives in the ideal case. However, whether the attack can really be conducted in the real world depends on how accurately we model the environment. For example, if an attack can only happen when the temperature is 100 degrees Celsius, then it is very likely to be a false positive in the real world.

Inaccurate modeling can also cause false negatives. One example is when a possible state does not show up in the model, and the other is missing transitions, such as the case of implicit dependencies discussed in the previous paragraph.

Another explanation for false negatives is when the model checker runs out of time. This rarely happens after optimization is performed, as our experiment results show.

**Tuning the re-checking interval.** Because an environment is non-trivial to model, we decided to re-verify the automation rules as the environment changes: We first predict how the environment will evolve during a short period of time, and then verify the automation rules with respect to this prediction to make sure attacks cannot be successful momentarily. After the prediction expires, we make another round of prediction and verification again. Because a shorter interval indicates a more accurate environment model, a rule of thumb is setting the interval to be the maximum time required to perform verification, which is about 1s in our implementation. This interval can be dynamically adjusted according to the verification speed.

**Selecting vulnerable devices.** Using the vulnerability database is one approach to determine the devices that are likely to be vulnerable, which is one of the inputs to our system. There are other approaches to achieve this as well. For example, users can manually select devices that require high level of protection, and our system can evaluate whether they may be attacked by assuming that the other devices are vulnerable.

Because the adversary's goal is to control an uncompromised device or leak data from an uncompromised device using automation rules, it is reasonable to consider cases where some devices are vulnerable and some are not. It is outside the scope of this work to consider an attacker that controls all devices, because this attacker succeeds immediately even without trigger-action rules.

**Limitations.** The security of our system largely depends on external sources such as the environment modeling, security policies and the list of vulnerable devices. The reported 100% accuracy in the evaluation sections means the system will never violate the given security policies. However, whether these policies cover all the security aspects are outside the scope of our system.

## 8 Related Work

The most closely related studies are discussed in §6.1. This section reviews other related work.

**End-user programming.** Ur et al. [43], [44] investigated the practicality of end-user programming in the trigger-action paradigm ("if trigger, then action") and collected 224,590 rules shared publicly on IFTTT. We borrowed their dataset for our large-scale evaluation. Mi et al. [36] conducted an empirical characterization of IFTTT, including its ecosystem, usage and performance. They ran a self-implemented IFTTT service in order to demystify the interaction between a partner service and IFTTT engine. Their results show that the current implementation of IFTTT uses a polling method for triggering, which justifies our assumption in §4.2.

**Securing service provider.** DTAP [26] explored the over-privilege problem in the IFTTT platform and found that 83% of examined channels lack the support of fine-grained scoping, and 75% of the tokens are granted more access than required to support triggers and actions. DTAP then proposed a decentralized trigger-action platform that prevents over-privileged service providers by using transfer tokens. DTAP addresses an orthogonal problem, and thus can be directly combined with our system to further enhance the overall security of automation platforms.

**Intention mismatching.** Huang and Cakmak [28] investigated common inconsistencies human users exhibit when interpreting and creating trigger-action rules. Their results confirm the need to verify whether the actual behaviors of user-generated rules match their high-level intentions. By analyzing corresponding actions, TrigGen [37] automatically suggests missing triggers in trigger-action rules that are composed incorrectly by users. Instead of fixing incorrect rules due to mismatched user intention, our work focuses on identifying rules that might be exploited when devices are compromised.

**Conflict resolution.** Ma et al. [33] proposed a watchdog architecture for detecting and resolving rule conflicts in the context of smart cities based on simulation. In addition to achieving conflict detection and resolution, CityGuard [32] further allows one to specify safety requirements for a city. These work focus on improving reliability and thus do not consider the presence of adversaries.

**SCADA and IoT.** Cárdenas et al. [22] studied the security issues in process control and SCADA systems and proposed to detect stealthy attacks by incorporating the knowledge of physical systems. They built a linear model that captures the nature of the physical system and detect attacks using change detection algorithms. However, building a precise model for every physical system is impractical. Hence, our work aims to model and verify the environment as it changes. To compare different attack detection approaches under different experiment settings, Urbina et al. [45] proposed a new metric to quantify attack detection algorithms and found that stateful detection methods outperform stateless ones.

ContexIoT [29] enhances IoT applications with context-dependent access control capabilities. ContexIoT checks whether an action can be executed based on the current context and asks the user to decide if the context is unclear. ContexIoT focuses on runtime enforcement; our work is a prevention system that eliminates potential threats before they are executed.

ProvThings [46] provides data provenance to diagnose the trace and root cause of behavior. However, data provenance is useful only in forensic analysis *after* an attack. While ProvThings can be extended to support dynamic policy enforcement based on the provenance of system events, which will notify a user when a possible attack occurs, SafeChain can detect an attack *before* it actually happens and prevent it by removing vulnerable rules.

**Temporal logics.** Dimitrova et al. [25] proposed SecLTL and SecCTL to incorporate information flow properties into temporal logics. These proposed logics suppose a new hide operator, which can be applied to define privacy on a finite state machine. SafeChain adopts a similar definition but also incorporates temporal constraints in the context of smart spaces. Since our definition can be viewed as a special case of the previous one, we can express our problem using ordinary LTL and CTL logics by building a product machine.

## 9 Conclusion

Finding the right balance between convenience and security has been a longstanding battle, and SafeChain is the first attempt to ease this tension in IoT trigger-action programming. By transforming this programming into a model-checking problem and formulating the security vulnerabilities into finite state machines, SafeChain identifies vulnerabilities among automation rules. To overcome the growing complexity between IoT devices and automation rules, SafeChain adopts pruning and grouping to ignore irrelevant devices and combine equivalent states. We anticipate that SafeChain takes a step towards securing custom automation rules in IoT for its further advancement.

## Acknowledgments

This research was supported in part by the Ministry of Science and Technology of Taiwan (MOST 106-2633-E-002-001 and 107-2636-E-002-005-), National Taiwan University (NTU-106R104045), and Intel Corporation. We gratefully thank Tiffany Hyun-Jin Kim and Matthias Schunter for their constructive comments and suggestions in all stages of this project. We also would like to thank Yen-Kuang Chen, Wei Jeng and Yujen Chen for contributing to the user studies, and Shin-Min Cheng, Kai-Chung Wang, Tony Tan and Yu-Fang Chen for their help in the app and model checking.

## References

[1] https://github.com/csienslab/SafeChain-dataset.
[2] "Android Vulnerabilities," http://www.androidvulnerabilities.org/.
[3] "Cybersecurity Vulnerabilities Identified in St. Jude Medical's Implantable Cardiac Devices and Merlin@home Transmitter: FDA Safety Communication," https://www.fda.gov/MedicalDevices/Safety/AlertsandNotices/ucm535843.htm.
[4] "Eclipse Kura," https://eclipse.org/kura/.
[5] "Fridge sends spam emails as attack hits smart gadgets," http://www.bbc.com/news/technology-25780908.
[6] "Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016," https://www.gartner.com/newsroom/id/3598917.
[7] "Getting lost near the Kremlin? Russia could be 'GPS spoofing'," http://money.cnn.com/2016/12/02/technology/kremlin-gps-signals/.
[8] "Hackers can hijack Wi-Fi Hello Barbie to spy on your children," https://www.theguardian.com/technology/2015/nov/26/hackers-can-hijack-wi-fi-hello-barbie-to-spy-on-your-children.
[9] "Households have 10 connected devices now, will rise to 50 by 2020 — ET CIO," http://cio.economictimes.indiatimes.com/news/internet-of-things/households-have-10-connected-devices-now-will-rise-to-50-by-2020/53765773.
[10] "Javaluator Home Page," http://javaluator.sourceforge.net/en/home/.
[11] "Learn how IFTTT works - IFTTT," https://ifttt.com/.
[12] "Microsoft Flow," https://flow.microsoft.com/en-us/.
[13] "New study details a security flaw with Philips Hue smart bulbs," https://www.cnet.com/news/new-study-details-a-security-flaw-with-philips-hue-smart-bulbs/.
[14] "NVD - Home," https://nvd.nist.gov/.
[15] "Remote smart car hacking with just a phone," https://medium.com/@evstykas/remote-smart-car-hacking-with-just-a-phone-2fe7ca682162.
[16] "Smart meters can be hacked to cut power bills," http://www.bbc.com/news/technology-29643276.
[17] "SmartThings," https://www.smartthings.com/.
[18] "The future is this one-click remote for everything in your life," https://qz.com/346767/ifttt-pares-down-its-automation-service-to-prepare-for-the-one-click-smartwatch-future/.
[19] "Zapier: The best apps. Better together." https://zapier.com/.
[20] I. Beltagy and C. Quirk, "Improved semantic parsers for if-then statements," in *Annual Meeting of the Association for Computational Linguistics*, 2016.
[21] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999.
[22] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: risk assessment, detection, and response," in *ACM symposium on information, computer and communications security*, 2011.
[23] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*, 2000.
[24] J. Classen, D. Wegemer, P. Patras, T. Spink, and M. Hollick, "Anatomy of a vulnerable fitness tracking system: Dissecting the fitbit cloud, app, and firmware," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 1, pp. 5:1–5:24, Mar. 2018.
[25] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl, "Model checking information flow in reactive systems," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2012.
[26] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized Action Integrity for Trigger-Action IoT Platforms," in *NDSS*, 2018.
[27] B. Gilles, R. Pedro, and T. Rezk, "Secure Information Flow by Self-Composition," in *IEEE Computer Security Foundations Workshop*, 2004.
[28] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2015.
[29] Y. J. Jia, A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," in *NDSS*, 2017.
[30] C.-j. M. Liang, L. Bu, Z. Li, and J. Zhang, "Systematically Debugging IoT Control System Correctness for Building Automation," in *ACM BuildSys*, 2016.
[31] C.-j. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, "SIFT: Building an Internet of Safe Things," in *ACM IPSN*, 2015.
[32] M. Ma, S. M. Preum, and J. A. Stankovic, "CityGuard: A Watchdog for Safety-Aware Conflict Detection in Smart Cities," in *IEEE/ACM IoTDI*, 2017.
[33] M. Ma, S. M. Preum, W. Tarneberg, M. Ahmed, M. Ruiters, and J. Stankovic, "Detection of Runtime Conflicts among Services in Smart Cities," in *IEEE International Conference on Smart Computing*, 2016.
[34] C. Maternaghan and K. J. Turner, "Policy conflicts in home automation," *Computer Networks*, vol. 57, no. 12, pp. 2429–2441, 2013.
[35] S. Mennicken, J. Vermeulen, and E. M. Huang, "From today's augmented houses to tomorrow's smart homes: new directions for home automation research," in *ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2014.
[36] X. Mi, F. Qian, Y. Zhang, and X. Wang, "An Empirical Characterization of IFTTT: Ecosystem, Usage, and Performance," in *ACM IMC*, 2017.
[37] C. Nandi and M. D. Ernst, "Automatic Trigger Generation for Rule-based Smart Homes," in *ACM Workshop on Programming Languages and Analysis for Security*, 2016.
[38] C. Quirk, R. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing*, 2015.
[39] S. Resendes, P. Carreira, and A. C. Santos, "Conflict detection and resolution in home and building automation systems: a literature review," *Journal of Ambient Intelligence and Humanized Computing*, vol. 5, no. 5, pp. 699–715, 2014.
[40] E. Ronen, C. O'Flynn, A. Shamir, and A.-O. Weingarten, "Iot goes nuclear: Creating a zigbee chain reaction," in *IEEE Symposium on Security and Privacy*, 2017.
[41] S. Soltan, P. Mittal, and H. V. Poor, "Blackiot: Iot botnet of high wattage devices can disrupt the power grid," in *Proceedings of the 27th USENIX Security Symposium*, 2018.
[42] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes," in *WWW*, 2017.
[43] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *ACM CHI*, 2014.
[44] B. Ur, M. Pak, Y. Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman, "Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes," in *ACM CHI*, 2016.
[45] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the impact of stealthy attacks on industrial control systems," in *ACM CCS*, 2016.
[46] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and Logging in the Internet of Things," in *NDSS*, 2018.
[47] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things," in *ACM HotNets*, 2015.

## Appendix A
## Summary of IoT Channels Used

Table 7 summarizes the IoT channels used in our experiment.

## Appendix B
## Dataset Collected from a Real Household

We analyzed a real dataset provided by a smart home owner. The house contains 85 connected devices, including one car, three alarms, one camera, one energy meter, 24 light bulbs, nine motion sensors, nine contact sensors, two smartphones, two presence sensors, 18 switches, four thermostats, six water sensors and five weather stations. These devices are inter-connected through about 70 automation rules. Some rules are designed for security, such as "if a guest arrives, turn on camera" and "if nobody is at home

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2019.2899758, IEEE Transactions on Information Forensics and Security



TABLE 7: 42 IoT Channels Used in the Experiment

| | | |
|---|---|---|
| Smart phone | | Android Battery, Android Device, Android Location, Android Phone Call, Android Photos, Android Wear, AppZapp, Boxcar 2, Phone Call, QualityTime |
| Wearable device | | Lifelog, Nike+ |
| Smart car | | Automatic, Dash, Mojio |
| Security | Camera | Camio, Manything |
| | Door | Garageio, HomeControl Flex |
| Appliances | Smart button | Bttn, Flic, Qblinks Qmote |
| | Oven | GE Appliances Cooking |
| | Water heater | GE Appliances GeoSpring |
| | Gardening | GreenIQ, Parrot Flower Power |
| | Printer | HP Printing |
| | Light bulb | LIFX, Lutron Caseta Wireless, ORBneXt, Philips Hue |
| | Sensors | Nest Protect, Netatmo Weather Station |
| | Air conditioner | Nest Thermostat |
| | Switch | WeMo Insight Switch |
| | Virtual assistant | Amazon Alexa |
| Embedded system | | Adafruit |
| Web services | | Boxoh Package Tracking, ESPN, Instapush, Is It Christmas?, Printhug |

and any motion has been detected in the room, make siren sound." Some are designed for convenience and power saving, such as "turn off the power strip when someone is not at home." The interaction between devices can be too complex to be reasoned manually.

This dataset demonstrated that the devices used in our experiments are similar to those being used in a real smart home. For example, one of the most important categories are the safety and security related devices, such as cameras, door locks, home controls and smoke sensors. We also covered the most common sensors and actuators such as thermostats, weather stations, light bulbs and switches.

Upon analyzing the rules in the dataset, we found some potential attack chains that can be exploited for privilege escalation. For instance, there is one rule "if any family member arrives home, switch to HOME mode," and another rule "if mode transition to HOME, disarm security cameras." By compromising the presence sensor, which might be less secure, the attacker can create an illusion of someone arrives home and finally disarm the cameras. Furthermore, when the mode is switched to HOME, many alarms are also disabled, for example, the rule "if a door is open, make siren sound and send me a message" is executed only when current mode is not HOME. This concludes that it is easy to form attack chains, especially when the number of rules is large, and attack chains pose threats to real-world smart home owners.

## Appendix C
## Feasibility of Example Rules Used in This Work

To verify the feasibility of the example rules used in this work, we ran a user study. We first extracted rules in Table 1 and Table 2, and removed highly similar rules ($n = 17$, hereafter: Group SafeChain). To mix them with an equal number of rules sampled from the IFTTT website (Group Reference), we then used the devices shown in Figure 1 (e.g., smart lock) as keywords to collect the first 15 rules displayed on the returned page for each device. After eliminating duplicates and highly similar rules (e.g., same rules for different brands) from the obtained 90 rules, we randomly sampled 17 rules and created a survey with 34 rules in total ($n = 17$ for Group SafeChain and Group Reference each). There were no overlapping rules in these two groups.

We created a question for each rule, and the question asked a participant to rate his/her willingness to use the rule in a Likert scale from 1 (totally disagree) to 5 (totally agree). The rules were displayed in a randomized order to avoid the sequential effect.

We recruited 108 participants who have prior experience with IFTTT rules using Amazon Mechanical Turk. The survey took approximately 10 minutes and we paid $2.00 for all participants. After eliminating those who did not have sufficient background knowledge on IFTTT rules, we eventually obtained 79 valid responses. Most participants (66, 83.5%) reported to own 1-5 smart home devices, 9 (11.4%) reported to own 6-10 smart home devices, and 4 (4.7%) participants reported to own 11-20 devices. Among the 79 valid responses, the top three popular devices are smart TV (n=36, 45.6%), smart thermostat (n=16, 20.3%), and smart speaker (n=11, 13.92%). The most popular hubs are Amazon Alexa (47 out of 79) Google assistant (25 out of 79), and Samsung SmartThings (22 out of 79). 19% of the participants reported themselves to be extremely experienced and 53.2% report somewhat experienced with IFTTT. In terms of participant demographics, 36.7% of the participants are females, and the majority of the participants are between ages 25 and 34 (54.4%, n=43) and between ages 35 and 44 (24.1%, n=19). 38%, 22.8% and 15.2% of the 79 participants reported to hold a highest degree in bachelor degree, master degree, and some college, respectively.

Based on their responses, we examined the willingness level of individual users as well as combinations of users. We used a T-test to check if these two groups of rules have significant mean difference, and we found some significant difference ($t(78) = -6.04$, $p < .0001$). This result suggests that users are more willing to use the rules in Group Reference than Group SafeChain, which is reasonable because Group SafeChain intentionally includes potential attack chains for demonstration purposes. Based on this result, we are unable to show that people may adopt the rules in Group SafeChain in practice. Hence, we performed further analysis and found that 78.5%, 45.6%, 67.7% and 70.9% of participants are willing (with a Likert score of 3 or higher) to use the four chained rules in Table 1, respectively. This result shows that a large percentage of users may adopt rules that lead to unintentional chained effects in practice.

In addition, IFTTT rules may be created by multiple users that share the same living/working space. To check the willingness to use with multiple users (say a 3-person family), we enumerated all possible combinations of three participants and examined their aggregated willingness by taking the maximum (out of the three) of the Likert scale values. We found that 78% of the combinations of the 3 participants would be willing (with a Likert score of 3 or higher) to use all the rules in Group SafeChain, while 91% of the combinations of the three participants will be willing to use all the rules in Group Reference. In short, this user study results imply that while the rules in Group SafeChain are less appealing than Group Reference to individual users, the likelihood of three or more users in a group to use all the rules in Group SafeChain remains high, implying the practicality of these rules for the real-world scenarios.