SmartPatch: Verifying the Authenticity of the Trigger-Event in the IoT Platform

Bin Yuan, *Member, IEEE*, Yuhan Wu, Maogen Yang, Luyi Xing, *Member, IEEE*, Xuchang Wang, Deqing Zou, and Hai Jin, *Fellow, IEEE*

Abstract—Emerging IoT clouds are playing a more important role in modern lives, enabling users/developers to program applications to make better use of smart devices. However, preliminary research has shown IoT cloud vulnerabilities could expose IoT users to security risks. To better understand the problem, we studied the SmartThings cloud, one of the most popular IoT cloud platforms that support user-defined device automation (SmartApps). Specifically, we found new vulnerabilities in SmartThings that allow attackers to fake events to trigger the SmartApps to operate devices (e.g., open a lock). Exploiting such vulnerabilities, we successfully faked 7 different types of events, which impact 138 (out of 187) SmartThings' official open-sourced SmartApps. To defeat such attacks, we propose an authenticity-verification-based scheme to deny fake events. Moreover, we designed a tool, `SmartPatch`, to help users secure their SmartThings systems. In specific, `SmartPatch` automatically patches the vulnerable SmartApps and Device Handlers (input) and outputs the flawless programs, which are ready for users to deploy in their SmartThings systems. We have made `SmartPatch` publicly available. With the help of `SmartPatch`, we patched all the vulnerable SmartThings' official open-sourced programs (146 SmartApps and 321 Device Handlers). Experiments have shown the compatibility, effectiveness, and efficiency of our proposed approach.

Index Terms—SmartThings platform, IoT security, trigger-action, smart home

1 Introduction

Emerging Internet of Things (IoT) cloud platforms have offered beyond basic convenience functionalities like device control and alert automations. For example, both users or third-party developers are enabled to program the IoT system to automatically and smartly control the IoT devices. Prominent examples of such IoT platforms include SmartThings [1], IFTTT [2], Zapier [3], and Microsoft Flow [4]. Supporting such programmability is the trigger-action based automation scheme. For example, an IFTTT user can create an applet to enable an automation rule like: "If the door is unlocked, send an alert message to my phone". Moreover, SmartThings allows users to install SmartApps [5], [6], which take the events (e.g., device state changes, sun rises and position changes)[1] as inputs to trigger certain actions like sending commands to devices or sending out alert/notification messages.

While IoT platforms provide tangible benefits, the security issues of them have become an important concern. Vulnerabilities in their communication protocols, access control, privacy and side channel risks in these systems have been studied [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25]. In this paper, we focused on the authenticity of trigger-event in the trigger-action based automation of IoT cloud platforms. Specifically, we studied this problem in the SmartThings platform, which is one of the most popular IoT platforms with its mobile app having over 100 million installs [26].

Favored by many software developers, SmartThings platform is centered on the separation of intelligence from devices, which makes it easier to create SmartApps that interact with and across a wide range of devices. Specially, it provides Device Handlers as the virtual representation of physical devices, which parse protocol-specific status messages from real devices and turn them into trigger-events. Then, the events will trigger the SmartApps that subscribe to such events to execute certain actions as defined by the event handler methods in the SmartApps. Further, SmartThings

1. We call these events *trigger-event*s or simply *event*s for short.

- *Bin Yuan is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China. E-mail: yuanbin@hust.edu.cn.*
- *Yuhan Wu, Xuchang Wang, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {2363627402, 867924212}@qq.com, hjin@hust.edu.cn.*
- *Maogen Yang and Deqing Zou are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: 1282860473@qq.com, deqingzou@hust.edu.cn.*
- *Luyi Xing is with the School of Informatics, Computing, and Engineering, Indiana University Bloomington, Bloomington, IN 47408 USA. E-mail: luyixing@indiana.edu.*

Manuscript received 11 Nov. 2020; revised 25 Feb. 2022; accepted 22 Mar. 2022. Date of publication 25 Mar. 2022; date of current version 14 Mar. 2023.
This work was supported in part by the National Natural Science Foundation of China under Grant 61902138, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2019B010139001, and in part by the Shenzhen Fundamental Research Program under Grant JCYJ20170413114215614.
(Corresponding author: Deqing Zou.)
Digital Object Identifier no. 10.1109/TDSC.2022.3162312

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

allows users/developers to write their own SmartApps and Device Handlers to enable user-specific automations.

Motivation. Despite the popularity of SmartThings, the built-in features of SmartThings expose users to many security risks [9], [11], [14], [15], [16], [17], [18], [19], [27], [28], [29]. Specifically, Yuan *et al.* [8] found that during the process of delegating device access from the SmartThings cloud to the Google Home cloud, device information (e.g., deviceId, etc.) could be leaked to a malicious Google Home user (the attacker). The leaked deviceId could then be exploited to send fake events to the SmartThings cloud, enabling the attacker to unlock the victim's August Smart Lock. Moreover, Fernandes *et al.* [11] illustrated that an unprivileged SmartApp could spoof a fake device event for the carbon monoxide detector, leading to false alarms. As we can see, the consequences of such attacks are devastating, endangering both the security and safety of users (e.g., allowing the attacker to break into the victim's home). Hence, it is crucial to pay attention to and mitigate the event spoofing attack in IoT platforms in time.

In matter of fact, recent works have started to look into the problem of event spoofing in IoT platforms. Fernandes *et al.* [11] pointed out that the SmartThings platform does not sufficiently protect events, enabling an attacker to steal the door lock codes. However, unlike [11], we not only identified new vulnerabilities in SmartThings' event management, but also proposed a defense scheme. In specific, we thoroughly studied the trigger-action programming pattern of SmartThings and discovered new methods to fake trigger-events (in addition to that presented in the previous study [11]). Leveraging these vulnerabilities, we conducted Proof of Concept (PoC) attacks and successfully faked seven types of trigger-events, which can all mislead the SmartApps to perform actions that they are not supposed to. We further evaluated the impacts of such attacks and found out that 138 (out of 187) SmartThings official open-sourced SmartApps are affected. The consequences of such attacks range from sending incorrect notification messages to remotely unlocking the victim's door.

To defend such attacks, we present a lightweight signature and verification based scheme to verify the integrity and authenticity of the events received by the SmartApps. Specifically, it generates a signature using the key field values (e.g., event name, event value, etc.) of the event during the event generation phase in the Device Handler and verifies the signature before the handler methods within SmartApps are triggered.

To make the proposed scheme compatible with SmartThings platform and easy to be deployed by the users to secure their IoT systems, we developed a tool, `SmartPatch`, to automatically patch the vulnerable SmartApps and Device Handlers by adding the codes for event signature generation in the Device Handlers and the codes for event verification in the SmartApps. To be more specific, `SmartPatch` takes the vulnerable SmartApps and Device Handlers as input, generates and analyzes the abstract syntax trees (AST) of their source codes to locate the flawed codes, patches them with codes for signature and verification, and finally outputs the patched and flawless SmartApps and Device Handlers, which are ready for users to deploy in their IoT systems. We have made the tool publicly available [30]. SmartThings users can use `SmartPatch` to patch the (potential) vulnerable SmartApps and Device Handlers (e.g., those developed by themselves or from the official/third-party market) and deploy the patched ones in their systems for secure IoT automations.

To evaluate our proposed scheme, we considered three types of attacks, which are event forge attack, event mimic attack and event replay attack (see Section 3.3).[2] Results show that our proposed scheme can successfully defend against all these 3 attacks with negligible overheads. To evaluate the feasibility of `SmartPatch`, we targeted the SmartApps and Device Handlers that are made publicly available by SmartThings. Results show that `SmartPatch` was able to automatically patch 146 (out of 187) SmartApps and 321 (out of 338) Device Handlers successfully. The average processing time with AST analysis is 308.20 ms for the SmartApps and 254.85 ms for the Device Handlers, which is negligible in practice.

With the emergence of trigger-action-based IoT device control automation, little know is, however, whether such automation introduces new risks. As far as we know, we are the first to conduct in-depth and systematic research on the trigger-action IoT platform, from the perspectives of the trigger-event generation, propagation, authentication, and consumption. Our new understanding of the trigger-action based IoT platforms and the proposed defense will lead to better protection of today's IoT applications and provide valuable insights towards securing future IoT systems. We summarize the contributions of this paper as follows:

- We discovered new methods to fake trigger-events in SmartThings, and conducted PoC attacks to successfully fake seven types of trigger-events.
- To defend against such attacks, we proposed an event authenticity verification mechanism, which is compatible and can be deployed without modification to the design of SmartThings platform.
- We developed a tool, `SmartPatch`, to help users easily patch their vulnerable SmartApps and Device Handlers to secure their IoT automations in SmartThings platform.
- We conducted extensive tests to analyze the impacts of our newly discovered vulnerabilities and to evaluate our proposed defense scheme. Results have shown both the severity and prevalence of newly discovered vulnerabilities and the effectiveness and efficiency of our proposed defense scheme.

The rest of the paper is organized as follows. We present the related work and introduce the background of SmartThings platform in Section 2. We describe the threat model, new identified vulnerabilities and PoC (Proof of Concept) attacks in Section 3.3. Section 4 presents the defense against event spoofing attacks. In Section 5, we elaborate on the design and implementation of `SmartPatch`, a tool we built to help users to automatically patch their vulnerable SmartApps and Device Handlers. Performance evaluation is presented in Section 6. We discuss the limitations of our proposal in Section 7. Finally, we conclude this paper in Section 8.

2. All the PoC attacks and evaluation experiments were conducted with our testing SmartThings account and devices, without interfering other users' normal functionalities in the SmartThings platform.

2 Related Work and Background

In this section, we elaborate on the background of the IoT security and the SmartThings platform, as well as the existing problems and some solutions.

2.1 Related Work

IoT Security. As the research on the security of the Internet of Things becomes more and more in-depth, there are a lot of detailed analyses on the security of the Internet of Things in various aspects such as devices, platforms and other aspects [31], [32], [33], [34], [35].

Many security flaws in the IoT devices have been discussed extensively in prior studies [36], [37], [38], [39], [40], [41]. Yu *et al.* [42] proposed that IoT devices may become the entry points into critical infrastructures and can be exploited to leak sensitive information. Kumar *et al.* [33] conducted a large-scale empirical analysis of physical IoT devices, including their vulnerabilities to the given attacks.

On the IoT platform level, access control [43], [44] and privacy issues [45], [46], [47], [48], [49] play an important role in the security of platforms. Fernandes *et al.* [10] discovered that OAuth tokens could be misused to arbitrarily manipulate users' devices and data for Trigger-Action IoT Platforms. Then, they introduced DTAP that can overcome practical challenges. Lee *et al.* [50] found that devices tend to suffer from over-privileged applications due to the coarse-grained access control, and the lack of resource isolation makes Denial-of-Service attacks possible. Therefore, they proposed FACT, which realizes a safe and efficient connection between applications and IoT devices.

In other aspects, Ronen *et al.* [51] described a new type of attack on IoT devices, which exploited their ad hoc networking capabilities via the Zigbee wireless protocol and verified this infection with the popular Philips Hue smart lamps. Zhou *et al.* [12] conducted in-depth research on five widely-used platforms and discovered a series of attacks against smart home platforms. And as more and more security problems are discovered [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], a variety of solutions have emerged [62], [63], [64], [65], [66], [67], such as restricting the network access of devices and using a novel digital forensic framework. Different from these works, we concentrate on the SmartThings platform and aim to deal with fake events.

SmartThings Platform. Closest to our work is a study by Fernandes *et al.* [11], which systematically analyzed the SmartThings platform from five aspects and discovered significant overprivilege flaws. Moreover, they found that the SmartThings platform did not verify the authenticity of the event and pointed out the SmartApps can spoof physical device events as well as location-related events, which may lead to the possible serious consequences, such as opening the door without permission.

Tian *et al.* [17] presented a user-centric and semantic-based authorization design called SmartAuth. It tries to tell users the difference between the function execution of the IoT APPs and the actual behavior, so as to help users make better decisions and avoid excessive authorization. SOTERIA [27] automatically extracts the state model of applications and adopts the model check for property violations to find security errors. Unfortunately, it is a pity that they are mainly concerned with the control of permissions on the platform but did not pay much attention to the solutions of fake events.

Jia *et al.* [16] proposed ContexIoT, which provides a context-based permission system and automatically patches SmartApps to provide rich contextual information at runtime, and finally achieve the goal of helping users perform effective access control. But sometimes malicious logic is not so easy to distinguish and in comparison, `SmartPatch` not only takes users out of the verification process but also accurately guarantees the authenticity of the event.

HoMonit [9] also tries to fix the misbehaving programs problem in SmartThings. However, HoMonit focuses on a different problem of SmartThings, which is to detect the abnormal behaviors of SmartApps from encrypted wireless traffic. Specifically, HoMonit first extracts the Deterministic Finite Automatons (DFA) of the SmartApps from the source codes and texts embedded in the descriptions and user interfaces of the SmartApps. The extracted DFAs represent the normal behaviors of the SmartApps. Then, HoMonit collects the Zigbee and Z-wave traffic between the IoT devices and the SmartThings Hub, and leverages wireless side-channel analysis to infer the state transition of the actual DFAs. At last, HoMonit applies the DFA matching algorithm to compare the actual DFA (inferred from the wireless traffic) and the DFA extracted from the texts of the SmartApp. If the DFA matching fails, HoMonit would send out alert messages to inform the owner that a misbehaving SmartApp is detected. We summarize the differences between HoMonit and our work as follows.

First of all, because the goal of the HoMonit is to detect the misbehaving SmartApps, HoMonit is not able to stop the malicious/abnormal operations to the victim devices, while our proposed solution can stop such operations and avoid actual damages. For example, if the attacker managed to unlock the victim's smart door when the victim is not home, HoMonit can only send alerts to the victim that the door is unlocked unexpectedly, while SmartPatch can patch the vulnerable programs and prevent the door from being unlocked by the attacker. Second, HoMonit can only deal with the partial devices, which are connected to the SmartThings platform through the SmartThings hub via Zigbee protocol or Z-wave protocol, because the detection is based on the analysis of the Zigbee/Z-wave wireless traffics. However, there are many other devices connected to the SmartThings without generating such traffics. For example, the LIFX devices connect to SmartThings through the LIFX cloud and Philips HUE devices connect to the SmartThings through the Ethernet network. HoMonit would fail to detect the misbehaving SmartApps that operate such devices. By contrast, how the device is connected to the SmartThings is irrelevant to our proposed scheme. That is, our proposal can deal with all kinds of devices. At last, some events might have the same wireless fingerprints, such as the *switch.on* event and the *switch.off* event of the same switch. This would introduce false negative to HoMonit. For example, when the normal/expected behavior is to turn off a device, the attack can turn on the device with a malicious SmartApp without being noticed by HoMonit. This problem does not exist in our scheme because the signature of a *switch.on* event would be different from the signature of the *switch.off* event.

In addition, it's worth noting that there are some other security risks such as cross-App interference threats, which were detected when IoT applications that follow the principle of least privilege interplay [68]. To address different issues, researchers have also proposed various solutions. IOTGUARD [15] described behaviors with the help of directed graphs and analyzed whether there were abnormalities to prevent security risks caused by the interactions among devices. SAINT [14] converted the source code into intermediate representations to identify potential threats about data leaks. Ding *et al.* [28] adopted a framework called IoTMon, which can discover any possible physical interactions in the IoT environment and identify high-risk chains of interactions between applications. Compared to these previous works, which mainly focused on the security risks due to different interactions, we aim to defeat the event spoofing attacks.

Spoofing Attacks in Other Services. Spoofing attacks are not new to the information systems. In this subsection, we summarize the previous works that focused on the similar vulnerability in other services.

Voice assistant is another typical cloud service that is vulnerable to spoofing attacks (e.g., hidden/inaudible command attacks). Zhang *et al.* [69] proposed DolphinAttack for speech recognition systems such as Siri, Google Now, which is a completely inaudible attack by modulating voice commands on ultrasonic carriers. This kind of attack can activate Siri to initiate a FaceTime call on iPhone and even manipulate the navigation system in an Audi automobile, which will lead to great security risks. Meanwhile, a defense method using Supported Vector Machine is proposed to deal with the attack. They also studied the feasibility of the attack and the boundary of inaudibility on its basis in their sequent work [70]. Further, Roy *et al.* [71] used multiple speakers and striped segments of the voice signal across them to expand the distance of inaudible attacks and find indelible non-linear traces to resist them. SurfingAttack [72] takes advantage of the solid materials to enable attackers to interact with the voice-controlled device in multiple rounds over a longer distance. In addition, Yuan *et al.* [73] leveraged an open-source ASR system Kaldi [74] to embed voice commands into music, which would be even harder to detect. To prevent spoofing attacks, Meng *et al.* proposed WSVA [75], which checks the consistency between the voice signal and its corresponding mouth movement to distinguish whether it is a spoofed one. Moreover, Watchdog [76] uses a two-step lightweight detecting algorithm to resist inaudible attacks. In addition, it is feasible to identify speakers with the help of millimeter-wave (mmWave) radar, which can capture both vocal cord vibration and lip motion as multimodal biometric technology [77].

With the proliferation of biometric authentication technology (e.g., face, fingerprint), they would also encounter spoofing attacks. In response to this growing threat, many hardware-based approaches (e.g., capture features with the help of sensors), or software-based approaches (e.g., extract features by CNN) are used to detect these spoofing attacks and reduce privacy risks and security issues [78], [79], [80], [81].

Note that spoofing attacks, in a more general matter, also threaten more traditional services like remote data storage [82] and networking applications, such as ARP forgery attacks [83], [84] and IP spoofing attacks [85], [86]. GPS spoofing attacks are also common and harmful [87], [88].

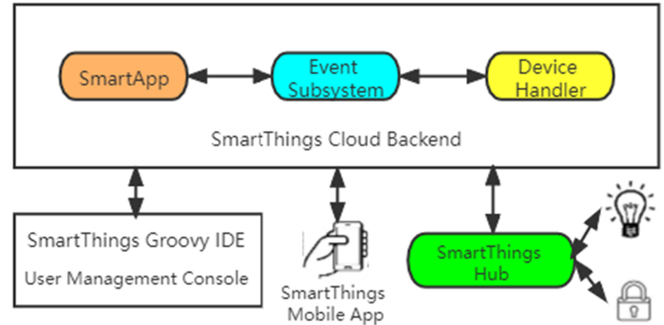


Fig. 1. SmartThings architecture overview.

2.2 Background

In this paper, we focus on the SmartThings, which is a popular smart home platform. SmartThings provides developers with a powerful development framework that enables the abstraction of physical devices and flexible automations.

As we can see in Fig. 1, the SmartThings platform consists of 4 major components: the hub, the cloud backend, the user management console and the companion smartphone app. The hub is an IoT gateway that connects the end devices (e.g., smart lights, sensors and smart locks) to the SmartThings cloud. SmartThings runs the Device Handlers and the SmartApps in its cloud backend with a sandboxed environment. The sandbox is an implementation of a Groovy source code transformation that only allows whitelisted method calls to succeed in the Device Handlers and SmartApps [89]. The Device Handlers are the virtual representations of end devices, which turn the messages generated in the end devices and passed by the hub (e.g., the door is unlocked and the switch is toggled) into trigger-events. The SmartApps are the automation rules that subscribe to events and define the actions to perform when receiving the subscribed events. Both the user management console and the companion smartphone app can be used by the users to manage their smart home systems, such as checking the status of devices and installing/removing SmartApps. In addition, users can upload/develop and install their own SmartApps and Device Handlers through the user management console to define more customized IoT automations.

Listing 1 shows the (demo) code snippet of a switch Device Handler. SmartThings adopts a `capability` system to describe the capabilities that the end devices support [90]. A Device Handler needs to declare its supported `capabilities` and define the functions for each `command` defined in the `capabilities`. Hence, the declaration of `capability "switch"` indicates that its corresponding physical device supports the *switch* capability with *on* and *off* commands.[3] As we can form Listing 1, after changing the status of the device through method `controlDevice()`, the `on()` and `off()` command methods both call the API `sendEvent()` [92] to generate an event to indicate the

3. Different capacities come with different commands. For example, the `capability lock` is defined with the `lock` and `unlock` commands [91].

status change of the underlying device. Meanwhile, the API `parse()` [93] is responsible for handling raw device messages and typically turns such messages into events by calling the API `createEvent()` [94].

Listing 1. A (Demo) Switch Device Handler

```
1  metadata {
2    definition (name: "Switch", namespace: "testing",
         author: "SmartPatch") {
3         capability "Switch"
4      }
5  }
6
7  def parse(String description) {
8    def pair = description.split(":")
9    createEvent(name: pair[0].trim(), value: pair[1].
         trim())
10 }
11
12 def on() {
13   controlDevice("on")
14   sendEvent(name: "switch", value: "on")
15 }
16
17 def off() {
18   controlDevice("off")
19   sendEvent(name: "switch", value: "off")
20 }
```

Listing 2. A SmartApp That Controls the Lock According to a Switch

```
1  definition(
2      name: "SmartApp that controls the lock",
3      namespace: "testing",
4      author: "SmartPatch",
5      description: "A SmartApp that controls the lock",
6      category: "My Apps")
7
8  preferences{
9    input "switch1", "capability.switch"
10   input "lock1", "capability.lock"
11 }
12
13 def installed() {
14   subscribe (switch1, "switch.on", onHandler)
15   subscribe (switch1, "switch.off", offHandler)
16 }
17
18 def onHandler(evt) {
19  lock1.unlock()
20 }
21
22 def offHandler(evt) {
23  lock1.lock()
24 }
```

To specify automation rules, a SmartApp needs to first ask the users to choose the trigger-device (generating trigger-events) and the action-device (being controlled by the SmartApp), which is coded in the `preferences` section of the SmartApp, as shown in Listing 2. Then, the SmartApp has to subscribe to certain trigger-events (lines 19 - 20 of Listing 2) by calling the API `subscribe()` [95] to specify the trigger-device, trigger-event and the event handler method. At last, the SmartApp defines the event handler methods that are called upon receiving of the corresponding trigger-event and calls the command methods (e.g., `lock()` and `unlock()`) defined in the Device Handler of the action-device (e.g., the `lock1` in Listing 2).

To link the devices selected in the SmartApps and their Device Handlers, the `deviceId` [96], the unique system identifier for a device, is used. To be more specific, when users select the trigger-devices and action-devices in the SmartApp, the `deviceIds` of these devices are recorded by the event subsystem. When a Device Handler generates a trigger-event, the corresponding physical device's `deviceId` is included in the event object and passes to the event subsystem, which then passes the trigger-event to the SmartApps accordingly.

It should be noted that SmartThings also supports location events (e.g., *mode* event, *position* event, *sunset* event, *sunrise* event, *sunsetTime* event and *sunriseTime* event) [97] other than the device events (a.k.a., events generated due to status changes of the devices). The location events are typically generated by the system due to users' operations and circadian rhythms. For example, when the user changes the `mode` (e.g., Home and Night) [98] of her home manually, a `mode` event will be generated. When the sun rises, the `sunrise` event and the `sunriseTime` event will be generated automatically by the system. Moreover, both the device events and location events can be subscribed by the SmartApps to trigger the execution of the corresponding event handlers.

3 New Vulnerabilities and Attacks

In this section, we present the new vulnerabilities that we discovered in the SmartThings' event management (e.g., new exploitable APIs to spoof events) and the exploitation of these vulnerabilities (e.g., how to abuse such APIs) with respect to the threat model that we considered.

3.1 Threat Model

Despite the exploitability of the APIs for event spoofing, which we will elaborate on in Section 3.1, there are 2 natural questions for an attacker to successfully conduct such attacks in the real world SmartThings system, which are *Q1*: how to obtain the `deviceId`, which is needed for spoofing device events? and *Q2*: how to install the malicious/attacking SmartApps and Device Handlers, which generate the spoofed events, in the victim's SmartThings system?

For *Q1*, previous studies have pointed out `deviceId` could be disclosed to the attacker through colluding SmartApp, exploiting WebService SmartApps remotely [11] and cross-cloud IoT device delegation [8]. For *Q2*, attackers can upload the malicious SmartApp and Device Handler online (e.g., online user community) and mislead the victim users (who may not be technical-savvy or security experts) to install such malicious programs in their systems. Moreover, in the scenario where the owner (the victim) grants access right to her SmartThings system to another user (the attacker) temporarily by inviting the attacker as a member [99],[4] the attacker would obtain the permission to replace the victim's SmartApps and Device Handlers with malicious ones without being notified by the victim.

Threat Model. Therefore, in this paper, we consider the attacker is a malicious SmartThings user (e.g., a program developer or an Airbnb guest with malicious intentions) that makes full use of his power to obtain the useful

4. This could happen in real world where an Airbnb host delegates access control over her smart home system to an Airbnb guest [8].

information and permission to install attacking SmartApps and Device Handlers in the victim's SmartThings system. We assume the attacker could manage to obtain the `deviceIds` of the victim's devices, read the logs of the victim's SmartThings system and to install attacking programs in the victim's SmartThings system using the various ways discussed above. We also assume that the attacker might be aware of our verification based defense (see Section 4) and conduct more complex attacks, such as replay attack (see Section 3.3), to bypass the defense. In addition, we assume the communications between the devices and hub, the hub and the SmartThings cloud and the internal communication of the SmartThings cloud are secured. That is, the attacker cannot intercept such communications to obtain any useful data. The attacker's goal is to spoof events to trigger the victim's benign SmartApps to perform actions that benefit the attacker, such as opening the victim's door that the attacker has no access to.

It is also worth mentioning that an attacker could perform other attacks. For example, the attacker could use the attacking SmartApp to send arbitrary control commands to the victim's devices directly. However, this type of attack would require the owner/victim's explicit authorization, because SmartApps can only operate the devices that are authorized to it by the owner/victim manually. Hence, to make the attacking stealthier, in this paper, we only consider the event spoofing attacks which can operate the victim's devices maliciously even without the victim's explicit authorization.

3.2 Event Spoof Through API Abusing

As shown in the Appendix A of the supplementary file of this paper, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2022.3162312, SmartThings defines the event object with many fields including `name`, `value`, `deviceId`, etc. [100]. When SmartApps and Device Handlers call certain APIs, event instances will be generated internally by the SmartThings platform and passed to SmartApp event handlers that have subscribed to those events [100]. Note that, the event fields can be set with different values according to the event type. For example, the `deviceId` field has to be set to the identifier of an end device in the *device* events, while it can be set to *null* in the *location* events.

SmartThings exposes multiple APIs that can be called by the SmartApp and Device Handler to generate events. Previous works have pointed out using a malicious SmartApp can spoof location events and device events by calling the `sendLocationEvent()` API [8], [11]. To thoroughly analyze the security implications of such event generation APIs, we manually checked all the related APIs to test whether they are exploitable for event spoofing.

Specifically, to get a full list of event related APIs, we searched for APIs whose name contains the string "event". We ended up with getting 5 APIs, which include 3 APIs exposed to SmartApp (e.g., `sendEvent()`, `sendLocationEvent()`, and `sendNotificationEvent()`) and 2 APIs exposed to Device Handler (e.g., `sendEvent()` and `createEvent()`). Then, we excluded the `sendLocationEvent()` API (for it has already been studied in previous studies [8], [11]) and the `sendNotificationEvent()` API (for it is used to display messages rather than generating events [101]). Then, we wrote testing SmartApps and Device Handlers to call the remaining 3 APIs and found that all of them are exploitable to spoof events.

We summarize the new APIs and the events that they can spoof in Table 1. We now elaborate on how to exploit/abuse these APIs for event spoofing.

TABLE 1
Newly Discovered Exploitable APIs

Type of spoofed event	SmartApp	Device Handler
device event	-	`createEvent()` `sendEvent()`
mode event	`device.sendEvent()`	`createEvent()` `sendEvent()`
position event	`device.sendEvent()`	`createEvent()` `sendEvent()`
sunrise event	`device.sendEvent()`	`createEvent()` `sendEvent()`
sunset event	`device.sendEvent()`	`createEvent()` `sendEvent()`
sunriseTime event	`device.sendEvent()`	`createEvent()` `sendEvent()`
sunsetTime event	`device.sendEvent()`	`createEvent()` `sendEvent()`

Listing 3. Code Snippet of a Fake Device Event

```
1  // spoof a "switch.on" event
2  def parse(String description) {
3      def name = "switch"
4      def value = "on"
5      //overwrite the deviceId to that of the victim
           device
6      def deviceId  = "dfc4ac53-31c8-48f8-b305-
           d95cee1ff90c"
7      device.id = deviceId
8      //parameters of the event object
9      def eventParams =[
10         name: name,
11         value: value,
12         isStateChange: true
13         ]
14     createEvent(eventParams)
15 }
16
17 def on() {
18     def name = "switch"
19     def value = "on"
20     def deviceId  = "dfc4ac53-31c8-48f8-b305-
           d95cee1ff90c"
21     device.id = deviceId
22     def eventParams2 =[
23         name: name,
24         value: value,
25         isStateChange: true
26         ]
27     sendEvent(eventParams2)
28 }
```

Spoof Device Event. The key to spoofing a device event is to set the `deviceId` field of the event to the identifier of the victim device (a.k.a., the device that does not generate the event, but the SmartThings event subsystem believes the spoofed event is generated by it). Recall that, device events can be generated by the Device Handlers of the end devices through calling the `sendEvent()` API or the `createEvent()` API. Further, each Device Handler is associated with a device object instance [102]. When generating the device event instance, the SmartThings platform would set the `deviceId` field of the event to the device's identifier obtained from the device instance (e.g., `device.id`).

Therefore, to spoof a device event, we have to overwrite the identifier of the device object instance with the `deviceId` of the victim device (as shown in lines 5 - 7 and lines 20 - 21 of Listing 3). It is also worth mentioning that the `isStateChange` field needs to be set to *true*, in order to trigger the SmartThings platform to generate and pass the event [103].

Spoof Mode Event. Unlike the device event, the *mode* event is not associated with devices. Therefore, the `deviceId` field of the *mode* event is usually discarded (e.g., set to *null* by default) by the SmartThings platform. Hence, it is not necessary to overwrite the identifier of the Device Handler to spoof *mode* events. Instead, the `name` field has to be set to "*mode*" to indicate the event type, and the `value` field needs to be set to a proper value (e.g., supported modes in the SmartThings platform). As shown in Listing 4, both the `createEvent()` API and the `sendEvent()` API can be abused in the Device Handlers to spoof *mode* events.

Listing 4. Code Snippet of a Fake Mode Event in Device Handler

```
1  def parse(description) {
2      def fakemodechange =[
3          name: "mode",
4          value: "Home"
5        ]
6      return createEvent(fakemodechange)
7  }
8
9  def on() {
10     def fakemodechange1 =[
11         name: "mode",
12         value: "Night"
13       ]
14     sendEvent(fakemodechange1)
15 }
```

Listing 5. Code Snippet of a Fake Mode Event in SmartApp

```
1  def handler(evt){
2      def deviceNetworkID = ""
3      def deviceType = "father␣Switch"
4      def tryaddDevice = addChildDevice("space",
          deviceType,deviceNetworkID)
5      if (tryaddDevice)
6          def idofaddeddevice = tryaddDevice.getId()
7      else
8          log.debug "addchilddevice␣failed"
9
10     def deviceAdded = getChildDevice(deviceNetworkID)
11     if(deviceAdded)
12         def idofswitch2 = deviceAdded.getId()
13     else
14         log.debug "no␣device␣found"
15
16     def fakemodechange =[
17         name: "mode",
18         value: "Home"
19       ]
20     deviceAdded.sendEvent(fakemodechange)
21 }
```

Furthermore, certain SmartApps (e.g., Service Manager SmartApps [104]) may need to add external devices[5] into the SmartThings system and to generate events for these devices by calling the `sendEvent()` API. However, we found that the `sendEvent()` API exposed to the SmartApp is also exploitable for spoofing *mode* events.[6] Similar to that of Device Handler, to spoof *mode* event within a SmartApp, the `name` filed and `value` filed need to be set to proper values (as shown in lines 17 - 18 of Listing 5).

Spoof Other Events. Similar to the *mode* event, the other 5 types of events (e.g., *position* event, *sunrise* event, *sunset* event, *sunriseTime* event and *sunsetTime* event) can also be spoofed through calling `sendEvent()` in the SmartApp and calling `sendEvent()` or `createEvent()` in the Device Handler with the event fields being set to proper values. Please refer to Appendix B of the supplementary file of this paper, available online, for the PoC exploitation codes.

3.3 Event Spoof Attacks

Following the threat model (Section 3.1), we consider the attackers would abuse the exploitable APIs (Section 3.2) to conduct the following three types of attacks.

Event Forge Attack. We define the event forge attack as the normal attack where an attacker simply abuses the APIs with attacking SmartApps or Device Handlers as discussed in Section 3.2. The forge attacks are the easiest type of attack we considered, since the attacker just needs to feed the APIs with the target parameters.

Recall that, we assume the attack might be aware of our defense mechanism. Therefore, we also consider two advanced attacks (see below) where the attacker tries to add more parameters in the APIs to bypass our verification.

Event Mimic Attack. In the case where an attacker is aware of the verification-based defense mechanism proposed in this paper (see Section 4.2.1), the attacker may attempt to bypass the verification by adding a signature to the original event object. We call this attack the event mimic attack. That is, in an event mimic attack, the attacker mimics the behaviors of a patched secure Device Handler (see Section 5), using the key fields of the event to generate a signature and adding it to the original event object. Then, the fake event object with the signature is sent to the SmartThings cloud platform instead of the original event object in the hope that such an event could pass the signature verification and further trigger the victim's benign SmartApps to operate devices that the attacker is not entitled to access.

Event Replay Attack. In more rare cases, the attacker (such as an Airbnb guest who has been invited as a member to the owner's SmartThings system [8], [99]) might be able to access the log of the victim's SmartThings system. Therefore, the attacker could obtain the full body of a real event in the log subsystem, including both the values of the event fields and the signature. Leveraging such information, the attacker could feed such value into the APIs discussed above to replay the event. Without careful design in the defense, such an event would pass the signature verification process and trigger the device actions desired by the attacker.

5. SmartThings supports users to authorize their IoT devices of another platform to SmartThings, so that the user can use SmartThings as a unique control console to manage all her IoT devices from different vendors, such as user's LIFX bulbs being delegated to SmartThings through OAuth [105].

6. Note that the `sendEvent()` API exposed to the SmartApp [106] and the `sendEvent()` API exposed to the Device Handler [92] are two different APIs.

4 Usability Preserving Defense

In this section, we present the defense scheme against event spoof attacks in SmartThings.

4.1 High Level Idea

Root Cause. It should be noted that it is not the faulty codes wrote by the developers, if there were any, that introduce the vulnerabilities discussed in Section 3.3. The developers just program the SmartApps and Device Handlers as instructed by SmartThings (e.g., calling APIs to generate events and operate devices). The fundamental problem is that SmartThings lacks an understanding of the security implication of trigger-events' authenticity and provides no authenticity verification APIs to the developers. Hence, even if the developers understood the security risks, they would not be able to defeat the event spoofing attacks on their own.

A straight forward way to defend against the event spoof attacks in SmartThings is to restrict the use of the exploitable APIs (listed in Table 1). However, these APIs are needed for implementing the functionalities desired by the users and have been widely used in the SmartApps and Device Handlers. To restrict the use of the APIs would significantly reduce the usability of the system. Such a change might even require fundamental re-design and re-deployment of the system and programs, which could take a long time even if it was possible to be carried out in the real world system.

Therefore, we present a usability preserved defense, which can minimize the impact on the SmartThings framework and the users. To this end, we propose the event authenticity verification based defense against event spoof attacks in SmartThings. That is, before calling the event handler method, the SmartApp verifies the authenticity of the trigger-events. Only the events that pass the verification would trigger the actions defined in the event handler methods. To defend against all the three types of attacks (see Section 3.3), we summarize the principles for the authenticity verification as follows.

- *P1: Verifiable signature*. The event should contain a signature that could be used as the proof of non-forged event.
- *P2: One-time trigger*. In a benign environment, each event would only trigger the corresponding event handler methods once. Therefore, we need to check whether a new received event has already been processed by the SmartApp, thus to, for example, defend against event replay attacks.
- *P3: Accordant event attribute*. IoT systems constantly interact with the physical world. Therefore, the attributes of the events should be in accordance with the physical world. For example, the SmartThings system only generates a `sunrise` event during the sunrise time of the user's physical location. Therefore, we also check the accordance of the event attributes for authenticity verification.

4.2 Event Authenticity Verification

In this section, we elaborate on the authenticity verification with respect to the different types of events and the different types of attacks. Specifically, the principles *P1* and *P2* are used for device events verification, while principles *P3* and *P2* for the location events.

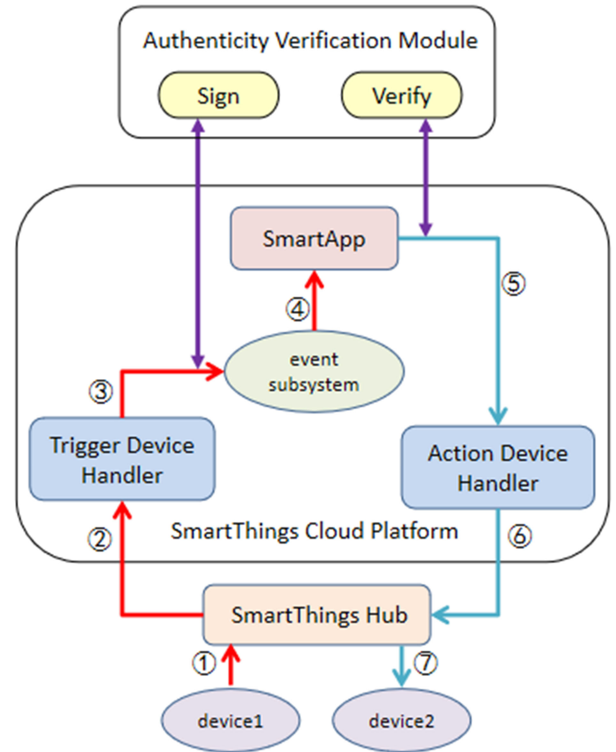


Fig. 2. Signature and verification of device events.

4.2.1 Device Events*: Signature Based Verification*

Lifecycle of a Device Event in SmartThings. As discussed in Section 2.2 and shown in Fig. 2, the typical process of a device event is as follows. First, the state of the end device changes due to its interaction with the physical world (such as, temperature increases or user operates the device physically). Then, the device sends a message to the SmartThings hub to report the state change (①). The hub then forwards the message to the device's Device Handler, which runs in the SmartThings cloud (②). Upon receiving the message, the Device Handler calls the APIs (such as `createEvent()`) and leads the SmartThings system to generate and propagate a device event (③). The event is then passed to the SmartApps that subscribe to it (④). The SmartApp then calls the command methods of the action-device defined in the SmartApp's event handler method (⑤). Then, the Device Handler of the action-device generates and sends command(s) to the hub (⑥). At last, the hub passes the command(s) to the action-device for execution (⑦).

To defend against device event spoofing attacks, we propose a signature based verification scheme to check the integrity of the device event. As shown in Fig. 2, the key idea is 1) to add a signature into the event structure body when calling APIs to generate the event in the Device Handler (③), and 2) to verify the correctness of the signature (contained in the received event) before calling the event handler method in the SmartApp (④). As aforementioned, the three most important fields of a device event are the `deviceId`, the event `name` and the event `value`. Therefore, we use all these three fields to generate the signature and

store the signature in the data field of the event. Hence, the event signature will also be propagated to the SmartApps that subscribe to the event along with all the other event fields like the event name and the event value. As a result, we can verify the correctness of the event signature later in the SmartApps.

Defend against device event forge attack (with principle *P1*). Recall that, in the event forge attack, the attacker simply abuses the APIs to generate a normal event which does NOT contain a signature. Therefore, forged device events would fail to pass the verification. As a result, device event forge attacks can be easily prevented by our proposed scheme.

Defend against device event mimic attack (with principle *P1*). As discussed in Section 3.1, the attacker might be able to obtain the correct fields (e.g., the deviceId, the event name and the event value) and use these fields to generate a valid signature based on these event fields. To prevent the mimic attacks, we need to include an extra secret into the signature which the attacker cannot obtain. Hence, we let the Device Handler generates the data ID0, which is a UUID [107], [108] for the device, and stores the ID0 in its private persistent storage [109]. When generating the signature, the Device Handler sends the ID0 to the *Sign* module (see Fig. 2) along with other fields (line 5 of Listing 6).

Including an ID0 for each device, the device event mimic attacks can be prevented because of the following reasons. First, the ID0 is safely stored in the benign Device Handler's storage in the SmartThings cloud, which is not accessible to the attacker. Second, the plaintext of ID0 is not contained in the event body. Hence, the attacker can't obtain it from the system log. At last, we provide safe management on the ID0. To be more specific, the *Sign* module maintains a mapping between the ID0 and deviceId. Upon receiving a new deviceId, the *Sign* module adds a new item recording the deviceId and the corresponding ID0 in the sign request. Upon receiving a sign request with a known deviceId, it checks whether the request contains a correct ID0, and discards the sign requests with incorrect ID0. Hence, our proposal can also prevent device event mimic attacks.

Defend against device event replay attack (with principle *P2*). Moreover, an attacker could obtain the complete structure of a valid device event (including the valid signature) from the SmartThings logging system and replay the event to mislead the benign SmartApps. Replay attacks are not new to SmartThings system. They have been seen in the smart city systems [110] and automatic speaker verification systems [111]. Timestamp and time to live (TTL), which indicates when the object is created and the period for which the object is valid, based defense against replay attack is commonly used [10]. However, even under non-adversary circumstances, the device events in SmartThings could be delayed. For example, when the hub goes offline because of an Internet outage, the events would be queued at the hub and sent to the SmartThings cloud when the internet is restored [112]. Hence, it is impractical to set a TTL value that could both deny the replayed events and accept the delayed valid events.

To tackle this problem, we present a unique defense against device event replay attack in SmartThings based on the fact that an identical device event would trigger the event handler method(s) defined in the subscribing SmartApp(s) only once (principle *P2*). Particularly, similar to the ID0 for each device, we generate an identification for each event handler method (ID_handler) and store it in the SmartApp's persistent private storage [109]. We also create an identification for each device event (ID_event) and store it in the *Authenticity Verification Module* when generating the signature for the event. Upon receiving a device event, the *Verify* module first checks whether the event contains a valid ID_event. An invalid ID_event indicates a tampered or out-of-date event. Then, the *Verify* module determines whether the event is replayed by checking whether there is an entity of ID_event: ID_handler, the presence of which indicates the event has already triggered the execution of the event handler method.

Note that, the *Verify* module adds and stores the entity of ID_event: ID_handler after successful verification. Also, to avoid storage explosion, the *Verify* module would delete the ID_events and the corresponding entities regularly. For instance, it could only store the information of events from the last 24 hours. An event that is delayed more than 24 hours is rarely seen and it is reasonable to discard such events.

As a result, to defend against all the three device event spoofing attacks above, the SmartApp sends the event name, event value, device identification, event identification, event handler method identification and the event signature (as shown in line 5 of Listing 7) to the *Verify* module for device event authenticity verification.

It should be noted that SmartApps usually respond to events and perform related actions, but in rare cases, the SmartApps may generate device events, too. For example, Service Manager SmartApps [113] are designed to manage devices that connect through the internet (cloud) or the user's local network: the Service Manager SmartApp makes the connection with the device, handling the input and output interactions, and the Device Handler parses messages. In specific, Service Manager SmartApps [104] will add child devices through addChildDevice() [114] and send device events using child.sendEvent() [106]. Since the events generated by the Service Manager SmartApps can also be forged, these events also need to be signed. Therefore, we add a getSignature() method in all the Device Handlers. The Service Manager SmartApp calls the getSignature() to pass the *event type* and *event value* to its child device, which will interact with the *Authenticity Verification Module* to obtain a valid signature. At last, the signature is returned to the SmartApp and is added into the structure of the device event. As a result, the device events generated by Service Manager SmartApps can also be protected.

4.2.2 Location Events*: Attribute Based Verification*

Lifecycle of a Location Event in SmartThings. As discussed in Section 2.2 and shown in Fig. 3, unlike the device events, location events are usually generated by the SmartThings system due to user's operations from the mobile app or circadian rhythms (①). For example, the SmartThings system generates a *position* event when the user changes the location of her smart home in the settings of the SmartThings mobile app. Thereafter, the process of a location event is similar to that of a device event. In specific, the location

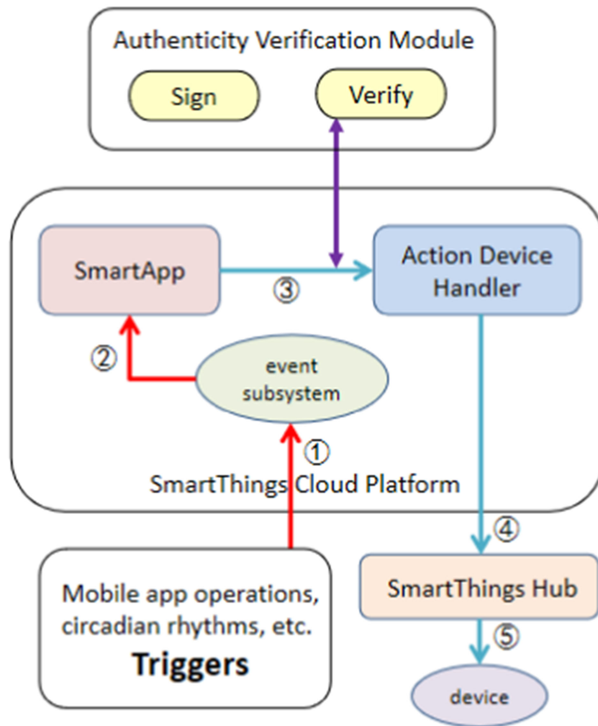


Fig. 3. Verification of location events.

event is propagated to the SmartApps that subscribe to it (②). The SmartApp then calls the command methods of the action-device defined in the SmartApp's event handler method (③), which is followed by the Device Handler of the action-device generating and sending command(s) to the hub (④). At last, the hub passes the command(s) to the action-device for execution (⑤).

The core idea of defending against location event spoofing attacks is the same as that of device event spoofing attacks, which is to verify the authenticity of the events before calling the event handler methods (step ③ of Fig. 3). However, since the location events are generated by the SmartThings internally, we could not add an event signature into the event structure from the outside. Instead, we propose attribute based verification built on our observation of the SmartThings and the rules in the physical world, to defend against location event spoofing attacks.

Defend against mode event and position event spoofing attacks (with principle *P3*). We observed that the `displayName` field of a spoofed *mode* (or *position*) event is always either "mode" (or "position") or the Device Handler's name of the child device, even if the attacker explicitly specifies its value to some other string when spoofing the event. Meanwhile, the `displayName` field of a real and valid *mode* (or *position*) event is always the name of the user's location. As a result, we can identify the spoofed *mode* events and *position* events by checking the values of the `displayName` contained in these events.

Note that, a spoofed *mode* event and/or *position* event will be identified no matter how the attacker spoofs the event (e.g., event forging or event replaying). As a result, such verification can defend against event forge attack, event mimic attack and event replay attack.

Defend against sunset event and sunrise event spoofing attacks (with principles *P3* and *P2*). The *sunset* (or *sunrise*) event is automatically generated by the SmartThings system at the sunset (or sunrise) for the user's location. The time of a sunset (or sunrise) for a specific location is a ground truth, ruled by the physical world.

Hence, to defend against *sunset* and *sunrise* events forge and mimic attacks, we first use the `getSunriseAndSunset()` API [115] to get the local sunset time and sunrise time (the ground truth time). Then, by comparing the arriving time of a *sunset* event (or *sunrise* event) with the ground truth time, we could identify the spoofed events and deny such events (principle *P3*). As to the *sunset* and *sunrise* events replay attack, we simply check whether the event happens more than once in a short period of time after the sunset or sunrise (principle *P2*).

Note that, a real and valid *sunset* event (or *sunrise* event) might be passed to the subscribing SmartApps at the time that is later than the actual sunset (or sunrise) time due to the normal delays in the SmartThings system. Therefore, we add an offset [115], [116] to the ground truth time to tolerate such benign delays. In our current implementation, the offset is set to be 10 minutes. However, it can be easily adjusted according to the users' requirements.

Defend against sunsetTime event and sunriseTime event spoofing attacks (with principles *P3* and *P2*). The *sunsetTime* event and *sunriseTime* event indicate the time of next sunset and sunrise for the user's location, respectively. These two events are generated either around the sunset or sunrise or after the position has been changed by the user. Therefore, defense against the *sunsetTime* and *sunriseTime* event spoofing attacks is similar to that of the *sunset* and *sunrise* event. Event forge and mimic attacks can be defended by comparing the arriving time of the events with the ground truth time (principle *P3*). To get the ground truth time, the `getSunriseAndSunset()` API is used again. We also record the time of the latest *position* as another ground truth time. Denying the replayed *sunsetTime* and *sunriseTime* events is also accomplished by checking whether the event happens more than once in a short period of time (principle *P2*). Offset is also introduced to the ground truth time for tolerating the benign event delays.

4.3 Deployment and Discussion

To enable event signature and verification, we patch the SmartApps and Device Handlers with security hardening codes, as shown in Listings 6 and 7 (We will present the automatic patching tool, `SmartPatch`, in Section 5). However, due to the limitations posed by the SmartThings, in our current design and implementation, we also used an external *Authenticity Verification Module*, which is deployed in our testing server.

To be more specific, SmartThings cloud platform only allows the SmartApps and Device Handlers to use the classes, methods and data types in the specified whitelist [89], which does NOT include the methods for signature and verification, such as `java.security.Signature.sign()` and `java.security.Signature.verify()`. Therefore, we deploy the *Authenticity Verification Module*, which implements the functionalities of signature generation and verification, in a separate web server. The SmartApps and Device Handlers are patched with codes to send/receive HTTP requests/responses to/from the server to obtain the event signature and the result of signature verification.

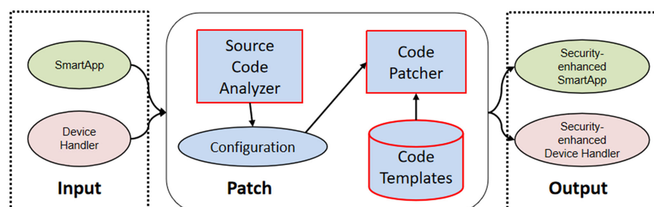


Fig. 4. SmartPatch architecture.

In particular, as shown in line 5 and lines 8 - 16 of Listing 6, the Device Handler sends *sign* requests, which contain the core data of an event (as discussed in Section 4.2.1), to the server. Upon receiving the *sign* request, the *Authenticity Verification Module* checks its data and returns the generated signature to the Device Handler if the request passes the check. Then, the Device Handler puts the received signature into the event and calls the APIs for event generation (line 22 of Listing 6). Similarly, as shown in line 5 and lines 9 - 17, the SmartApp sends *sign* requests, which contain the event signature among other critical data, to the server. The *Authenticity Verification Module* verifies the signature and sends back the verification result to the SmartApp. The SmartApp ends the execution of the event method handler (s) if the event fails to pass the verification (line 18 - 20 of Listing 7).

Note that, the use of a separate server in our current deployment can be eliminated by SmartThings to add the sign-and-verify related methods to the whitelist. We treat our current deployment, which can provide immediate protection to the SmartThings users' smart home systems, as a first step and temporary solution before the SmartThings makes the changes. Besides, even if SmartThings doesn't change the whitelist, our proposed solution can be easily deployed by the users, working with their current smart home systems compatibly and interactively.

5 Automated Patching

In this section, we will elaborate on how to automatically patch the SmartApps and Device Handlers with security hardening codes to enable event signature and verification.

5.1 Overview

To identify the usage of vulnerable APIs and add codes for security enhancement are quite challenging for users, most of which might not be security experts or even technical-savvy. Therefore, it is important to help users to automatically patch their SmartApps and Device Handlers.

Architecture. To this end, we built `SmartPatch` that includes 3 core components: a *Source Code Analyzer*, a *Code Patcher* and a *Code Templates Database*, as outlined in Fig. 4. More specifically, `SmartPatch` takes the vulnerable SmartApps and Device Handlers as input, analyzes their source codes (e.g., locating the vulnerable APIs), automatically patches them and outputs the security-enhanced SmartApps and Device Handlers. With the help of `SmartPatch`, even an IoT newbie user can easily patch their IoT programs and strengthen their smart home systems in SmartThings. We have made the tool publicly available [30].

5.2 Examples of Patched Programs

In this section, we use two examples to illustrate how SmartApps and Device Handlers are patched with `SmartPatch`. We call the SmartApps and Device Handlers before patching original SmartApps and original Device Handlers, while the ones after patching the patched SmartApps and patched Device Handlers.

Example 1: Patched Device Handler. As aforementioned, Device Handlers could use the `sendEvent()` API to pass parameters to the SmartThings system for event generation. As shown in lines 18 - 19 of Listing 6, the original Device Handler calls `sendEvent()` with the event name ("switch") and event value ("on") parameters to generate a *switch.on* device event. To patch the Device Handler, we need to add a device event signature into the event structure and register the device's `ID0` in the *Authenticity Verification Module*.

As we can see from Listing 6, the key parameters of the event (e.g., "switch" and "on") are extracted from the original Device Handler (line 19). These key parameters are then used to generate the device signature (line 5) and reused as parameters in the patched Device Handler's call to the `sendEvent()` API (line 22). The call to this API in the original Device Handler is commented out (line 19). Also, the codes for sending *sign* request to and receiving event signature from the *Authenticity Verification Module*, as mentioned in Section 4.2.1, is added into the patched Device Handler (lines 1 - 16).

Listing 6. Code Snippet of a Patched Device Handler (Generating Signature for Device Event)

```
1  // The added signature code:
2  params = [
3      uri: state.URL,
4      path: state.PATH,
5      query: [ID0: state.ID0, deviceId: device.id, func
           : "sign", name: "switch", value: "on"]
6  ]
7
8  signatureResult = null
9  try {
10     httpGet(params) { resp ->
11         signatureResult = resp.data
12         log.debug "response data: ${signatureResult}"
13     }
14 } catch (e) {
15     log.error "something went wrong: $e"
16 }
17
18 //orignal sendEvent is:
19 //sendEvent(name: "switch", value: "on")
20
21 //new sendEvent is:
22 sendEvent(name: "switch", value: "on", data: [sign: "
       ${signatureResult}"])
```

Example 2: Patched SmartApp. As shown in Listing 7, the original SmartApp tries to unlock the *lock1* (line 23) upon receiving the trigger-device-event. The codes for device event verification are added (lines 2 - 21) to patch the SmartApp. More specifically, the *verify* request is sent out first (lines 2 - 17). Then, the patched SmartApp checks the result of the verification, blocking the execution of the following actions if a "false" verification result is received (lines 18 - 20).

For clarity, we only show the core added codes in the above 2 examples. For the patched SmartApps and Device Handlers to be executable in the users' real-world smart

home systems, some supporting codes are also added, such as the codes to generate and store the ID0, the URL of the external server, and the ID_handler. Please refer to the GitHub page of SmartPatch [30] for the complete patched SmartApps and Device Handlers.

Listing 7. Code Snippet of a Patched SmartApp (Verifying Device Event)

```
1  // The added verification code:
2  def params = [
3      uri: state.URL,
4      path: state.PATH,
5      query: [func: "verify", name: "$evt.name", value:
          "$evt.value", deviceId: "$evt.deviceId",
          ID_event: ID_event, ID_handler: state.
          ID_handler, sign: signString]
6  ]
7  def verify = null
8
9  try {
10     httpGet(params) { resp ->
11         verify = resp.data
12         log.debug "response data: ${resp.data}"
13     }
14 } catch (e) {
15     log.error "something went wrong verify(): $e"
16     return
17 }
18 if("$verify".contentEquals('false\n')){
19     log.trace "event verification failed..."
20     return
21 }
22 //Below is the original code:
23 lock1.unlock()
```

5.3 Implementation of SmartPatch

We provide implementation details of SmartPatch in this section; its full source code is released online [30]. To patch a SmartApp or a Device Handler, there are 2 major phases in patching SmartApps and Device Handlers automatically. The first phase is to locate the relevant method calls (e.g., the vulnerable APIs calls, event handler method calls and subscription calls, etc.) in the source codes, which is accomplished by the *Source Code Analyzer*. The second phase is to replace the vulnerable APIs calls with security hardening codes, which is done by the *Code Patcher*.

The Source Code Analyzer (*Analyzer* for short). We inspected the API documentation of SmartThings [117], and identifies two categories of APIs/methods that are relevant to our defense: 1) the method calls that generate events, including sendEvent(), createEvent(), and parse() in the Device Handlers and .sendEvent() in the SmartApps; 2) the method calls that process events, including subscribe(), install(), eventhandler() and initialize() in the SmartApps and install() in the Device Handlers.

To strengthen the SmartApps and Device Handlers, we have to locate both of the categories of APIs/methods, which are called the target methods. To this end, the *Analyzer* takes the source codes of the SmartApps and Device Handlers as input and generates an abstract syntax tree (AST) for each SmartApp and Device Handler. Then, the *Analyzer* visits the AST to obtain the information of the target methods, such as the line numbers of the start line and end line of the method. *Analyzer* also extracts the arguments of the target methods with the help of AST analyses. At last, the *Analyzer* generates a Configuration file for each SmartApp and Device Handler to store the key metadata (e.g., the line number of the start line, the line number of the end line, arguments, etc.) of the target methods existing in the SmartApp (or the Device Handler).

It should be mentioned that the *Analyzer* fails to generate AST for some SmartApps and Device Handlers. This is because the SmartApps and Device Handlers are actually instances of abstract executor classes [89], whose complete source codes are not available to us. Therefore, for example, when a Device Handler of a Zigbee device uses the Zigbee related package and methods [118], whose codes are explicitly listed in the Device Handler's source code, the *Analyzer* would fail to generate the AST because of *unable to resolve class physicalgraph.zigbee.zcl.DataType*. To tackle this problem, we use regular expressions to search for the strings of the target methods in the source code of this type of SmartApps and Device Handlers. Once the target methods are located, we use the same method discussed above to obtain and store the other metadata of the target methods.

The Code Templates Database and *the Code Patcher* (*Patcher* for short). As discussed in Section 5.2, we have to add various codes when patching the SmartApps and Device Handlers. Some of the codes are the same in different SmartApps or Device Handlers, such as the codes to determine whether to continue the execution of the event handler method (lines 18 - 21 of Listing 7) and the codes to generate the ID0, ID_handler, and ID_event. We call these codes static codes. For efficiency, all the static codes are predefined and stored in the *Code Templates Database*, and are reused in the runtime when the *Patcher* patches the SmartApps and Device Handlers. Meanwhile, some codes to be added can only be generated in the runtime, such as the replacement code of vulnerable API calls (line 22 of Listing 6) based on the metadata of the target methods. We call these codes the dynamic codes. Therefore, the *Patcher* takes the source code and the Configuration file of a SmartApp (or a Device Handler) and the *Code Templates Database* as input, adds both the necessary static codes and dynamic codes into the source code of the original SmartApp (or Device Handler), and outputs the security-enhanced SmartApp (or Device Handler).

6 Evaluation

In this section, we measure the impacts of the spoofed events and evaluate the performance of our proposed defense from the aspects of security benefits, practicality and efficiency.

6.1 Impacts of Spoofed Events

We inspected all the 187 SmartApps open-sourced by SmartThings [6] to investigate how each SmartApp is affected by the event spoofing attacks. Mainly, we looked into the trigger-event(s) that each SmartApp subscribes to and the action(s) that each SmartApp executes upon receiving the trigger-event(s). In particular, we counted the number of SmartApps that subscribe to each type of the events and summarized the (representative) corresponding consequences (e.g., the actions of the SmartApps being affected). The results are shown in Table 2.

TABLE 2
The Number of SmartApps Impacted by Faked Events and the Corresponding Consequences

Event type	# of SmartApps	Actions of SmartApps
device	127	- Controlling devices including locks, cameras and switches- Sending alert messages
mode	22	- Changing the states of the devices such as lights and speakers
position	3	- Changing the mode - Controlling devices such as lights
sunrise sunset	3	- Changing the mode - Changing the states of sensors
sunriseTime sunsetTime	3	- Changing the mode - Controlling devices such as dimmers

Of all the SmartApps affected by the attacks, 12 of them subscribe to both mode *event and* device *event, 2 of them subscribe to* device *event,* position *event,* sunriseTime *event and* sunsetTime *event at the same time, 1 of them subscribes to* position *event,* sunriseTime *event and* sunsetTime *event at the same time, and 3 of them subscribe to* device *event,* sunrise *event and* sunset *event simultaneously.*

Prevalence of Vulnerable SmartApps. Of all the 187 SmartApps we inspected, 138 (over 73%) SmartApps are vulnerable to event spoofing attacks. Only a small number of SmartApps are not affected by the attack, because these SmartApps do not subscribe to any event. To be more specific, 127 (out of 138) vulnerable SmartApps subscribe to device events and are vulnerable to device event spoofing attacks. Meanwhile, 28 SmartApps are vulnerable to location event spoofing attacks. Besides, there are 8 other SmartApps that do not subscribe to events but are also the victims of event spoofing attacks: these 8 SmartApps generate (their child) device events, which could be spoofed by the attacker.

Scope of the Impact. As discussed in Section 3.1, the goal of event spoofing attacks is to mislead the victim's benign SmartApps to perform actions that benefit the attacks. Therefore, the consequences of the attacks actually depend on the actions defined in the vulnerable SmartApps. According to our investigation result, the consequences are devastating, ranging from sending incorrect alerts [119] to unauthorized device access, such as controlling the victim's smart locks [120] and cameras [121].

6.2 Performance Evaluation

In this section, we evaluate the performance of our proposed approach in terms of security benefit, practicality and efficiency.

6.2.1 Security Benefit

In this section, we elaborate on how our proposal defends against the event spoofing attacks, thus to provide security benefits to the SmartThings users' smart home systems.

Defending Against Device Event Spoofing Attacks. Our proposed defense can defend against all the three device event spoofing attacks. As shown in Fig. 5, the *device event forge attack* was identified and stopped by the *Authenticity Verification Module* because there is no signature in the event. Fig. 6 illustrates a *device event mimic attack* was blocked because the event contains an incorrect pair of `ID0` and `deviceId`. Also, as shown in Fig. 7, a *device event replay attack* was stopped because the *Authenticity Verification Module* found out the identical event has already been processed by the same event handler method.

trace No signatrue in event

Fig. 5. Stop the device event forge attack.

debug response data: Error: wrong ID0 for deviceID

Fig. 6. Stop the device event mimic attack.

Error: event 354a262d-3617-420d-85a2-ffbbaf191828
has already been processed by handler
53614c1a-f07b-4e3a-af99-3fccfbb4e479 (replay attack)

Fig. 7. Stop the device event replay attack.

Defending Against Location Event Spoofing Attacks. Our proposed defense can also defend against all the location event spoofing attacks. As shown in Figs. 8 and 9, a *mode event spoof attack* and a *position event spoof attack* were stopped because of the incorrect value of `displayName`. We illustrate how *sunrise event spoof attack* and *sunset event spoof attack* are stopped in Fig. 10: the *sunrise* event didn't happen at (or near) sunrise time. Similarly, as shown in Fig. 11, *sunriseTime event spoof attack* and *sunsetTime event spoof attack* can be successfully defended because they happen at a wrong time.

6.2.2 Practicality

In this section, we evaluate the practicality of our proposal in two aspects: 1) can the patched SmartApps and Device Handlers work compatibly within the SmartThings system? and 2) can `SmartPatch` patch all the SmartApps and Device Handlers automatically?

Secured and Compatible Device Event Processing With Patched SmartApps and Device Handlers. It is critical for the patched programs to work compatibly, interactively and correctly with the SmartThings users' smart home systems. To show that, we present the execution results of the patched SmartApp named "The Big Switch", which is provided by SmartThings for users to "*turns on, off and dim a collection of lights based on the state of a specific switch*" [122].

We first patched the "The Big Switch" SmartApp and the Device Handler of a switch with `SmartPatch` and installed the patched programs in our testing SmartThings system. We configured the SmartApp to turn on/off the lights in the kitchen, living room and bedroom according to the switch. The execution results are shown in Fig. 12. In specific, as discussed in Section 4.2.1, the processing of device event is protected by signing the event at generation and verifying the signature at triggering actions. Fig. 12a presents how the *Authenticity Verification Module* provides such protection and returns a result of successful verification. Meanwhile, as shown in Fig. 12b, the patched SmartApp receives a *switch.off* event, sends out the *sign* request, and receives a successful event verification result from the *Authenticity Verification Module*. As a result, the SmartApp executes the

trace evt.displayName is mode, it shoudl be location01

Fig. 8. Stop *mode* event spoofing attack.

trace evt.displayName is position, it shoudl be location01

Fig. 9. Stop the *position* event spoofing attack.

evt.date is Wed Sep 30 07:39:59 UTC 2020, the sunrise event should happen near Tue Sep 29 22:15:00 UTC 2020 (from Tue Sep 29 22:10:00 UTC 2020 to Tue Sep 29 22:20:00 UTC 2020)

Fig. 10. Stop the *sunrise* event spoofing attack.

trace it has been too long since last event (postion, sunrise or sunset)

Fig. 11. Stop the *sunriseTime* event spoofing attack.

```
before signEvent
Key = a5c4372e-e143-4db2-92ea-54144c87329d,
Value (ID0) = c1899ff3-f2c3-4465-b09d-5ab4be427def
deviceID is: a5c4372e-e143-4db2-92ea-54144c87329d
ID0 is: c1899ff3-f2c3-4465-b09d-5ab4be427def
evtName is: switch
evtValue is: off
toBeSinged: switchoffc1899ff3-f2c3-4465-b09d-5ab4b
e427defa5c4372e-e143-4db2-92ea-54144c87329d3c81446
4-1494-4525-af57-6a98eb216ab5
signature: [44, 70, 98, -39, -42, -38, -126, 0, 89,
35, 1, 80, 77, 26, -95, 7, -120, 55, 89, 13, -86, 91,
50, -61, 35, 3, -49, 60, 77, 101, -68, -96, -33, -114,
-100, 97, 44, -120, 38, -12, -107, 8, 89, 6, 106, 42,
27, -94, -27, -76, -58, 45, 25, 103, 43, 101, -126, -64,
-125, -119, 60, -124, 85, -112, 20, 13, -123, 115, 44,
65, 34, -120, -61, 63, -3, -107, 79, -42, 4, 20, -20, 86,
-91, -10, 104, 38, 64, -50, 80, -42, -47, 61, 70, 9, 104,
67, 71, -103, -54, -88, -73, -81, -93, -40, -70, -61, -89,
96, -37, 5, 108, 59, -81, -75, 82, -57, -79, -36, 1, 54,
62, 28, 49, 47, -75, -28, 25, 35]
after signEvent
Key = a5c4372e-e143-4db2-92ea-54144c87329d,
Value (ID0) = c1899ff3-f2c3-4465-b09d-5ab4be427def
before verifyEvent
Key = a5c4372e-e143-4db2-92ea-54144c87329d,
Value (ID0) = c1899ff3-f2c3-4465-b09d-5ab4be427def
eventName is: switch
eventValue is: off
deviceID is: a5c4372e-e143-4db2-92ea-54144c87329d
ID_e is: 3c814464-1494-4525-af57-6a98eb216ab5
ID_h is: 88bf7d26-a652-40a4-b51f-aa6268028bce
passedSignature is: [44, 70, 98, -39, -42, -38, -126, 0, 89,
35, 1, 80, 77, 26, -95, 7, -120, 55, 89, 13, -86, 91,
50, -61, 35, 3, -49, 60, 77, 101, -68, -96, -33, -114,
-100, 97, 44, -120, 38, -12, -107, 8, 89, 6, 106, 42,
27, -94, -27, -76, -58, 45, 25, 103, 43, 101, -126, -64,
-125, -119, 60, -124, 85, -112, 20, 13, -123, 115, 44,
65, 34, -120, -61, 63, -3, -107, 79, -42, 4, 20, -20, 86,
-91, -10, 104, 38, 64, -50, 80, -42, -47, 61, 70, 9, 104,
67, 71, -103, -54, -88, -73, -81, -93, -40, -70, -61, -89,
96, -37, 5, 108, 59, -81, -75, 82, -57, -79, -36, 1, 54,
62, 28, 49, 47, -75, -28, 25, 35]
verify succeeded
```

(a) The generation and verification of device event signature

debug [bedroomLight, kitchenLight, livingroomLight]
debug off
debug response data: true

(b) The log of the security-enhanced SmartApp

Big Switch bedroomLight kitchenLight livingroomLight
10:31:18 debug PUBLISHED off()

(c) The log of the Device Handler of the action-device (bedroomLight)

Fig. 12. The successful execution of patched programs.

actions, for example, to turn off the light in the bedroom (as shown in Fig. 12c).

Feasibility of Automated Patching. Of equal importance is the ability to automatically patch as many programs as possible. To evaluate the feasibility of `SmartPatch`, we downloaded all the SmartThings open-sourced SmartApps and Device Handlers [6], and used `SmartPatch` to patch them. The result shows that `SmartPatch` can automatically patch all of the vulnerable SmartApps and Device Handlers, as shown in Table 3.

To be more specific, since we aim to defend against event spoofing attacks in this paper, only the SmartApps that subscribe to events and Device Handlers that generates events with the exploitable APIs (as listed in Table 1) are vulnerable. We find out that 41 (out of 187) SmartApps and 17 (out of 338) Device Handlers are not vulnerable. Therefore, these programs don't need to be patched. All the other vulnerable (146) SmartApps and (321) Device Handlers can be successfully patched with `SmartPatch`. Further, as we discussed in Section 5.3, AST analysis and string matching are used in the *Source Code Analyzer*. The result shows that 132 (out of 146) SmartApps and 125 (out of 321) Device Handlers are patched with AST analysis, while 14 (out of 146) SmartApps and 196 (out of 321) Device Handlers are patched with string matching.

6.2.3 Efficiency

In this section, we evaluate the efficiency of our proposed approach by measuring the runtime overheads in event processing and the time for automated patching.

Runtime Overheads in Event Processing. The runtime overheads are introduced by the event signature generation in the patched Device Handlers and the event verification in the patched SmartApps. To measure such overheads, we record the time from the Device Handler sending out the *sign* request to the verification result being received by the SmartApp.

As shown in Fig. 13, the runtime overheads are negligible. Most of the performance penalties are less than 900 ms. The first run of a patched SmartApp and Device Handler introduces a bit more latency (around 1000 ms), because the identifications of the entities (e.g., `ID0`, `ID_handler`) need to be generated and stored only in the first run.

It should be noted that the latency we presented in Fig. 13 also includes the network delay between the SmartThings cloud (where the SmartApps and Device Handlers are running) and our testing server that hosts the *Authenticity Verification Module*. Hence, the performance overheads can be further reduced if the server is placed within the SmartThings cloud or SmartThings allows SmartApps and Device Handlers to use the methods for signature and verification (as discussed in Section 4.3).

Time for Automated Patching With `SmartPatch`. It is also important that `SmartPatch` could automatically patch the programs fast, so that the users can use `SmartPatch` to enhance the security of their SmartThings smart home systems conveniently. To measure the time needed for patching

TABLE 3
The Number of SmartApps and Device Handlers Dealed With Different Processing Methods

	# of SmartApps	# of Device Handlers
Patched with AST ananlysis	132	125
Patched with string matching	14	196
Unpatched	41	17
Total	187	338

vulnerable programs, we recorded the time that SmartPatch used to patch 132 SmartApps and 125 Device Handlers [6] with AST analysis.

As shown in Fig. 14, the patching times for most (over 94%) SmartApps are less than 900 ms and the average time to patch the SmartApp is 308.20 ms. Also, the times to patch most (over 95%) of the Device Handlers are less than 900 ms and the average time of the Device Handler is 254.85 ms. The longest time used for automated patching is 5867 ms. This is because the corresponding SmartApp ("Spruce Scheduler" [123]) contains more than 100 functions, which is very rare. Hence, we believe it is effective enough to automatically patch SmartApps and Device Handlers in real world.

7 Discussion and Limitations

First of all, in order not to affect the normal functionalities of the SmartThings cloud platform, we adopt an external server to host the *Authenticity Verification Module*. In our current implementation, the URL of the hosted service is exposed in the source codes of the patched programs. Therefore, an attacker can conduct DDoS attacks to the server to disrupt the verification process. However, SmartThings, as the service provider, can fix this problem easily, for example, to provide internal closed-source APIs that implement the event verification.

Second, under the current stage, a powerful attacker who has the permission to install malicious programs in the victim's SmartThings account and is also aware of our defense mechanism could bypass the proposed verification, by replacing the patched secure programs with unpatched insecure ones. However, once SmartThings adopts our proposal, the unpatched programs (installed by the attacker) will not be able to pass the verification and thus fail to conduct a successful attack.

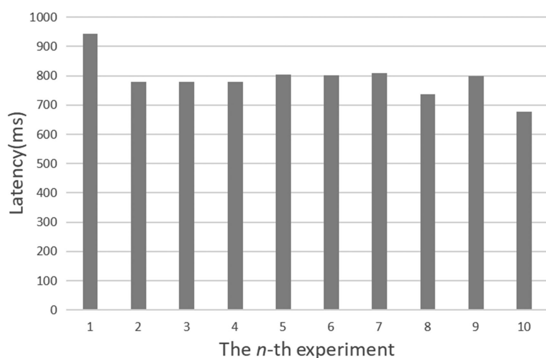


Fig. 13. The runtime overheads (latency) of patched programs.

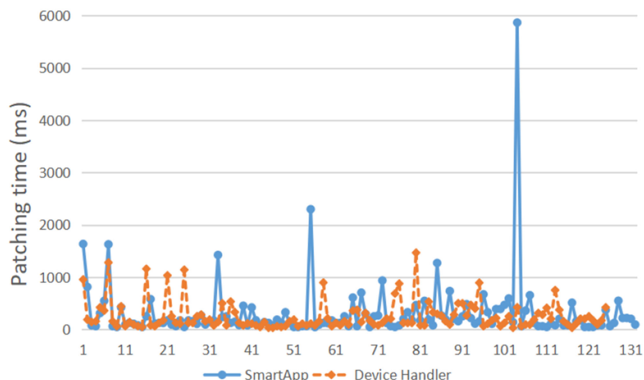


Fig. 14. Time used for automated patching with AST analysis.

Third, the time that SmartPatch needs to patch a SmartApp or Device Handler relies on the size of the codes. More codes indicate longer patching time. However, SmartApps and Device Handlers are usually very simple, with hundreds of lines of codes. Hence, we believe SmartPatch is efficient enough for real-world use.

Fourth, our approach can only strengthen the SmartApps and Device Handlers whose source codes are available to the users. However, the developers of the closed source SmartApps and Device Handlers can also leverage SmartPatch to patch the programs before publishing them.

Lastly, there are many other trigger-action IoT platforms, such as IFTTT, Zapier, and Microsoft Flow. Previous researches have shown that vulnerabilities similar to those identified in this work also exist in other IoT platforms [10], [54]. To solve such problems, Bastys *et al.* [54] proposed to disable the access to sensitive actions via JavaScript in the filter codes wrote by the attacker, while Fernandes *et al.* [10] presented a clean slate design that requires modification to the IoT platform (e.g., IFTTT) and all the connected third-party trigger/action services. Comparing to these solutions, which would take a long time for real world deployment by the related vendors, our solution requires no such modification and can provide immediate protection to the end users. It should be noted that, due to the differences of the design and implementation details of all these IoT platforms, our tool, SmartPatch , can only automatically patch the vulnerable programs in SmartThings. However, our proposed authenticity verification based defense can provide valuable insights in solving the problems in other IoT platforms.

8 Conclusion

In this paper, we focus on the event spoofing attack and its defense in SmartThings, which is one of the most popular IoT platforms. Through systematical analyses, we identified new attacks to spoof events that would mislead the benign SmartApp to execute actions at the attacker's desire. We also present a usability preserved defense based on event verification. We build a tool SmartPatch to help users to automatically patch their vulnerable SmartApps and Device Handlers. Extensive experiments have shown the effectiveness, efficiency and practicality of our proposal. Our new findings and understanding will provide better protection for today's IoT applications and shed light on designing more secure IoT systems in the future.

REFERENCES

[1] Samsung SmartThings. Add a little smartness to your things. Accessed: Jun. 2019. [Online]. Available: https://www.smartthings.com/

[2] IFTTT: Every thing works better together. Accessed: Jun. 2019. [Online]. Available: https://ifttt.com/

[3] Zapier: The easiest way to automate your work. Accessed: Jun. 2019. [Online]. Available: https://zapier.com/

[4] Microsoft Flow. Accessed: Jun. 2019. [Online]. Available: https://flow.microsoft.com/en-us/

[5] SmartApp. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic

[6] SmartThings Git. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThings Public

[7] Y. Jia *et al.*, "Burglars' IoT paradise: Understanding and mitigating security risks of general messaging protocols on IoT clouds," in *Proc. 41st IEEE Symp. Secur. Privacy*, 2020, pp. 465–481.

[8] B. Yuan *et al.*, "Shattered chain of trust: Understanding security risks in cross-cloud IoT access delegation," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1183–1200.

[9] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "HoMonit: Monitoring smart home apps from encrypted traffic," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1074–1088.

[10] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action IoT platforms," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–16.

[11] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *Proc. 37th IEEE Symp. Secur. Privacy*, 2016, pp. 636–654.

[12] W. Zhou *et al.*, "Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1133–1150.

[13] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action IoT platforms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1439–1453.

[14] Z. B. Celik *et al.*, "Sensitive information tracking in commodity IoT," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1687–1704.

[15] Z. B. Celik, G. Tan, and P. D. McDaniel , "IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[16] Y. J. Jia *et al.*, "ContexIoT: Towards providing contextual integrity to appified IoT platforms," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[17] Y. Tian *et al.*, "SmartAuth: User-centered authorization for the Internet of Things," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 361–378.

[18] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the Internet of Things," in *Proc. 25th Annu. Netw. Distrib. Syst. Symp.*, 2018, pp. 1–15.

[19] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical data protection for emerging IoT application frameworks," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 531–548.

[20] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in IoT apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1102–1119.

[21] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–16.

[22] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proc. IEEE Symp. Secur. Privacy*, 1996, pp. 164–173.

[23] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," in *Proc. IEEE Symp. Secur. Privacy*, 2002, pp. 114–130.

[24] K. E. Seamons *et al.*, "Requirements for policy languages for trust negotiation," in *Proc. 3rd Int. Workshop Policies Distrib. Syst. Netw.*, 2002, pp. 68–79.

[25] M. P. Andersen *et al.*, "WAVE: A decentralized authorization framework with transitive delegation," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1375–1392.

[26] Installs of SmartThings App. Accessed: Jun. 2019. [Online]. Available: https://play.google.com/store/apps/details?id=com.samsung.android.oneconnect

[27] Z. B. Celik, P. D. McDaniel, and G. Tan, "SOTERIA: Automated IoT safety and security analysis," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 147–158.

[28] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 832–846.

[29] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. D. McDaniel , "IotSan: Fortifying the safety of IoT systems," in *Proc. 14th Int. Conf. Emerg. Netw. Experiments Technol.*, 2018, pp. 191–203.

[30] SmartPatch. Accessed: Aug. 2020. [Online]. Available: https://github.com/IoT-Security-Tool/SmartPatch

[31] S. Manandhar, K. Moran, K. Kafle, R. Tang, D. Poshyvanyk, and A. Nadkarni, "Towards a natural perspective of smart homes for practical security and safety analyses," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 482–499.

[32] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security evaluation of home-based IoT deployments," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1362–1380.

[33] D. Kumar *et al.*, "All things considered: An analysis of IoT devices on home networks," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1169–1185.

[34] X. Feng *et al.*, "Understanding and securing device vulnerabilities through automated bug report analysis," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 887–903.

[35] H. Wen, Q. A. Chen, and Z. Lin, "Plug-N-Pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new over-the-air attack surface in automotive IoT," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 949–965.

[36] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the mirror: Evaluating IoT device security through mobile companion apps," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1151–1167.

[37] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable BLE IoT devices with static UUIDs from mobile apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1469–1483.

[38] H. M. Moghaddam *et al.*, "Watching you watch: The tracking ecosystem of over-the-top TV streaming devices," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 131–147.

[39] A. K. Sikder, H. Aksu, and A. S. Uluagac, "6thSense: A context-aware sensor-based attack detector for smart devices," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 397–414.

[40] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home devices," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[41] Q. Qi, X. Chen, C. Zhong, and Z. Zhang, "Physical layer security for massive access in cellular Internet of Things," *Sci. China Inf. Sci.*, vol. 63, no. 2, 2020, Art. no. 121301.

[42] 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 *Proc. 14th ACM Workshop Hot Top. Netw.*, 2015, pp. 5:1–5:7.

[43] R. Schuster, V. Shmatikov, and E. Tromer, "Situational access control in the Internet of Things," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1056–1073.

[44] W. He *et al.*, "Rethinking access control and authentication for the home Internet of Things (IoT)," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 255–272.

[45] N. J. Apthorpe, S. Varghese, and N. Feamster, "Evaluating the contextual integrity of privacy regulation: Parents' IoT toy privacy norms versus COPPA," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 123–140.

[46] P. E. Naeini, Y. Agarwal, L. F. Cranor, and H. Hibshi, "Ask the experts: What should be on an IoT privacy and security label?" in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 447–464.

[47] E. Zeng and F. Roesner, "Understanding and improving security and privacy in multi-user smart homes: A design exploration and in-home user study," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 159–176.

[48] H. Yu, J. Lim, K. Kim, and S. Lee, "Pinto: Enabling video privacy for commodity IoT cameras," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1089–1101.

[49] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1381–1396.

[50] S. Lee *et al.*, "FACT: Functionality-centric access control system for IoT programming frameworks," in *Proc. 22nd ACM Symp. Access Control Models Technol.*, 2017, pp. 43–54.

[51] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "IoT goes nuclear: Creating a Zigbee chain reaction," *IEEE Secur. Privacy*, vol. 16, no. 1, pp. 54–62, Jan./Feb. 2018.

[52] P. Morgner, C. Mai, N. Koschate-Fischer, F. Freiling, and Z. Benenson, "Security update labels: Establishing economic incentives for security patching of IoT consumer products," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 429–446.

[53] M. Xu *et al.*, "Dominance as a new trusted computing primitive for the Internet of Things," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1415–1430.

[54] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in IoT apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1102–1119.

[55] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1099–1114.

[56] B. Huang, A. A. Cárdenas, and R. Baldick, "Not everything is dark and gloomy: Power grid protections against IoT demand attacks," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1115–1132.

[57] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler, "JEDI: Many-to-many end-to-end encryption and key delegation for IoT," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1519–1536.

[58] S. Soltan, P. Mittal, and H. V. Poor, "BlackIoT: IoT botnet of high wattage devices can disrupt the power grid," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 15–32.

[59] D. Kumar *et al.*, "Skill squatting attacks on Amazon Alexa," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 33–47.

[60] X. Feng, Q. Li, H. Wang, and L. Sun, "Acquisitional rule-based engine for discovering Internet-of-Thing devices," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 327–341.

[61] J. Chen *et al.*, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[62] I. Zavalyshyn, N. O. Duarte, and N. Santos, "HomePad: A privacy-aware smart hub for home environments," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 58–73.

[63] M. Mohsin, Z. Anwar, F. Zaman, and E. Al-Shaer, "IoTChecker: A data-driven framework for security analytics of Internet of Things configurations," *Comput. Secur.*, vol. 70, pp. 199–223, 2017.

[64] S. Demetriou *et al.*, "HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps," in *Proc. 10th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2017, pp. 122–133.

[65] S. Li, D. Zhai, P. Du, and T. Han, "Energy-efficient task offloading, load balancing, and resource allocation in mobile edge computing enabled IoT networks," *Sci. China Inf. Sci.*, vol. 62, no. 2, pp. 29 307:1–29 307:3, 2019.

[66] R. Trimananda, A. Younis, B. Wang, B. Xu, B. Demsky, and G. H. Xu, "Vigilia: Securing smart home edge computing," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 74–89.

[67] L. Babun, A. K. Sikder, A. Acar, and A. S. Uluagac, "A digital forensics framework for smart settings: poster," in *Proc. 12th Conf. Secur. Privacy Wireless Mobile Networks*, 2019, pp. 332–333.

[68] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks*, 2020, pp. 411–423.

[69] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "DolphinAttack: Inaudible voice commands," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 103–117.

[70] C. Yan, G. Zhang, X. Ji, T. Zhang, T. Zhang, and W. Xu, "The feasibility of injecting inaudible voice commands to voice assistants," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 3, pp. 1108–1124, May/Jun. 2021.

[71] N. Roy, S. Shen, H. Hassanieh, and R. R. Choudhury, "Inaudible voice commands: The long-range attack and defense," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 547–560.

[72] Q. Yan, K. Liu, Q. Zhou, H. Guo, and N. Zhang, "SurfingAttack: Interactive hidden attack on voice assistants using ultrasonic guided waves," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[73] X. Yuan *et al.*, "CommanderSong: A systematic approach for practical adversarial voice recognition," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 49–64.

[74] Kaldi. Accessed: Jun. 2019. [Online]. Available: http://kaldi-asr.org

[75] Y. Meng, H. Zhu, J. Li, J. Li, and Y. Liu, "Liveness detection for voice user interface via wireless signals in IoT environment," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 6, pp. 2996–3011, Nov./Dec. 2021.

[76] J. Mao, S. Zhu, X. Dai, Q. Lin, and J. Liu, "Watchdog: Detecting ultrasonic-based inaudible voice attacks to smart home systems," *IEEE Internet Things J.*, vol. 7, no. 9, pp. 8025–8035, Sep. 2020.

[77] Y. Dong and Y.-D. Yao, "Secure mmWave-radar-based speaker verification for IoT smart home," *IEEE Internet Things J.*, vol. 8, no. 5, pp. 3500–3511, Mar. 2021.

[78] T. Chugh and A. K. Jain, "Fingerprint spoof detector generalization," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 42–55, 2021.

[79] T. Chugh, K. Cao, and A. K. Jain, "Fingerprint spoof buster: Use of minutiae-centered patches," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 9, pp. 2190–2202, Sep. 2018.

[80] Y. Jia, J. Zhang, S. Shan, and X. Chen, "Single-side domain generalization for face anti-spoofing," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 8481–8490.

[81] Z. Wang *et al.*, "Deep spatial gradient and temporal depth learning for face anti-spoofing," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 5041–5050.

[82] G. Xu, Y. Yang, C. Yan, and Y. Gan, "A probabilistic verification algorithm against spoofing attacks on remote data storage," *Int. J. High Perform. Comput. Netw.*, vol. 9, no. 3, pp. 218–229, 2016.

[83] V. K. Tchendji, F. Mvah, C. T. Djamégni, and Y. F. Yankam, "E2BaSeP: Efficient bayes based security protocol against ARP spoofing attacks in SDN architectures," *J. Hardw. Syst. Secur.*, vol. 5, no. 1, pp. 58–74, 2021.

[84] S. Hijazi and M. S. Obaidat, "Address resolution protocol spoofing attacks and security approaches: A survey," *Secur. Privacy*, vol. 2, no. 1, 2019, Art. no. e49.

[85] C. Zhang *et al.* "Towards a SDN-based integrated architecture for mitigating IP spoofing attack," *IEEE Access*, vol. 6, pp. 22 764–22 777, 2018.

[86] A. Rengarajan, R. Sugumar, and C. Jayakumar, "Secure verification technique for defending IP spoofing attacks," *Int. Arab J. Inf. Technol.*, vol. 13, no. 2, pp. 302–309, 2016.

[87] J. Xie, A. P. S. Meliopoulos, and G. J. Cokkinides, "PMU-based line differential protection under GPS spoofing attack," in *Proc. 54th Hawaii Int. Conf. Syst. Sci.*, 2021, pp. 1–9.

[88] J. Xie and A. P. S. Meliopoulos, "Sensitive detection of GPS spoofing attack in phasor measurement units via quasi-dynamic state estimation," *Computer*, vol. 53, no. 5, pp. 63–72, 2020.

[89] Smartthings: How it works. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/getting-started/groovy-for-smartthings.html?#how-it-works

[90] Capabilities reference. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/capabilities-reference.html

[91] The Lock Capacity. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/capabilities-reference.html#lock

[92] API sendevent() of device handler. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/device-handler-ref.html#sendevent

[93] API parse(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/device-handler-ref.html#parse

[94] API CreateEvent(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/device-handler-ref.html#createevent

[95] API subscribe(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=subscribe#subscribe

[96] API getid(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/device-ref.html#getid

[97] Subscribe to location events. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/smartapp-developers-guide/simple-event-handler-smartapps.html?highlight=location#subscribe-to-location-events

[98] Modes. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/smartapp-developers-guide/modes.html

[99] Invite a member to smartthings. Accessed: Jun. 2019. [Online]. Available: https://support.smartthings.com/hc/en-us/articles/115002085066-How-can-I-invite-members-in-the-SmartThings-app-

[100] Event. Accessed: Aug. 2020. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/event-ref.html?highlight=event#event

[101] API sendnotificationevent(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=sendnotificationevent#sendnotificationevent

[102] Device. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/device-handler-ref.html#device

[103] Parse events and attributes. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/device-type-developers-guide/parse.html#parse-events-and-attributes

[104] Building the service manager. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/cloud-and-lan-connected-device-types-developers-guide/building-cloud-connected-device-types/building-the-service-manager.html

[105] How to connect LIFX light bulbs. Accessed: Jun. 2019. [Online]. Available: https://support.smartthings.com/hc/en-us/articles/205956530-How-to-connect-LIFX-light-bulbs

[106] API sendevent() of smartapp. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=sendevent#sendevent

[107] A universally unique identifier (UUID) URN namespace. Accessed: Jun. 2019. [Online]. Available: https://tools.ietf.org/html/rfc4122

[108] Uses of class Java.Util.UUID. Accessed: Jun. 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/class-use/UUID.html

[109] Storing data with state. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/smartapp-developers-guide/state.html?highlight=storage

[110] A. A. Elsaeidy, N. Jagannath, A. G. Sanchis, A. Jamalipour, and K. S. Munasinghe, "Replay attack detection in smart cities using deep learning," *IEEE Access*, vol. 8, pp. 137 825–137 837, 2020.

[111] S.-H. Yoon, M.-S. Koh, J.-H. Park, and H.-J. Yu, "A new replay attack against automatic speaker verification systems," *IEEE Access*, vol. 8, pp. 36 080–36 088, 2020.

[112] SmartThings: What happens when the internet to the Hub goes out?. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/sept-2015-faq.html?highlight=offline#samsung-smartthings-hub-faq

[113] Service manager design pattern. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/cloud-and-lan-connected-device-types-developers-guide/understanding-the-service-manage-device-handler-design-pattern.html

[114] API addchilddevice() of smartapp. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=addChildDevice#add childdevice

[115] API getsunriseandsunset(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=getSunriseAndSunset#getsunriseandsunset

[116] API timeoffset(). Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/smartapp-ref.html?highlight=getSunriseAndSunset#timeoffset

[117] API documentation. Accessed: Jun. 2019. [Online]. Available: https://docs.smartthings.com/en/latest/ref-docs/reference.html

[118] Zigbee-accessory-dimmer. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/devicetypes/smartthings/zigbee-accessory-dimmer.src/zigbee-accessory-dimmer.groovy

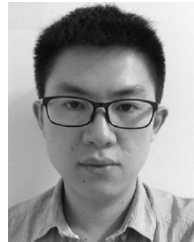
[119] Door knocker. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/imbrianj/door-knocker.src/door-knocker.groovy

[120] Unlock it when I arrive. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/unlock-it-when-i-arrive.src/unlock-it-when-i-arrive.groovy

[121] Photo burst when.... Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/photo-burst-when.src/photo-burst-when.groovy

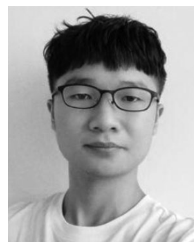
[122] The big switch. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/the-big-switch.src/the-big-switch.groovy

[123] Spruce-scheduler. Accessed: Jun. 2019. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/plaidsystems/spruce-scheduler.src/spruce-scheduler.groovy

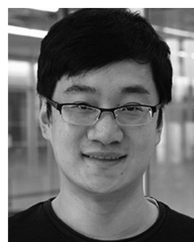


Bin Yuan (Member, IEEE) received the BS and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2013 and 2018, respectively. He is an associate professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include software-defined network security, network function virtualization, cloud security, privacy, and IoT security. He has published several technical papers in top conferences/journals, such as USENIX Security, CCS, the *IEEE Transactions on Services Computing*, *IEEE Transactions on Network and Service Management*, *IEEE Transactions on Network Science and Engineering*, *IEEE Internet of Things Journal*, Journal and *Future Generation Computer Systems*.

Yuhan Wu received the BS degree in software engineering from Hunan University, Changsha, China, in 2019. She is currently working toward the master's degree at the Huazhong University of Science and Technology (HUST), Wuhan, China. Her research interests include cloud security, privacy, and IoT security.



Maogen Yang received the BS degree in computer science and technology from Anhui Normal University, Wuhu, China, in 2020. He is currently working toward the master's degree at the Huazhong University of Science and Technology (HUST), Wuhan, China. His research interests include cloud security, privacy, and IoT security.



Luyi Xing (Member, IEEE) received the PhD degree from Indiana University Bloomington, Bloomington, Indiana, in 2017. He is currently an assistant professor of computer science with Indiana University Bloomington. He served on the Technical Program Committee of ACM CCS (2017–2020) and NDSS (2019–2021). He worked with Amazon.com, Inc. between 2015 and 2018. He published 17 papers at security conferences IEEE S&P, Usenix Security, ACM CCS, and NDSS. He published three papers at Black Hat. His research interests include security and privacy on IoT, mobile systems, and cloud platforms and services. His research broadly involves protocol design and analysis, program analysis, formal verification, machine learning, and NLP.

Xuchang Wang received the BS degree in information security from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2020. He is currently working toward the undergraduate degree at the Huazhong University of Science and Technology, Wuhan, China. His research interests include network security, cloud security, and IoT security.

Deqing Zou received the PhD degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2004. He is a professor of computer science with HUST. His main research interests include system security, trusted computing, virtualization, and cloud security. He has been the leader of one “863” project of China and three National Natural Science Foundation of China (NSFC) projects, and core member of several important national projects, such as National 973 Basic Research Program of China. He has applied almost 20 patents, published two books (one is entitled “Xen virtualization Technologies” and the other is entitled “Trusted Computing Technologies and Principles”) and more than 50 High-quality papers, including papers published by the *IEEE Transactions on Dependable and Secure Computing*, IEEE Symposium on Reliable Distributed Systems and so on. He has always served as a reviewer for several prestigious Journals, such as the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Cloud Computing*, and so on. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung scholars chair professor of computer science and engineering with HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published more than 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.