# ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms

Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang†, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, Atul Prakash

University of Michigan,
Shanghai JiaoTong Unviersity†
{jackjia, alfchen, rahmati, earlence, zmao, aprakash}@umich.edu,
{wangshiqi}@sjtu.edu.cn†

***Abstract*—The Internet-of-Things (IoT) has quickly evolved to a new appified era where third-party developers can write apps for IoT platforms using programming frameworks. Like other appified platforms, e.g., the smartphone platform, the permission system plays an important role in platform security. However, design flaws in current IoT platform permission models have been reported recently, exposing users to significant harm such as break-ins and theft. To solve these problems, a new access control model is needed for both current and future IoT platforms. In this paper, we propose ContexIoT, a context-based permission system for appified IoT platforms that provides contextual integrity by supporting fine-grained context identification for sensitive actions, and runtime prompts with rich context information to help users perform effective access control. Context definition in ContexIoT is at the inter-procedure control and data flow levels, that we show to be more comprehensive than previous context-based permission systems for the smartphone platform. ContexIoT is designed to be backward compatible and thus can be directly adopted by current IoT platforms.**

**We prototype ContexIoT on the Samsung SmartThings platform, with an automatic app patching mechanism developed to support unmodified commodity SmartThings apps. To evaluate the system's effectiveness, we perform the first extensive study of possible attacks on appified IoT platforms by reproducing reported IoT attacks and constructing new IoT attacks based on smartphone malware classes. We categorize these attacks based on lifecycle and adversary techniques, and build the first taxonomized IoT attack app dataset. Evaluating ContexIoT on this dataset, we find that it can effectively distinguish the attack context for all the tested apps. The performance evaluation on 283 commodity IoT apps shows that the app patching adds nearly negligible delay to the event triggering latency, and the permission request frequency is far below the threshold that is considered to risk user habituation or annoyance.**

## I. Introduction

The Internet-of-Things (IoT) has quickly evolved from its initial stage where sensors and actuators each provide hard-coded and disjoint functionality, to a new *appified* era, where programming frameworks are provided for third-party developers to build applications (apps) to manage a single or even a number of smart devices at the same time to realize more advanced and smarter control. Many such appified IoT platforms, for example Samsung SmartThings [13], Apple HomeKit [3], and Google Weave/Brillo [7], have already gained great popularity among home users today.

Like other appified platforms such as the smartphone platform, the permission model plays an important role in the security of these appified IoT platforms, defining an app's access to sensitive resources [19]. However, security-critical design flaws in the permission[1] model of these platforms, for example *overprivilege* problems due to the current coarse-grained permission definitions, have already been reported recently, exposing smart home users to significant harm such as break-ins and theft [35]. To solve these problems, a new access control model is needed in these appified IoT platforms in order to provide home users with more fine-grained control of app behavior.

Existing access control mechanisms employed by the most recent appified platform with huge popularity—the smartphone platform—have long been criticized to be coarse-grained, insufficient, and undemanding [19], [56], [64], [67], which are quite similar to the aforementioned problems in current IoT platforms. From numerous studies on Android and iOS permission systems, a key design flaw is that they either require users to make uninformed decision at install time [1] or prompt users at runtime when an app requests any of a handful of resources, without providing essential contextual information [2], [8]. These studies conclude that it is highly desirable to put the user in context when making permission granting decisions at runtime. This helps ensure a property known as "contextual integrity" defined by Nissenbaum [52] with which "information flows according to contextual norms," and it is advocated as the desired norm for future permission system design of the smartphone platform [64], [68], [19].

Taking lessons from previous permission systems, in this paper we aim to provide contextual integrity in appified IoT platforms in order to solve the security problems arising in current IoT platform permission systems. However, as discussed in previous attempts to support it in the smartphone platform [64], [19], providing contextual integrity in appified systems is challenging due to two reasons:

[1] SmartThings uses the term `capability` instead of `permission` [35]


NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
 ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23051

- **Availability of context** is not guaranteed in the lifecycle of the app: majority of sensitive permission requests occur when the user is not interacting with the requesting app [64]. This situation only gets worse in the IoT scenario, since unlike the UI-oriented smartphone apps, the whole point of developing IoT apps is to provide automated device control with minimum user involvement. Except sending notifications, usually no user interaction is required after the app setup procedure, making it more difficult to involve user in the context at runtime.
- **Frequency of prompts** is another important factor for a permission system to be effective [64]. On the smartphone platform, since the request frequencies for some permissions are too high to prompt the user each time a request occurs without risking user habituation or annoyance, current designs shift toward a model of only prompting the user the first time a request occurs to increase usability [64]. However, this harms contextual integrity since the subsequent sensitive actions may be performed in a completely different context than that of the initial request.

In view of these challenges, we design and implement *ContexIoT* (means putting IoT into context), a context-based permission system for appified IoT platforms which supports fine-grained identification of context for a sensitive action and runtime prompts with rich context information to help provide contextual integrity. In our design, the context is defined at inter-procedure control and data flow levels, and can be flexibly tuned to support different context granularity in order to best balance security and usability. ContexIoT is designed to be backward compatible, and thus can be directly adopted by current IoT platforms to provide more effective access control.

At a high level, ContexIoT design is based on the observation that a permission granted by the user is expected to allow the triggered app functionality *only under that particular usage context*. We abstract the usage context of an app functionality as a program path, and thus define the context as the execution flow of the code at runtime, including how the functionality is triggered and what data is flowing along the execution path. This definition falls into the trigger-action based programming model of IoT apps [62], so that when the user is prompted, the context can be naturally represented as the triggering sequence of real-world physical events. To help the user make a more informed decision, we use taint analysis to track the runtime data on the execution path and label the data source when presenting the context information to the user, e.g., showing whether the data to be sent out is the user password or just the battery level. We compare ContexIoT context definition with existing context-based security approaches for smartphone platforms, and find that our fine-grained definition at inter-procedural control and data flow levels can successfully identify stealthy attack paths that can evade other systems, showing better visibility than previous design.

We built a prototype of ContexIoT on the Samsung SmartThings platform, which at the time of writing has the largest number of supported device types and IoT apps (called SmartApps) among all the IoT platforms [35]. To support existing SmartApps without changing the closed-source SmartThings cloud backend, we developed an app patching mechanism that can convert unmodified commodity SmartApps to ContexIoT-compatible SmartApps. The patching process separates the execution flow of a sensitive action in the original SmartApp into two steps: (1) Collect the context information before the action is executed, and (2) Allow or deny the action based on the in-context user decision. ContexIoT uses a cloud backend to remember the previous decisions by maintaining a mapping between an in-context sensitive action for an app and the granting decision. If no mapping is found, the system prompts the user with the context and the requested action, and stores the user decision to the ContexIoT cloud backend.

To evaluate the effectiveness of our approach, we extensively collect the reported IoT attacks from multiple sources. For exploits on non-appified platforms, we explore the possibility of migrating them to appified IoT platforms. In total, we have constructed 10 SmartApps that are either malware or vulnerable apps based on the reported IoT attacks. Considering that appified IoT platforms are still in a primitive stage and not many attacks are reported, we further survey malware classes from appified smartphone platforms. We taxonomize them into 4 categories based on the malware lifecycle, with 3–6 species in each category. Out of the 17 species in total, we find that 15 of them can be naturally migrated to IoT platforms due to the similarity of appified platforms. Overall, we build an IoT attack app dataset with 25 SmartApps, each representing a unique attack class. Evaluating ContexIoT on this dataset, we find that all 25 different attack execution paths have been successfully distinguished with the context information correctness manually confirmed.

For performance evaluation, we build a dynamic testing framework based on the device simulator provided by the SmartThings IDE. Using this framework, we dynamically inject virtual device events and are able to trigger all the 916 event handling logic in 283 SmartApps. From the performance measurement results, we find that the SmartApp patching logic only introduces 67.1 ms additional delay on average, which is negligible in practice since the end-to-end latency is dominated by the network latency between the SmartThings cloud backend and the physical device. We also evaluate the frequency of prompts, and find that the average possible lifetime of permission request prompts is only 3.5 times for each SmartApp on average, which is far below the threshold that is considered to risk user habituation or annoyance [67], [64].

To summarize, our contributions are three-fold:

- To understand the design requirements for a context-based permission system on IoT platforms, we perform the first extensive study of possible attacks on appified IoT platforms by reproducing reported IoT attacks and constructing new IoT attacks based on smartphone malware classes. We categorize these attacks based on the lifecycle and adversary techniques, and build the first taxonomized IoT attack app dataset with 25 SmartApps, each representing a unique attack class.
- We design and implement ContexIoT, a context-based permission system for appified IoT platforms that supports fine-grained context identification and rich context information prompting at runtime to help pro-

vide contextual integrity. To distinguish fine-grained context, ContexIoT defines context as execution paths at inter-procedure control and data flow levels, which is shown to be more comprehensive than previous designs for smartphone platforms. To help users make more informed decisions, ContexIoT also labels the data source of the runtime data using taint analysis. To provide backward compatibility, ContexIoT contributes an app-patching mechanism that converts existing IoT apps to ContexIoT-compatible apps.

- We prototype ContexIoT on the Samsung SmartThings platform, and evaluate it on our IoT attack app dataset for system effectiveness with over 283 existing SmartApps for system performance. For the attack app dataset, we find that all attack execution paths have been successfully distinguished with correct context information annotated. The performance evaluation results indicate that ContexIoT app patching adds nearly negligible delay, and the permission request frequency is far below the threshold that is considered to risk user habituation or annoyance.

## II. Related Work and Background

In this section, we cover previous work on permission-based access access control and IoT security, and necessary background for Samsung SmartThings platform.

### A. Related Work

*1) Permission-based Access Control:* The permission-based access control plays an important role in the security of appified platforms, and has received a lot of attention by the security research community [33], [21], [50], [67]. Acar *et al.* pointed out that the current concept of permission granting mechanism has failed in practice, and proposed a clean break to seek for permission revolutions [19]. Backes *et al.* advocates *contextual integrity* as the desired norm for future permission systems design based on a rigorous user study on Android platform [64]. Roesner *et al.* introduced User-Driven Access Control where the user is kept involved with access control decisions in case-by-case basis by using access control gadgers [56]. Rahmati *et al.* introduced the concept of context-specific access control [54] in Android where app Activities are used to distinguish different user contexts. Compared to these previous systems for the smartphone platforms, this paper aims to provide contextual integrity to the IoT platforms, which faces several IoT specific challenges, e.g., it is more difficult to involve users in the context. Also, the context in ContexIoT is defined at both control and data flow levels, which is more fine-grained. We compare our context definition with those in these previous work later in §V, and it shows that our definition is more comprehensive and can defeat attacks that can evade these previous design.

Another line of research focuses on improving the usability of the permission system. Felt *et al.* introduced a set of guidelines on when and how to request permissions [34], which can instruct the design of default security policies. Wijesekera *et al.* suggested the systems to learn about their users' privacy preferences and only confront users with consent dialogs when a permission request is unexpected for the user [64]. Keley *et al.* proposed to enrich permission dialogs with more detailed privacy-related information to help users make more effective decision [44]. In comparison, ContexIoT targets an orthogonal goal, i.e., enabling effective identification of fine-grained context for security sensitive actions. Leveraging the rich context information collected in ContexIoT, these approaches can be combined with ContexIoT to improve usability.

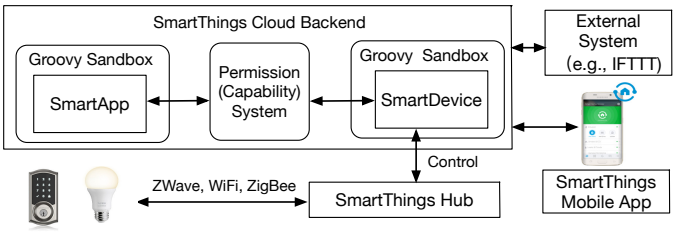



Fig. 1: SmartThings architecture overview

*2) IoT Security:* The IoT security research is centered around three themes: Devices, Protocols and Platforms. In the IoT device scope, many Telnet-capable IoT devices are reported to be vulnerable due to weak/default password or unprotected debugging interfaces [69]. Ur *et al.* identified problems in the access control of the Philips Hue lighting system and the Kwikset door lock that fails to enable essential use cases [61]. Ronen *et al.* demonstrated extended functionality attacks on smart lights that can leak information and causing seizures using strobed light [57].

On the protocol level, researchers demonstrated flaws in the ZigBee and ZWave protocol implementations of IoT devices [37]. More recently, the misusing of some protocols in some IoT specific scenarios has been reported to cause security and safety problems [40]. For example, using the BLE (Bluetooth Low Energy) range as the proof to verify physical proximity is considered insecure in the auto-unlock usage scenario. In our work, we extensively survey these IoT attacks, and explore the feasibility of migrating them to the appified IoT platform.

On the IoT platform level, recent work discovered a series of security-critical design flaws such as the coarse-grained permission definition on the SmartThings platform [35]. To limit the usage of sensitive data, Fernandes *et al.* proposed the FlowFence framework [36] that supports flow policy rules for IoT apps. Our work is similarly motivated by the security problems in the appified IoT platforms. However, unlike FlowFence, our approach does not require additional developer effort and is backward compatible. Moreover, ContexIoT allows user control in cases where a particular data flow might be allowed in one scenario, but should be blocked in another.

### B. Background

In this paper, we focus on the Samsung SmartThings platform, which uses a popular cloud-backed architecture design as shown in Figure 1. Other popular IoT platforms such as Apple's HomeKit and Google's Weave/Brillo also use such design, and the differences only lie in the communication protocols used in the wireless hop. As shown later in §V, ContexIoT also leverages such cloud-backed architecture, and

thus is generally applicable to these popular IoT platforms today.

As shown in Figure 1, the SmartThings ecosystem consists of three major components: a hub, a SmartThings cloud backend, and a smartphone Companion App. The IoT apps in the SmartThings platform are called *SmartApps*, which are written in Groovy using the Web based IDE provided by SmartThings. These SmartApps are not running on the IoT devices. Instead, they are executed by the SmartThings cloud platform within a sandboxed environment. The sandbox is an implementation of a Groovy source code transformation that only allows whitelisted method calls to succeed in the SmartApp, and thus disables some object-oriented language features in Groovy such as creating classes. The SmartApp can choose to expose web service endpoints to respond to HTTP requests from external application, which is protected by OAuth-based authentication. Note that SmartApps support dynamic method invocation (using the `GString` feature), and thus similar to the reflection feature in some programming languages such as Java, a method can be invoked by providing its name as a string parameter. In later sections, we detail the security problems caused by this dynamic feature and how ContexIoT addresses it.

The cloud backend also runs the *SmartDevices*, which are software wrappers for physical devices in the user's home. A SmartApp and a SmartDevice communicate in two ways (1) The SmartApp invokes operations on the SmartDevices via method call (e.g., to lock the door), (2) The SmartApp subscribes to events that the SmartDevices generates (e.g., smoke detected). The communication between a SmartApp and the functionality of a SmartDevice are controlled by the permission model, which is called the *capability* system of the SmartThings platform.

The current capability model of the SmartThings platform only provides coarse-grained binding between SmartApps and SmartDevices. Capability defines a set of commands and attributes that devices can support, and SmartApps state the capabilities they need. Based on that, users bind SmartDevices to SmartApps at the app installation time. Recent work has uncovered several security problems with the permission/capability system of the SmartThings platform such as *overprivilege* [35]. In this paper, we design and implement ContexIoT, a context-based permission system for appified IoT platforms to address these problems.

## III. Threat Model and Problem Scope

**Threat model.** In this paper, we consider app-level IoT attacks on the appified IoT platforms which attempt to access IoT users' sensitive data or execute privileged functionality. The attacker can launch the attack through either (1) *malware*, in which the malicious logic is embedded at the IoT app install time, or (2) *vulnerable apps*, which contain design or implementation flaws that can be exploited by a co-located malicious IoT app or a remote network attacker to escalate its privilege and cause damages such as unauthorized device control and sensitive data theft. In this paper, we assume the platform itself to be trustworthy and uncompromised, thus some recent IoT attacks exploting the unprocted management interfaces of IoT devices to compromise the hardware (e.g., Mirai attack[9]) are not in our scope; Securing the platform by reducing its attack surface is orthogonal to our research (e.g., [22]).

**Goal and problem scope.** The goal of ContexIoT is to raise the bar for the aforementioned app-level attacks by providing context integrity support. To achieve the goal, ContexIoT aims to enable a user to validate two important properties when a sensitive action is triggered at runtime: (1) *When*: whether the sensitive action is triggered at the user-desired conditions, and (2) *What*: whether the sensitive action matches the user-intended action. Runtime data content validation and protection are also in our scope, since to perform effective access control, the user needs to understand what the data is being accessed or about to be sent.

Since we target app-level attacks, attacks not exploiting app-level vulnerabilities are out of our scope. For example, attacks using stolen external service security tokens due to the weak protection of these external services [35] are considered as a separate problem, and should be taken care of by the provider of each service integrated with SmartThings. Also, the denial-of-service (DoS) behavior of "ignoring the functionality" [57] is not in our scope. For instance, a malicious break-in alert app that claims to notify the user when it detects a break-in may ignore the event instead of sending alerts. In this paper, we target attacks with explicit code-level malicious logic, which can cause more severe damage such as privilege escalation and sensitive data theft compared to DoS.

## IV. Attack Taxonomy

To better understand the security and privacy issues associated with the current appified IoT platforms, we performed an extensive survey of attacks reported on both IoT devices and the smartphone platforms, and studied the feasibility of their migration to the SmartThings platform. For all attacks that are applicable, we constructed misbehaving SmartApps that achieve similar malicious functionality to guide our design and evaluate the effectiveness of our system.

### A. Reported IoT Attacks

Similar to the early stages of any emerging technology, the priority of most vendors are functionalities and faster time-to-market of their products, while security and privacy have not received much attention. The security of IoT platforms are not hypothetical concerns as a number of real attacks have already been reported. For example, IoT devices being compromised to use as bots to launch DDoS attack [69], the misusing of BLE range to confirm physical proximity are leveraged by attacker to unlock your vehicle and door [40], [38]. Table I lists the reported attack instances we collected from sources including both academic papers and news articles. We categorize them into three classes based on the problem area.

*1) Vulnerable Authentication:* Authentication plays an important role in the whole lifecycle of IoT devices, and vulnerable authentication is spotted in many critical procedures of IoT devices. For example, a vulnerable device-pairing mechanism may allow attacker to take full control of the device. Due to the lack of displaying functionality in many IoT devices, a management console is typically provided using protocols such as Telnet, HTTP and SSH, which can suffer from problems

TABLE I: A taxonomy of reported IoT attacks and their applicability to the SmartThings platform

| Problem area | Attack description | Platform | Attack vectors | References | Applicable to ST? |
|---|---|---|---|---|---|
| Vulnerable authentication | Backdoor pin code injection | SmartThings | Stealing OAuth tokens; Inject command into Web Service SmartApp | [35] | ✓ |
| | Get remote shell of device | Telnet-capable IoT devices | Weak/default password; Credential included in the image; Unprotected debugging interface | [53], [69], [12] | N/A |
| | Leaking information / creating seizures using strobed light | Smart connected LEDs | Unsecured device pairing procedure | [57] | ✓ |
| | Impersonate device to steal data | Bonjour-supported IoT devices | Unable to handle name collision in the local network | [24] | N/A |
| Malicious app/firmware | Door lock pin code snooping | SmartThings | Overprivilege due to the SmartApp-SmartDevice coarse-binding | [35] | ✓ |
| | Disabling vacation mode | SmartThings | Misusing logic of a benign SmartApp to do event spoofing | [35] | ✓ |
| | Fake alarm | SmartThings | Controlling device without gaining appropriate capability | [35] | ✓ |
| | Surreptitious surveillance | Sony surveillance camera | Installed with malware in the device retailing process | [17] | ✓ |
| | Spyware | Barcode scanner | Preloaded with malicious firmware | [5] | ✓ |
| Problematic usage scenario | Undesired unlocking | BLE Smart locks | Misusing BLE range to confirm the physical proximity of user | [40] | ✓ |
| | BLE relay unlocking | BLE Smart locks | Misusing BLE range to confirm physical proximity of user; BLE Replay attack | [40], [38] | ✓ |
| | Lock access revocation / logging evasion | DGC lock | Failing to ensure state consistency between device and server | [40] | ✓ |

such as weak or default password. We find that beside gaining a remote shell on the devices, many attacks can be easily implemented as malicious SmartApps and distributed in the platform. For example, a malicious smart light control app can perform similar malicious activities as described in [57] to use luminance as side channel to inform thieves near the house that the owner is not at home, or creating seizures using strobed lights.

*2) Malicious App/Firmware:* Even before the emerging of appified IoT platforms, malicious preloaded application or firmware have already been reported [17], [5]. Functionalities of these malicious app/firmware can be easily migrated to SmartThings platform, as it opens a broad range of device capabilities to 3rd party developers. For example, one of our constructed malicious SmartApp show how an attacker can surreptitiously spy on the daily life of house owner if the user installed the malware disguised as normal surveillance camera app. In addition, some attacks that has already been reported as feasible on SmartThings platform [35], such as snooping the door lock pin code, are also included, and used to guide our system design.

*3) Problematic Usage Scenario:* Another category of IoT attacks exploits the misusing of technology in some IoT specific usage scenarios. For example, using the presence of user's device in the BLE range as indicator of user's presence at the door is considered problematic, since it may undesirably unlock all the doors of the house due to the long range of BLE. We found that such problems can also be reproduced in SmartThings using the capability granted to the SmartApps.

### B. Migrated from The Smartphone Platforms

Security requirements of appified IoT platforms and the smartphone platforms share many similarities, including the definition of access to resources, and privilege separation. We surveyed the mobile malware ecosystem, and categorized them based on the different techniques they used in 4 aspects of their lifecycle below. We discussed the possibility of each malware classes to be appified on SmartThings platforms, and construct real malicious SmartApps for demonstration and evaluation purposes if applicable.

*1) Installation.:* The most common technique seen in mobile malware samples to distribute themselves is to repackage their malicious app logic into commodity apps that claim normal functionality. This attack venue is clearly applicable to IoT malware. Moreover, we find that the app update procedure, which is reported to have been leveraged by mobile malware to carry out their malicious payload is also vulnerable in the SmartThings platform. SmartThings makes it very convenient for SmartApp developers to deploy their updates, by automatically updating the cloud instances of the SmartApp for all the user. In this mode, the attacker can disguise the malicious logic of their apps by not piggybacking the entire malicious payload into the original app, but slowly introducing it through future updates. In addition, drive-by download can also be easily adopted by attacker to entice users to download the malware app.

*2) Activation:* Malicious logic can potentially be triggered by various events. We categorize these events into three categories: (1) *Remote command* (e.g., incoming SMS), (2) *User events* (e.g., user click), and (3) *System events*. The trigger-action programming model of IoT provides similar flexibility for attacker to embed their malicious app logic into any of the three types of events. Specifically, some IoT events are very informative and may leak sensitive data to untrusted apps that don't have essential capability. For instance, the mode change (home/away/night) events are broadcasted system-wide and an malicious app that doesn't have the access to any sensing devices can know when the house owner is leaving by receiving the broadcast, and facilitating potential break-in.

*3) Adversary Technique:* Two basic principles that guide the design of malware are to (1) carry out the malicious payload as fully as possible under the system constraints

TABLE II: A taxonomy of smartphone malware classes and their applicability to the SmartThings platform

| | Category and descriptions | References | Applicable to ST? |
|---|---|---|---|
| Installation | **Repackaging**: Malicious logic are enclosed into high-profile apps to trick user to download | [27], [74], [26], [42] | ✓ |
| | **App update**: Malicious payloads are downloaded during the app update process for disguising purpose | [66], [74] | ✓ |
| | **Drive-by Download**: Enticing user to download the "interesting" or "feature-rich" apps | [74] | ✓ |
| Activation | **Remote command**: Attacker controlled remote input, e.g., incoming SMS | [74], [39] | ✓ |
| | **User events**: Event triggered by the user, e.g., button click | [39] | ✓ |
| | **System events**: Event generated by the system, e.g., boot complete event | [74], [46] | ✓ |
| Adversary technique | **Abusing permission**: malicious app logic abuses the privilege granted to the app | [39], [31], [51] | ✓ |
| | **Exploiting weakness of general system design**: generic system mechanisms such as IPC | [63], [23] | ✓ |
| | **Exploiting weakness of platform specific features**: techniques specific to platform, e.g., native code | [19], [20], [49], [47] | ✓ |
| | **Exploiting system vulnerability**: security flaws and bugs in the system e.g., root exploits | [59], [71], [43], [65], [18] | N/A |
| | **Shadow payload**: disguise malicious payload using obfuscation or encryption techniques | [74], [55] | ✓ |
| | **Side channel**: carry out malicious payload using covert channel | [32], [70], [72], [29] | ✓ |
| Malicious payload | **Remote control**: Taking control of user's device with C&C servers | [74], [46] | ✓ |
| | **Spyware**: Aiming to gather information from the victims without their knowledge | [39], [31], [51], [72], [48] | ✓ |
| | **Adware**: Downloading and displaying unwanted ads on the user's device | [58], [46], [42] | ✓ |
| | **Ransomware**: Installed covertly to DoS the device and demands a ransom payment to restore it | [45], [43] | ✓ |
| | **Privilege escalation**: Exploiting a bug or design flaw of the system to gain elevated access | [59], [65], [73], [47] | N/A |

to achieve maximum benefit; (2) evade detection to prolong their life-time. Guided by these principles various adversary techniques are used that we categorize into 6 classes shown in Table II. Except exploiting system vulnerability, such as root exploits which is orthogonal to our research, techniques in all other 5 categories can be applied to the appified IoT platforms. For example, the permission mechanism of commodity platforms offers "all or nothing", meaning that once the permission is granted, the privilege can be used for any purpose. This allows malicious app logic to abuse the trusted granted to the declared benign functionality of the same app. Such *overprivilege* are common in SmartThings platform where a AutoLock SmartApp also has the capability to unlock the door anytime [35]. Another interesting evasion technique is to use IPC between malicious apps to carry out malicious payload. and we demonstrate on SmartThings that even IPC is not supported by the platform, malicious SmartApps with least privilege can collaborate to leak sensitive data such as door lock pin code through the device status as side-channel [35]. In addition, weaknesses in platform specific features can also be leveraged by IoT malware. For example, the `GString` support of the Groovy language enables attacker to modify the control flow of the app at runtime, which can be used to evade all static analysis based malware detection systems.

*4) Malicious Payload.:* Existing smartphone malware can be largely characterized by their carried payloads. We partition these payloads into five different categories: *remote control*,*spyware*,*adware*,*ransomware*, and *privilege escalation*. Among them, privilege escalation leverages the vulnerabilities of the system, and is out of the scope of our work. Remote control and spyware are two common types of payload on the smartphone platforms and can be easily adopted by IoT malware. Adware is a type of app that downloads and display unwanted ads to the user. There are many channels including push notification and SMS in IoT platforms that can be leveraged by adware to spread ads. Ransomware is an emerging threat to modern systems, and we demonstrate examples showing that IoT malware can also demand ransom payment in situations where the effect caused by the ransomware cannot be easily reverted (e.g., when the user is on vacation).

Among all the 29 categories of attacks shown in Table I and II, 25 of them are in the scope of our research and are the attacks that our proposed system is designed to defeat. Using these 25 categories, we implemented 25 malicious apps corresponding to each of the category on SmartThings platform, and evaluated our system against them in §VII. We provide all of malware samples developed in this project on our website [11] to benefit future research. Below, we will provide more detail about three instances of the proof-of-concept attacks we have implemented. We will refer to these three attacks in later sections to show how our design and implementation choices defeat these attacks.

**Surveillance disabling attack** (Listing 1) repackages its malicious payload in an home monitoring app, and abuse the switch control capability granted to this app to turn off the surveillance camera when it detects that the owner has left to facilitate potential break-in. It also leverages the vulnerability in the event system of SmartThings to subscribe on the mode change events without explicitly requiring any capability.

**Pin code snooping attack** (Listing 2) uses a battery monitor SmartApp to disguise its malicious intent at the source code level, and is first proposed in the recent work [35]. The app subscribes on the battery report of the lock, and sends the battery data to remote client for visualization purpose. However, it won't reveal its malicious payload until the victim sets up a new pin code. Due to the *overprivilege* issue of the SmartThings platform, the app subscribing on the battery report can also receive the `codeReport` event when pin code is updated, and user can distinguish the benign and malicious behaviors only based on the runtime value.

**Remote control attack** (Listing 3) leverages the Groovy dynamic method invocation and the asynchronous execution flow to disguise its malicious payload. It pulls the attack server everyday for new malicious command and stores them in the global variables shared by all event handlers. A separated process that is scheduled to run every 5 minutes invokes the malicious command stored in the global variables using `GString`, which allows attacker to potentially control all of the devices associated with this app.

Listing 1: Code snippet of surveillance disabling attack

```
input "switch", "capability.switch", title:
    "The switch your camera is controlled by"
// subscribe the mode change event
subscribe(location,"mode",handler)

def handler(evt){
//turn switch off if the owner has left
   if(evt.value == "Away"){
      switch.off()
   }
}
```

Listing 2: Code snippet of pin code snooping attack

```
input "lock","capability.battery", title:
    "The device you want to have its battery
    monitored"
// subscribe the battery report from the lock
subscribe(lock,"battery",handler)

def handler(evt){
//transmit battery data to graphing webservice
   httpPost (url, evt.jsonValue)
}
```

Listing 3: Code snippet of remote control attack

```
//Subscribe on the sunset event
subscribe(location,"sunset",dispatcher)
//Schedule the handler to be executed every 5
    minutes
schedule("0 5 * * * ?", handler)

def dispatcher(){
   httpGet(url){
//Query attack server for command and store
    them in global variables
      resp->
         state.method = resp.data['method']
         state.flag = true
   }
}
def handler(){
//Execute the command if it's updated
   if(state.flag == true){
      "$state.method"()
      state.flag = false
   }
}
```

## V. ContexIoT Design

Guided by the attack survey and taxonomy, we present the context definition in ContexIoT by identifying a set of information that is essential to distinguish the attack and benign logic in an app at runtime. To better clarify our context definition, we perform a comparison between ContexIoT and previous context-based approaches that aim at detecting malicious app logic or enforcing policies.

### A. Context Definition

We use the term *sink* to refer to all the security sensitive actions of the app in later sections. As shown in Table III, we extract the context definitions from a list of representative related work and categorize them into 5 classes:

**UID/GID.** For app-level access control mechanisms, the context used to make permission granting decision is the identify of the app, i.e., the UID/GID from the system perspective. Mandatory Access Control (MAC) and Discretionary Access Control (DAC) systems on the smartphone platforms are several examples that use this context definition [25], [60], [31], but they are not able to distinguish attack and benign app logic within the same app.

**UI Activity.** Runtime access control systems on mobile platforms put user in the context of an app's UI activity to make permission granting decisions [56], [67], [33], [31]. The problem of using UI indicators alone is that it cannot restrict how the app uses the sensitive data. In addition, since the UI is generally not available in the IoT apps due to the design goal of minimum user involvement, it cannot be integrated into the context definition of permission systems on the IoT platforms.

**Control flow.** The events that trigger the execution of the payload, and the conditional statements (e.g., environmental attributes controlling the execution of payloads) consist of the control flow data in the definition of context. The control flow context of a sink is useful to distinguish attack and benign execution paths. For example, a door `unlock()` action triggered by a remote command is more suspicious than that triggered by entering the correct pin code. However, not all malicious behaviors can be distinguished using control flow context alone, the data floating on the execution paths at runtime is also necessary for making proper permission granting decisions in certain scenarios.

**Runtime value.** In our attack survey, we find that the same control-flow path can be used either for benign purpose or carrying out attack payload depending on the runtime value of the variables that are related to the security sensitive behavior of the app. As shown in the pin code snooping attack (Listing 2), the control flow paths of receiving the battery report and pin code update report are the same, and it depends on the runtime value to distinguish them. However, using runtime value to check contextual integrity causes usability problems since even a tiny change in the data results in different context, and user can be overwhelmed with the large number of decisions to make. Moreover, presenting raw runtime data may not necessarily inform the user about the whether the data is sensitive or not, especially when the malware uses shadow payload technique shown in Table II to conceal the content.

**Data flow.** The data dependency information in the data flow context is critical to communicate the context to the user. Integrating it into the context definition mitigates the problems mentioned above by (1) reducing the number of different context by merging the runtime data that come from the same data origin, and (2) tagging the data dependency information to the runtime values to help user make more informed permission granting decision based on the property of data being sent out.

As shown, besides the *UI activity* that is generally not available for IoT apps, ContexIoT integrates all the other context components and thus has the most comprehensive definition of context among related work. Later in §V-C, we further use evasion attack discussion to show how this

TABLE III: Comparison of the context definitions among related work

| Name | Description | Definition of context | | | | | Decision made in context? |
|---|---|---|---|---|---|---|---|
| | | Uid/Gid | UI Activity | Control flow | Runtime value | Data flow | |
| ACG [56] | User-driven access control | ✓ | ✓ | | | | ✓ |
| AppContext* [68] | Static context-based analysis for malware detection | ✓ | | ✓ | | ✓ | - |
| AppFence [41] | Protecting private data from being exfiltrated | ✓ | | | | ✓ | |
| Aurasium [67] | Repackaging app to attach policy enforcement code | ✓ | ✓ | | ✓ | | ✓ |
| CRePE [30] | Enforcing context-based fine-grained policy | ✓ | | | ✓ | | |
| FlaskDroid [25] | Fine-grained MAC on middleware and kernel layer | ✓ | | | ✓ | | |
| SEAndroid [60] | Flexible MAC for Android apps | ✓ | | | | | |
| SEACAT [31] | Integrating both MAC and DAC in the policy checks | ✓ | ✓ | | ✓ | | ✓ |
| TaintDroid [33] | Dynamic taint tracking and analysis system | ✓ | ✓ | | ✓ | ✓ | ✓ |
| TriggerScope* [39] | Static trigger-based analysis for malware detection | ✓ | | ✓ | | ✓ | - |
| ***ContexIoT*** | Providing contextual integrity to permission granting | ✓ | - | ✓ | ✓ | ✓ | ✓ |

* These work focus on detecting malicious behavior with static analysis, but not enforcing access control at runtime. However, their methodologies of distinguishing benign and malicious behavior are based on their definitions of context.

comprehensive definition can help improve the permission system effectiveness.

### B. ContexIoT Approach

We next present ContexIoT, our approach that provide contextual integrity to the permission granting process of IoT apps using the context definition defined in §V-A. As shown in Figure 2, the general design of ContexIoT consists of two major steps: (1) At the app installation time, ContexIoT patches the app with security-focused logic to collect essential context and separate the execution flow of the security sensitive behaviors into asynchronous procedures: first request the permission in the current context, and then perform the action when receiving permission granting response. (2) At runtime, the cloud-backed permission management service handles the request from the patched apps and prompts to the user with the context if necessary. Figure 2 shows an example in which a malicious home temperature control app is granted with the capability to control the window based on the temperature. However, the attack logic embedded in the app code covertly opens the window when it detects the mode of the home is changed to `Sleep`, which allows the attacker to break in. The ContexIoT patched app puts the open window execution on hold and sends the collected context information to the backend. If previous decision for this context is not found in the backend, the permission service prompts user and takes the permission granting decision that is made in context. The permission service learns the security preference under this context of the user to prevent unnecessary prompts in the future. We introduce how ContexIoT approach collects and uses the context as follows.

**Context collection.** To overcome the black-box nature of the cloud-backed IoT platform, ContexIoT patches the context collection logic to the app code, allowing the patched apps to gather essential information of their own running context without requiring system access. However, precisely tracking all the control and data flow attributes of the app requires adding the logging logic to almost every instruction in the app code, which may at least double the computation overhead. To address this challenge, ContexIoT takes an hybrid approach combining static analysis and runtime logging to collect essential context efficiently – using static analysis results to reduce the overhead of runtime logging.

The static analysis first identifies all the potential sinks, which are those secure sensitive behaviors in the app code, and constructs an Inter-procedural Control Flow Graph (ICFG) from the program entry points to the sinks. App code that is not on any control-flow path from the entry points to the sinks does not need to be patched with the runtime logging logic since it won't affect the behavior of the sensitive action. However, some exceptions need to be made for the app logic that may implicitly affect the sinks, which are detailed in §VI. In addition, some context information that are deterministic statically that doesn't depend on runtime values are precomputed by the static analysis and annotated on the statements of the app code to further reduce runtime computation overhead.

ContexIoT then efficiently patches the app with the context collection logic. The general approach is to maintain an environment variable to store the context information for each application variable that are labeled as related to the program sink by the static analysis. The environment variables are automatically updated during the execution of the app based on the logic implemented by ContexIoT. And when the sensitive execution is triggered at runtime, a context collection function gathers the essential information from the environment of all the variables along the execution path. And the context information is sent to the backend to request permission for executing the sensitive action.

**Context usage.** Among the different types of information in our context definition, the control flow information, which describes the triggering action of the sensitive action, together with the runtime data should be able to distinguish the context of attack execution path and benign execution path. And the data flow information is used to communicate the context to user to better inform the user the security implications.

As shown in Figure 2, the backend permission service maintains an authorized permission-context mapping table for each user. Every time when a ContexIoT patched app attempts to perform a security sensitive action, a permission request containing the context information is sent to the backend. The cloud-based permission service checks whether the context has been previously allowed or denied. If not, it prompts user with a dialog presenting the permission request and the associated context and adds an additional entry to the mapping table to store the user's decision as security preferences. The context structure contains the 4 out of 5 context components

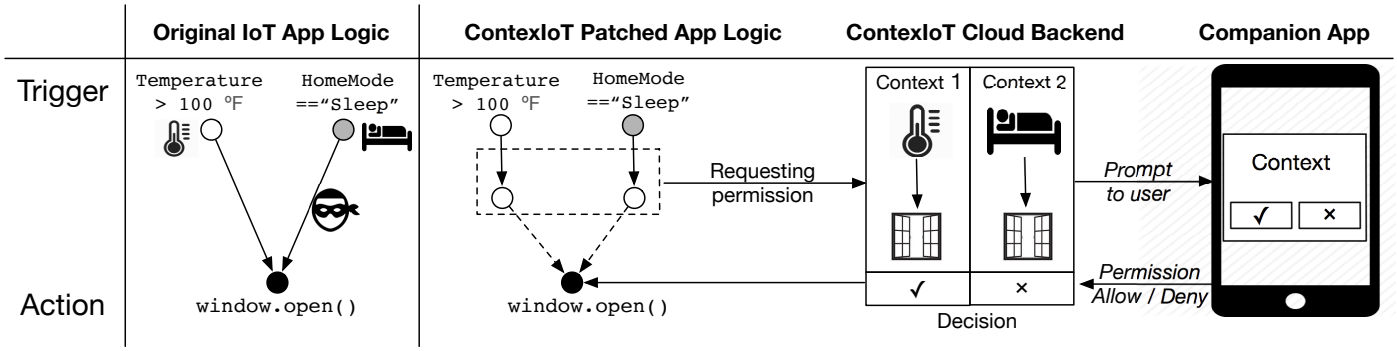



Fig. 2: ContexIoT overview with a concrete example showing our context-based access control

TABLE IV: Evasion attacks on context-based security approaches

| Name | Evasion attacks | | | |
|---|---|---|---|---|
| | Asynchronous leakage | Control flow abuse | Dynamic code loading | Policy abuse |
| ACG | × | | | |
| AppContext* | | | × | |
| AppFence | × | | | |
| Aurasium | | × | | |
| CRePE | × | | | |
| FlaskDroid | | | | × |
| SEAndroid | | | | × |
| SEACAT | | | | × |
| TaintDroid | | × | | |
| TriggerScope* | | | × | |

* These work focus on detecting malicious behavior with static analysis using context, and are thus vulnerable to dynamic code loading.

described earlier in §V-A. Our definition cannot include the UI Activity class since it is not available in IoT apps. In the implementation, some optimization can be applied to reduce the frequency of prompts by merging some components, which is detailed later in §VI.

### C. Comparison with Other Context-based Security Approaches

Since our context definition contains the complete inter-procedure control and data flow information, it can distinguish any attack logic in the app code level. To show the effectiveness of this design, we compare our context definition with other context-based security approaches proposed by previous work for smartphone platforms. In Table IV, we list a set of evasion attacks that can bypass those systems but can be defeated by ContexIoT. These evasion attacks fall into 4 categories as follows.

**Asynchronous leakage.** Sensitive data can be leaked to remote attacker stealthily, where the accessing and the transferring of the data are executed asynchronously, using global variables or other sources to share the data between the two procedures. The remote control attack shown in Listing 3 is one such example. Access control mechanisms without information flow tracking support [56], [41], [30] can be evaded by malware that abuses the granted access to resources and covertly leak them to the attacker. ContexIoT defeats such attack by integrating data dependency into the context definition. For the remote control attack, when the malicious payload is executed, ContexIoT presents users with the data dependency information for the sensitive action, which explicitly tells user that the method about to be executed comes from the response received in a separate procedure.

**Control flow abuse.** Access control systems that enforce policies only at the granularity of sinks without tracking how the sink is triggered [67], [33] is vulnerable to malware that abuses control flow to carry out malicious payload. For instance, a malicious lock manager app is granted with the `unlock()` capability by such sink-based access control systems the first time it attempts to unlock the door when it detects the house owner is back. However it can reuse the same code snippet that has already been permitted to unlock the door upon the remote attacker's request. In the context design of ContexIoT, the inconsistency of the control flows in these two scenarios are then detected, and the user's permission are required separately.

**Dynamic code loading.** Static analysis based malware detection approaches [39], [68] can be evaded by dynamic code loading. The `GString` support of SmartApps allows malicious payload to be revealed only at runtime. ContexIoT statically detects potential sinks for dynamic code loading, thus prevents malicious logic from evading the access control.

**Policy abuse.** MAC and DAC based approaches [25], [60], [31] grant the access at the application level based on user or system defined policies. And they are intrinsically vulnerable to malicious app logic that abuses the trust granted to the app itself, which is usually seen in repackaged apps. ContexIoT performs access control on the program path level and can enforce finer-grained policy to distinguish benign and attack logic in the same app.

As shown, our fine-grained context definition at inter-procedure control and data flow levels can successfully defeat these evasion attacks that can bypass other context-based security approach, showing that such a comprehensive definition can greatly improve the access control effectiveness in permission systems.

## VI. Implementation

We build the ContexIoT mechanism on the SmartThings platform, which supports yet the largest number of device types (204) among all the appified IoT platforms [15]. SmartApps are executed in the proprietary Samsung backend and our

Listing 4: Code snippet showing how the surveillance disabling attack app is patched with the ContexIoT secure logic

```
//Define Web API to get permission response
path("/response/:data"){
   action[POST: "onResponse"]
}
state.actionQueue = []
subscribe (location, "mode", handler)
def handler(evt){

  if(evt.value == "Away"){
//Pseudo function that collects context
      def context = collect_context()
//Enqueue the sensitive app logic
          state.actionQueue <<
              [device:"switch",command:"on()"]
          def id = state.actionQueue.length
//Request permission with the context
          httpPost(url,"['$id,$context]")
   }
}
def onResponse(){
  def id = params.data.id
    def permission = params.data.response
    if(permission == "Allow"){
//Dequeue the sensitive execution
     action = state.actionQueue[id-1]
//Execute the sink using dynamic code loading
     if('switch'==$action['device']){
       switch."$action['command']"()
     }
   }
   ...
}
```

prototype of ContexIoT automatically patches the app before they are submitted for execution. The patched security logic communicates the context information collected at app runtime to our own backend permission management server when security-sensitive behavior is triggered. We detail the key components of the app patching mechanism of ContexIoT and the end-to-end implementation.

### A. SmartApp Patching Implementation

Recall that our context definition in §V, which contains control and data flow attributes and also runtime values. ContexIoT enables the patched SmartApp to collect these context information at runtime by adding the logging logic to the instruction set of the original app. To reduce the runtime computation overhead of maintaining the context for the whole program, ContexIoT also employs a static analysis approach to (1) Identify a subset of the app code that require runtime logging to track the sensitive execution, and (2) Precompute some context information that is deterministic statically. Based on the annotations done by the static analysis, ContexIoT efficiently adds instructions to log the context attributes only for the subset of app code that is related to the sensitive behaviors of the untrusted app.

We define the *sinks* of SmartApps as the security-sensitive behaviors of the app, which contain both capability-protected APIs that are used to control or actuate the device, and other security-critical APIs such as `sendSMS()` and `setLocationMode()`. As of July 2016, 83 device-control APIs protected by 67 capabilities are supported by SmartThings, and will be recognized as sinks in our analysis. In addition, we also consider a set of SmartApp APIs that can be potentially used by attacker to carry out malicious payload. For example, malware can use the `setLocationMode()` to disarm the house by changing the mode to "*Home*", use `httpPost` to leak sensitive data, and use `sendNotificationToContacts()` to send phishing messages to the victim's contacts. We therefore collected 36 such APIs and added them to the sink API set.

*1) Static Analysis:* To model the lifecycle of the SmartApp and computes the minimum set of app code that can potentially affect the behavior of the sink, ContexIoT builds an ICFG for the SmartApp. The ICFG is constructed using the AST transformation support of Groovy language [6], which allows the static analysis to be performed directly on the Abstract Syntax Tree (AST) generated during the compiling process. Specifically, In the programming model of SmartApp, the app is not continuously running, app logic is embedded in different event handlers that are triggered by the events they have subscribed on. We adapt our design to the trigger-action based programming model of SmartApps, and models all the program entry points that can potentially be triggered by runtime events. In general, ContexIoT doesn't patch the app code that are not in the ICFG from the program entry points to the sinks. However, one exception is that it will patch all the statements that modify the value of global variables, which are shared among executions, since they will also determine the behavior of the sinks. The remote control attack shown in Listing 3 is one example.

The static analysis of ContexIoT further reduces the runtime overhead by precomputing the intra-procedural control-flow context for program statements, which doesn't contain dynamic method invocation (`GString`) in their control-flow paths. Similar to reflection, the dynamic method invocation support of Groovy can modify the control-flow at runtime. The dynamic features will be handled using runtime logging, however, except that, the intra-procedure control-flow information for all the other statements are deterministic statically, and ContexIoT annotates these statements with the precomputed context. Based on the call trace collected at runtime, the complete control-flow context for a certain sink is obtained by composing the annotated intra-procedural context of all the statements along the method invocation chain.

*2) Runtime Logging:* Excluding the intra-procedure control-flow context that is already been computed and annotated statically, there remain four major types of information that are required to be collected at runtime, in order to complete the context definition. (1) *Method invocation trace* is logged by adding a variable for each method in the app to keep track of its calling function. Every method call expression will set the variable of the callee function with the calling function's signature. Once a sink is triggered, the call trace can be extracted by tracing back the method signatures stored in these variables. (2) *Dynamic method invocation* is captured by using a variable to track the value of each `GString`, which can only be determined at runtime. For

example in the remote control attack shown in Listing 3, when the sensitive execution in the `handler` is triggered through dynamic method invocation, the patched SmartApp will gather the device and the method name and put them into the context. (3) *Runtime data* can be directly obtained from the variables without adding new instructions to track them. When a sink is reached at runtime, the context collection logic gathers the current value of all the variables that the sink statement is control-dependent on, and use them as the runtime data in our context definition. (4) *Data dependency* is critical to communicate the context with the user as described in §V, and we design and implement a dynamic taint analysis framework to track the data dependency information of sink-related variables.

The dynamic taint analysis of ContexIoT maintains a *taint environment* for each variable in the subset of app code output by the static analysis. Each program variable $v$ is associated with a taint environment $\gamma : v \rightarrow T$, where $T$ is a set of taint values $\{t_i | i = 1, ..., k\}$. Each taint value $t_i$ is associated with a variable $v_i$, meaning that $v$ is data dependent on variable $v_i$. In our design, variables in the taint environment $gamma$ include local, global, and function return variables, and are maintained as `JSON` objects in our implementation. Specifically, in the SmartThings programming language, the only global variable is the `state` object, which allows developer to store data into different fields of the object and shares the data across executions. And our taint logic is designed to be field-sensitive to precisely track the data dependency relationship of all the global variables in different fields of `state`.

The taint propagation of ContexIoT follows the generic approach [28] and handles some Groovy-specific operations such as the array insertion operation (<<), `closure`, and also the library functions of SmartThings. We manually summarized all the 85 SmartThings APIs available as of June 2016 in a file, which specifies how the taint values are propagated through the function variables to the return value. We also considered the side effect when modeling these library functions, which is the potential impact the functions have on the global variables. For example, once the `changeLocationMode()` function sets the global variable $Mode$, it will affect all the variables that depends on it. Our analysis handles such case by updating the taint environments for the corresponding variables when such function calls are executed.

In addition, ContexIoT also considers implicit flows, where the taint value is in the conditional statement that the sink is control dependent on. The implicit flow helps capture data dependency that is not directly propagated by assignments, and enables the detection of information leakage through side-channel. For instance, a malicious app we have constructed uses the light luminance as side-channel to send sensitive information such as the home occupancy and the lock status to attacker nearby, which will not be detected by explicit data flow analysis, but can be captured by implicit flow. We label each taint value with 2 boolean values $E$ and $I$ in its taint environment, for taint value coming from explicit flows ($E = true$) or implicit flows ($I = true$). When merging two taint values with different labels $E_1$,$I_1$ and $E_2$,$I_2$, the merged taint value's label is ($E_1||E_2$) and ($I_1||I_2$).

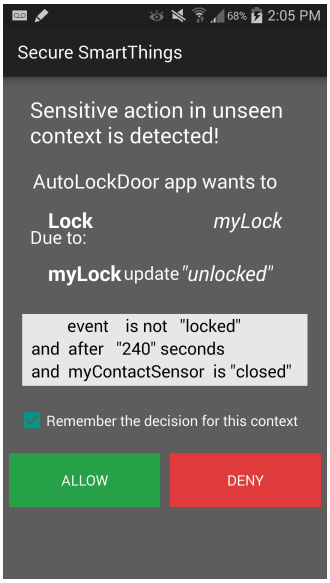
*3) Secure logic patching:* ContexIoT separates the execution flow of the sink into two asynchronous procedures: first requests the permission with the collected context, and then resumes the execution upon permission granting response. Listing 4 shows the code snippet of the surveillance disabling attack logic being patched by the ContexIoT. The original sink (`switch.off()`) is modified to the secure logic of enqueuing the action for future reference, and sending the collected context to permission server. ContexIoT patches the app with a Web API interface to receive the permission response in an separate process to overcome the restrictions of SmartThings on the execution time of each code block. On receiving the response from the server, the `onResponse` handler retrieves the information of the sink from the queue and execute it, if the server allows the execution in this context. The performance overhead of adding such secure logic is evaluated in §VII.

### B. End-to-End Implementation.

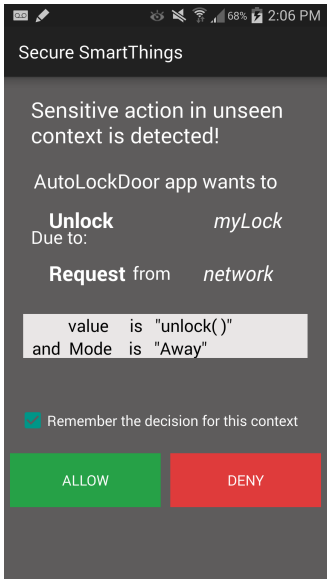
We set up the ContexIoT cloud backend service on a cloud instance in Google App Engine, which stores the previous user permission granting decisions. If no previous decision is found, it prompts the user using Google Cloud Messaging through the ContexIoT companion app to display the context and learn the user's decision. The basic approach for the permission service to distinguish two context is to compare the values of all the context attributes. However in our implementation, differences in the runtime value may be ignored under certain circumstances. The data floating in between SmartDevices and SmartApps are usually in the format of key-value pair (KVP), and in our threat model, we trust the SmartDevices to provide the authentic KVP, where the key reflects the real property of the data. Thus the runtime value are compared by their keys in our implementation, and we use the pin code snooping attack (Listing 2) to show that this design choice reduces the prompt frequency, while maintains the effectiveness. The malicious battery app subscribes on the device reports, and if the KVP of two battery reports are presented in the context as [”$battery\_level$” : 99] and [$battery\_level$” : 95], they will be considered as the same; However, if a pin code update report is sent out, the KVP in the context, which looks like [”$code$” : 9998], will be distinguished from previous paths, and prompted to user separately.

Figure 3 shows screenshots of the presented context information for a malware adapted from a real SmartApp called AutoLockDoor [4]. The legitimate functionality of this app is to automatically lock the door after a certain period of time, which is set by the user. In addition it checks the contact sensor of the door before issuing the `lock()` command to ensure that the door can be properly closed. Figure 3a shows the permission dialog for this legitimate execution path, which is consistent with the app description. However, this malware also includes a backdoor, which allows the attacker to unlock the door remotely via network commands when the user is away. As shown in Figure 3b, since this malicious logic is distinguishable in the control and data flow level, this backdoor logic is revealed clearly in the displayed context information.

It is important to note that this is only a proof-of-concept context presentation which dumps everything in the context structure to the dialog. Since our context definition contains the complete inter-procedure control and data flow information, future IoT platforms which adopt the ContexIoT approach can




(a) Legitimate logic: Automatically lock the door after a specified period of time




(b) Backdoor logic: Unlock the door when a remote command is received from the network

Fig. 3: Screenshots showing the benign and attack context in the malicious AutoLockDoor app

flexibly tune the context granularity, e.g., by shortening the length of recorded control flow or merging current context components, and also design better context presentation to meet their usability requirements.

## VII. Evaluation

In this section, we evaluate our prototype implementation of ContexIoT in (1) Effectiveness of secure logic patching; (2) Permission request frequency, which is important for the effectiveness of runtime permission system in practice [64], [19]; (3) Runtime performance overhead of the additional patching logic.

### A. Effectiveness of Secure Logic Patching

To evaluate the secure logic patching mechanism, we use ContexIoT to patch the 25 SmartApps we constructed, each representing a unique class of malware or an vulnerable app based on our IoT attack taxonomy in §IV. The SmartThings IDE provides a simulator that can model the behavior of native SmartThings devices without requiring a physical device [16]. Unfortunately, 3 of the attacks in our taxonomy involve 3rd party devices, for example a camera with advanced features used in the surreptitious surveillance attack [10], and thus cannot be dynamically tested. Thus, we test our system against 22 attacks in the runtime, and only manually examine the patched SmartApp code for the remaining 3 apps.

For effectiveness evaluation, we first check whether all the potential sinks in these SmartApps are patched with the secure logic, and find that ContexIoT accurately identifies and patches all 72 potential sinks including dangerous usage of `GString`. Next, we evaluate whether the attack execution paths in these

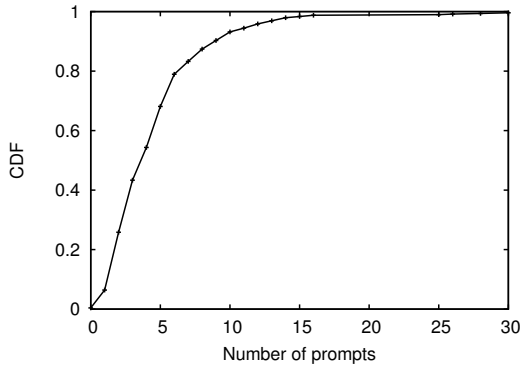



Fig. 4: CDF of the estimated life-time permission request prompts for 283 commodity SmartApps patched by ContexIoT

SmartApps can be distinguished from the remaining benign paths in the runtime. By triggering all the program paths of the 22 attack execution paths, we confirm that all of them can be successfully recognized without any ambiguity, and the other 3 attacks can also be identified based on the statically computed intra-procedural context information. Overall, these results show that our secure logic patching can accurately identify sinks and logging essential context to distinguish attacks execution paths.

### B. Permission Request Frequency

**Experimental setup.** To measure permission request frequency, we dynamically trigger the execution paths in a set of SmartApps using the SmartThings IDE simulator. In the experiment, we use 283 SmartApps out of the 502 commodity SmartApps we collected since the rest of them can not be simulated due to limited physical device support in the current simulator. To automate the test, we leverage the trigger-action programming model of SmartApp, i.e., app logics are all triggered by external events, to generate inputs. Leveraging the events generation support in the SmartThings IDE, we build an automatic SmartApp dynamic testing framework using web automation technique that can generate input to SmartApps and efficiently trigger different event handling logic.

Using our dynamic testing framework, we measure the life-time permission request number for each SmartApp by triggering all possible execution paths in the app. This is an upper bound estimation for a home user in practice, since typically a user only use a subset of all features in an app. For each SmartApp, we use the fuzz testing approach to randomly generate all types of external events in different triggering order to ensure good code coverage. Once a sink is triggered, we log the permission requests, and automatically grants permission to avoid double counting. The test stops only if no prompts are generated after 50 consecutive random events. As shown in Figure 4, the average life-time permission request number among the 283 SmartApps are only 3.5, which is far below the threshold that is considered to risk user habituation or annoyance [67], [64].

### C. Runtime Performance

In this section, we measure the performance overhead on the event handling latency introduced by the secure logic

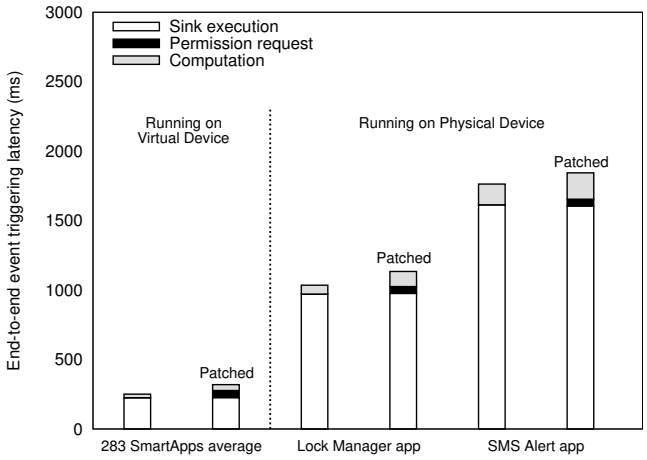



Fig. 5: Breakdown of the end-to-end event trigger latency with and without ContexIoT patching on virtual and physical devices

patching in ContexIoT. The end-to-end latency for an event execution can be broken down into 2 parts: (1) *computation* latency, which is the time taken in executing the app code, (2) *sink execution* latency, which is the time taken in communication between the SmartThings cloud backend and the physical device. In ContexIoT, the secure logic patching adds latency to the *computation* part since additional instructions are added to the SmartApp used for tracking the control and data flows. At the same time, it also adds *permission request* latency, which is taken in communicating with the ContexIoT cloud backend for permission granting. In this evaluation, we only measure the latency added by the ContexIoT system itself, and thus the decision making time taken for a user is not considered.

Using our dynamic testing framework, we inject events to trigger all the 916 event handling logic in the 283 SmartApps, and the measurement results are shown in Figure 5. Overall, we observe 67.1 ms (26.7%) additional latency on average when running those patched SmartApps on virtual devices. In addition, we find that the end-to-end latency is dominated by the sink execution latency, which is at least one magnitude higher than the additional latency from ContexIoT secure logic patching. Thus, we believe that the performance overhead from ContexIoT is negligible in real world scenarios.

Besides testing on virtual devices, we also evaluate the performance overhead for two events using physical devices: (1) Locking a Schlage Z-Wave lock [14] using a commodity lock manager SmartApp, and (2) Sending SMS to the user's phone from a SMS alert SmartApp once it detects motion sensor event. We trigger both events 50 times, and as also shown in Figure 5, the patched logic added only 9.6% and 4.5% delay on average. The breakdown of the end-to-end latency shows that compared with running on the virtual device, the time it takes to execute the command on physical device is significantly longer, which is likely due to the latency introduced by the wireless communication between hub and the device. Thus, the overhead of ContexIoT introduced in the computation and permission request procedures becomes negligible in the physical device settings.

## VIII. Usage Discussion

Acar *et al.* suggests to take both users and developers out of the loop as a potential solution to the permission comprehension problem [19]. Their proposed approach for achieving this is to enable the automatic generation of security policies. We believe that the rich context definition and flexible design of ContexIoT benefits innovations in this direction. For example, one potential approach can be to provide recommended context-based security settings to users for different apps, and the recommended settings come from the security preferences that are learned by ContexIoT from a group of expert users using the ContexIoT-patched apps.

Another possible approach that takes human out of the loop is automatic generation of policy based on the app logic and its interactions with user. For example, the SmartApp has a setup procedure during the app installation, which requires user to set some parameters that will guide the automation of the SmartApp (e.g., automatically locks the door when it has been opened for 2 minutes). Using the data-dependency tracking support of ContexIoT, the security logic can monitor how the app uses the user input. If it detects that the events corresponding to the unmodified user input triggers the user's desired action at the runtime, the execution can be automatically allowed since it conforms with the user specified routine. Natural Language Processing (NLP) technique may be required to infer the user desired action. We leave it as future work to explore these extended usages of the ContexIoT.

## IX. Conclusion

In this paper, we design and prototype ContexIoT, a context-based permission system for appified IoT platforms, which can support identification of fine-grained context defined at inter-procedure control and data flow levels, and runtime prompts with rich context information to help users perform effective access control. By comparing our context definition with those in previous context-based permission systems, we shows that our definition is more comprehensive and can defeat attacks that can evade these previous designs. Based on the extensive survey of existing and potential attacks on the appified IoT platform, we demonstrate that ContexIoT can effectively distinguish all attack context in the tested apps. Dynamic testing on 283 commodity SmartApps shows that ContexIoT introduces negligible performance overhead and has a low prompt frequency which is far below the threshold that is considered to risk user habituation or annoyance.

## Acknowledgements

We thank the anonymous reviewers for their valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1318306, CNS-1526455, CNS-1345226, and CNS-1318722 as well as by the Office of Naval Research under grant N00014-14-1-0440.

## References

[1] Android 5.0, Lollipop. https://www.android.com/versions/lollipop-5-0/.
[2] Android 6.0, Marshmallow. https://www.android.com/versions/marshmallow-6-0/.
[3] Apple HomeKit. https://developer.apple.com/homekit/.

[4] AudoLockDoor SmartApp. https://github.com/smartthings-users/smartapp.auto-lock-door/blob/master/auto-lock-door.smartapp.groovy.

[5] China-Made Handheld Barcode Scanners Ship with Spyware. http://www.tomsguide.com/us/chinese-barcode-scanner-spyware, news-19157.html.

[6] Compile-time Metaprogramming. http://groovy-lang.org/metaprogramming.html.

[7] Google Weave Project. https://developers.google.com/weave/.

[8] iOS 9. http://www.apple.com/ios/.

[9] Mirai Attacks. https://securityledger.com/2016/11/report-millions-and-millions-of-devices-vulnerable-in-latest-mirai-attacks/.

[10] Nest Cam Indoor. https://nest.com/camera/meet-nest-cam/.

[11] Our Project Website. https://sites.google.com/site/iotcontextualintegrity/home.

[12] Researchers exploit ZigBee security flaws that compromise security of smart homes. http://www.networkworld.com/article/2969402/microsoft-subnet/researchers-exploit-zigbee-security-flaws-that-compromise-security-of-smart-homes.html.

[13] Samsung SmartThings. https://www.smartthings.com.

[14] Schlage Z-Wave Lock. http://www.schlage.com/en/home/products/products-connected-devices.html.

[15] SmartThings Compatible Products. https://www.smartthings.com/compatible-products.

[16] SmartThings Simulator. http://docs.smartthings.com/en/latest/device-type-developers-guide/simulator-metadata.html.

[17] Surveillance cameras sold on Amazon infected with malware. http://www.zdnet.com/article/amazon-surveillance-cameras-infected-with-malware/.

[18] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1248–1259. ACM, 2015.

[19] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. Sok: Lessons learned from android security research for appified software platforms. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[20] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[21] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[22] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad. Proposed embedded security framework for internet of things (iot). In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, pages 1–5. IEEE, 2011.

[23] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[24] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Staying secure and unprepared: Understanding and mitigating the security risks of apple zeroconf. 2016.

[25] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146, 2013.

[26] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674, 2015.

[27] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devils footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[28] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 388–400. ACM, 2015.

[29] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[30] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *International Conference on Information Security*, pages 331–345. Springer, 2010.

[31] S. Demetriou, X.-y. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter. What's in your dongle and bank account? mandatory and discretionary protection of android external resources. In *NDSS*, 2015.

[32] W. Diao, X. Liu, Z. Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *2016 IEEE Symposium on Security and Privacy*. IEEE, 2016.

[33] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):5, 2014.

[34] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, D. Wagner, et al. How to ask for permission. In *HotSec*, 2012.

[35] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[36] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[37] B. Fouladi and S. Ghanoun. Honey, im home!!-hacking z-wave home automation systems. *Black Hat*, 2013.

[38] A. Francillon, B. Danev, and S. Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In *NDSS*, 2011.

[39] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy*, 2016.

[40] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 461–472. ACM, 2016.

[41] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[42] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu. Duet: library integrity verification for android applications. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 141–152. ACM, 2014.

[43] H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[44] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013.

[45] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda. Unveil: A large-scale, automated approach to detecting ransomware.

[46] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 978–989. ACM, 2014.

[47] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *NDSS*, 2014.

[48] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee. The price of free: Privacy leakage in personalized mobile in-app ads. 2016.

[49] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[50] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[51] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *NDSS*, 2014.

[52] H. Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[53] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. Iotpot: analysing the rise of iot compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[54] A. Rahmati and H. V. Madhyastha. Context-specific access control: Conforming permissions with user expectations. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, 2015.

[55] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[56] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *2012 IEEE Symposium on Security and Privacy*, pages 224–238. IEEE, 2012.

[57] E. Ronen and A. Shamir. Extended functionality attacks on iot devices: The case of smart lights. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 3–12. IEEE, 2016.

[58] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin. Flexdroid: Enforcing in-app privilege separation in android. 2016.

[59] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS16). ISOC*, 2016.

[60] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[61] B. Ur, J. Jung, and S. Schechter. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security (HUPS)*, 2013.

[62] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014.

[63] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646. ACM, 2013.

[64] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[65] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[66] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[67] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[68] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE, 2015.

[69] 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 *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2015.

[70] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security and Privacy*, pages 915–930. IEEE, 2015.

[71] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*, 2016.

[72] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

[73] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[74] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.