SmartAuth: User-Centered Authorization for the Internet of Things

Yuan Tian[1], Nan Zhang[2], Yueh-Hsun Lin[3], XiaoFeng Wang[2], Blase Ur[4], XianZheng Guo[1] and Patrick Tague[1]

[1]Carnegie Mellon University
[2]Indiana University Bloomington
[3]Samsung
[4]University of Chicago

Abstract

Internet of Things (IoT) platforms often require users to grant permissions to third-party apps, such as the ability to control a lock. Unfortunately, because few users act based upon, or even comprehend, permission screens, malicious or careless apps can become overprivileged by requesting unneeded permissions. To meet the IoT's unique security demands, such as cross-device, context-based, and automatic operations, we present a new design that supports user-centric, semantic-based "smart" authorization. Our technique, called *SmartAuth*, automatically collects security-relevant information from an IoT app's description, code and annotations, and generates an authorization user interface to bridge the gap between the functionalities explained to the user and the operations the app actually performs. Through the interface, security policies can be generated and enforced by enhancing existing platforms. To address the unique challenges in IoT app authorization, where states of multiple devices are used to determine the operations that can happen on other devices, we devise new technologies that link a device's context (e.g., a humidity sensor in a bath room) to an activity's semantics (e.g., taking a bath) using natural language processing and program analysis. We evaluate SmartAuth through user studies, finding participants who use SmartAuth are significantly more likely to avoid overprivileged apps.

1 Introduction

The rapid progress of Internet of Things (IoT) technologies has led to a new era of home automation, with numerous smart-home systems appearing on the market. Prominent examples include Samsung's SmartThings [49], Google's Weave and Brillo [23, 25] and Apple's HomeKit [5]. These systems use cloud frameworks to integrate numerous home IoT devices, ranging from sensors to large digital appliances, and enable complicated operations across devices (e.g., "turn on the air conditioner when the window is closed") to be performed by a set of applications. Such an application, called a *SmartApp* in Samsung SmartThings or generally an *IoT app*, is instantiated in the cloud. A user interface (UI) component on the user's smartphone enables monitoring and management. Like mobile apps, IoT apps are disseminated through app stores (e.g., the SmartThings Marketplace [47]), which accept third-party developers' apps to foster a home-automation ecosystem. Unlike mobile apps, IoT applications control potentially security-critical physical devices in the home, like door locks. Without proper protection, these devices can inflict serious harm.

A recent study on Samsung SmartThings brought to light security risks of such IoT apps, largely caused by inadequate protection under the framework [19]. Most concerning is the *overprivilege* problem in SmartApp authorization. Each SmartApp asks for a set of *capabilities* (the device functionality the app needs), and the user must choose the IoT devices to perform respective functions for the app (for example, see Figure 1). In mapping capabilities to devices, the user allows the IoT app to perform the set of operations defined by those capabilities (e.g., turn on a light, unlock the door) based on event triggers (e.g., the room becomes dark, a valid token is detected near the door). However, this *implicit authorization* suffers from issues related to coarse granularity and context ignorance, namely that an app given *any* capability (e.g., monitoring battery status) of a device (e.g., a smart lock) is automatically granted *unlimited* access to the whole device (e.g., lock, unlock) and allowed to subscribe to *all* its events (e.g., when locked or unlocked).

In addition to the overprivilege that results from conflating all capabilities of a single device, malicious IoT apps can overprivilege themselves by requesting unneeded, and sometimes dangerous, permissions. While asking users to authorize third-party apps' access to IoT devices would, in concept, seem to prevent this sort of overprivileging, prior work on permissions systems for mobile apps has repeatedly documented that users often fail to act based on, or even understand, these permission screens [18, 32, 33].

Even worse, unlike the Android permission model, which asks the user for permission to access specific resources on a *single* device (e.g., location, audio, camera), access control in a smart home system is much more

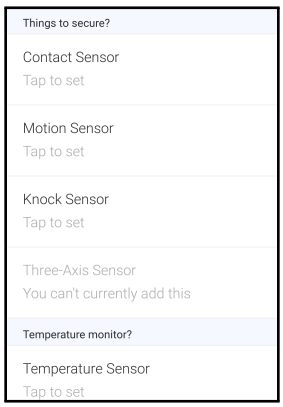

Figure 1: The installation interface of SmartApp SAFETY WATCH lists configuration options without connecting to higher-level app functionality. There is also no guarantee the app's actual behavior is consistent with its description.

complicated. The policy applies across devices, defining the operations of certain devices in certain scenarios based on observations of other devices (e.g., "ring the bell when someone knocks on the door"). Explaining such complicated policies to users is challenging, and effective authorization assistance should certainly go beyond what is provided by SmartThings (illustrated in Figure 1). In particular, it may be difficult for a user to understand what is being requested in the capability authorization UI, due to the gap between the app's high-level mission and the technical details of capabilities across devices. For example, a user may have no idea how reading from an accelerometer relates to detection of someone knocking on a door. Furthermore, in the absence of robust monitoring and enforcement by the platform, the authorization system provides little guarantee that the capabilities requested by an app actually align with the app description.

As a result, despite the existing authorization system for IoT platforms, there can exist a crucial *gap between what a user believe an IoT app will do, and what the app actually does*. The idea that privacy is context-sensitive has been widely studied [41]. For example, providing an individual's sensitive health information to a doctor for the purpose of treating the individual would often not violate the notion of contextual integrity, whereas providing the same information to the individual's financial institution would likely violate his or her privacy. A similar principle holds in the IoT ecosystem. If an IoT app describes its own purpose as unlocking the door when a visitor arrives, it is likely no surprise to a user that the app can unlock the door. If, however, the same app had advertised itself as a temperature-monitoring app, a user would likely find the app's ability to unlock the door to be a security risk.

In this paper, we propose new user-centered authorization and system-level enforcement mechanisms for current and future IoT platforms. We designed our approach, *SmartAuth*, to minimize the gap between a user's expectations of what an IoT app will do and the app's true functionality. To this end, SmartAuth learns about each IoT app's actual functionality by automatically harvesting and analyzing information from sources such as the app's source code, code annotations, and capability requests. Because the natural-language description developers write to advertise an app in the app store is the key source of users' expectations for what the app will do, we use natural language processing (NLP) to automatically extract capabilities from this description.

SmartAuth then compares the app's actual functionality (determined through program analysis) to the functionality developers represent (determined through NLP). This automated process is far from trivial because an in-depth understanding of the app focuses not only on the *semantics* of the app activities, but also their *context* among the app's broader goals. Our approach for achieving this level of contextual understanding relies on program analysis of the SmartApp's source code and applying NLP techniques to code annotations (e.g., the constant string for explaining the position of a sensor). We use further NLP to analyze the app description provided by the developer to extract higher-level information about the stated functionality, including entities (e.g., "a coffee machine"), actions (e.g., "taking a shower"), and their relationships (e.g., "turn on the coffee machine after taking a shower"). SmartAuth then compares such descriptions against insights from program and annotation analysis to verify that the requested capabilities and called APIs match the stated functionality, leveraging semantic relations among different concepts and auxiliary information that correlates them. For example, an annotation indicating a "bathroom" and the activity "take a shower" are used to identify the location of the humidity sensor of interest.

To minimize user burden, SmartAuth automatically allows functionality that is consistent between the app's natural-language description and code, yet points out discrepancies between the description and code since these are potentially unexpected behaviors. SmartAuth employs natural-language-generation techniques to explain, and seek approval for, these unexpected behaviors. The outcome of this verification analysis is presented to the user through an automatically created interface that is built around a typical user's mental model (for example, as in Figure 4). SmartAuth then works within the platform to enforce the user's authorization policy for the IoT app.

We incorporated SmartAuth into Samsung SmartThings as a proof of concept and evaluated our implementation over the 180 apps available in the SmartThings marketplace. SmartAuth successfully recovered authorization-related information (with 3.8% false positive rate and no false negatives) within 10 seconds. We found that 16.7% of apps exhibit the new type of overprivilege in which some functionality is not described to the

user despite passing Samsung's official code review [51]. Examples of cases found stem from vague descriptions (e.g., an app stating it can "control some devices" in your home) or hidden security-sensitive functionality (e.g., accessing and actuating an alarm without consent).

We also performed user studies to evaluate SmartAuth's impact on users' decision-making process for installing IoT apps[1]. In a 100-participant laboratory study, we found that SmartAuth helped users better understand the implicit policies within apps, effectively identify security risks, and make well-informed decisions regarding overprivilege. For instance, given two similar apps, one of which was overprivileged, using the current Samsung SmartThings interface, 48% of participants chose to install the overprivileged app in each of five tasks. With SmartAuth, however, this rate reduces to 16%, demonstrating the value of SmartAuth in avoiding overprivileged apps.

We also generated automated patches to the 180 SmartApps to validate compatibility of our policy enforcement mechanism, and we found no apparent conflicts with SmartAuth. Given our observations of the effectiveness of the technique, the low performance cost, and the high compatibility with existing apps and platforms, we believe that SmartAuth can be utilized by IoT app marketplaces to vet submitted apps and enhance authorization mechanisms, thereby providing better user protection.

Our key contributions in this paper are as follows:

- We propose the SmartAuth authorization mechanism for protecting users under current and future smart home platforms, using insights from code analysis and NLP of app descriptions. We provide a new solution to the overprivilege problem and contribute to the process of human-centered secure computing.
- We design a new policy enforcement mechanism, compatible with current home automation frameworks, that enforces complicated, context-sensitive security policies with low overhead.
- We evaluate SmartAuth over real-world applications and human subjects, demonstrating the efficacy and usability of our approach to mitigate the security risks of overprivileged IoT apps.

The remainder of this paper is organized as follows. In Section 2, we present the models and assumptions for our work. In Section 4, we describe the high-level design of SmartAuth. We present the details of our design and implementation in Section 5, and our evaluation of SmartAuth follows in Section 6. We highlight relevant related work in Section 7 and conclusions in Section 9.

[1]Our user studies were conducted with IRB approval.

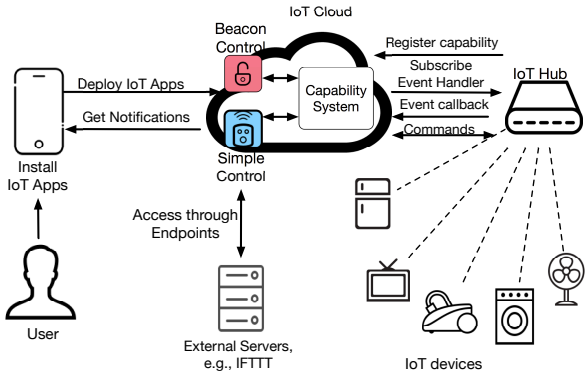

Figure 2: Users install IoT apps through mobile devices, allowing the vendor's IoT cloud to interact with the user's locally deployed devices. IoT apps pair event handlers to devices, issue direct commands, and enable external interaction via the web.

2 Background

2.1 Home Automation Systems

Home automation is growing with consumers, and many homeowners deploy cloud-connected devices such as thermostats, surveillance systems, and smart door locks. Recent studies predict home automation revenue of over $100 billion by 2020 [31], drawing even more vendors into the area. As representative examples, Samsung SmartThings and Vera MiOS [54] connect smart devices with a cloud-connected smart hub. Such vendors typically host third-party IoT apps in the cloud, allowing remote monitoring and control in a user's home environment.

Figure 2 illustrates a typical home automation system architecture. We use Samsung SmartThings to exemplify key concepts and components of such a system.

IoT apps written by third-party developers can get access to the status of sensors and control devices within a user's home environment. Such access provides the basic building blocks of functionality to help users manage their home, for example turning on a heater only when the temperature falls below the set point. Figure 2 depicts cloud-based IoT apps Beacon Control and Simple Control installed by a user from their mobile device and with access to the user's relevant IoT devices.

Current IoT platforms use *capabilities* [36] to describe app functionality and request access control and authorization decisions from users. Unlike permissions, capability schemes are not designed for security, but rather for device functionality. A smart lighting application, for example, would have capabilities to read or control the light switch, light level, and battery level. Due to complexity, capabilities in home automation platforms are often coarse grained. One capability might allow an app to check several device attributes (status variables) or issue a variety of commands. This functionality-oriented design creates potential privacy risks, as granting an app

a capability for a device allows it to access all aspects of the device's status and fully control the device.

An IoT app can also act as a *web service* (an *endpoint* in Samsung Smartthings) to interact with the outside world. Such an app handles remote commands from servers and reacts accordingly. Many home automation platforms support standard authentication and authorization mechanisms such as OAuth [4, 50] to grant permission to third parties for commanding or accessing devices.

2.2 NLP Technologies

Since our approach analyzes app descriptions and gaps in users' mental models, we rely on several existing tools and techniques for natural language processing (NLP). The following tools are employed in our work.

Word2Vec [22] is a state-of-the-art tool used to produce word embedding that maps words to vectors of real numbers. Specifically, Word2Vec models are neural networks trained to represent linguistic contexts of words. We use Word2Vec to determine the relation between two words by calculating the distance between the words in the embedding space. Word2Vec has many advantages over previous approaches, including catching syntactic and semantic information better than WordNet [39] and achieving lower false positive rates than ESA [21].

Part-of-speech (POS) tagging is used to identify a word's part of speech (e.g., noun or verb) based on definition and context. A word's relations with adjacent and related words in a phrase, sentence, or paragraph impact the POS tag assigned to a word of interest. In our work, we rely on the highly accurate Stanford POS Tagger [38].

We also rely on the typed dependencies analysis [27] to understand the grammatical structures of sentences, grouping words together to recognize phrases and pair subjects or objects with verbs. The Stanford parser applies language models constructed from hand-parsed sentences to accurately analyze sentences of interest.

3 IoT App Security Challenges

Beyond basic overprivilege where an app requests an unnecessary capability, previous IoT research has studied two additional types of overprivilege [19]: coarse capability and device-app binding. In the former, a capability needed to support app functionality also allows unneeded activities. In the latter, a device is implicitly granted additional capabilities that are not needed or intended.

We have identified an additional type of overprivilege that relates not only to the functionality of the IoT app, but also to the user's perception of the app functionality, as seen through the app description. We observe that several IoT apps exhibit capability-enabled functional behaviors that are not disclosed to the user, causing a discrepancy between the user's mental model and the actual privilege of the app. We refer to this problem as *undisclosed overprivilege*. This kind of overprivilege has been discussed in mobile apps, but was never studied in the IoT space. An example of this type is an IoT app that describes the ability to control lights while requesting capabilities to read and control a door lock. Previous approaches may not flag this app as overprivileged, as long as the capabilities are used. In fact, even after a majority of Samsung SmartThings apps were removed from the market due to the previously reported overprivilege issues [19], we found that 16.7% of the remaining apps still exhibit overprivilege risks.

Remote access is also an important security risk, as it enables apps to send sensitive data to and receive commands from third-party servers. In our study, we found 27 cases of such behavior, including cases where data was shared without user consent, a clear privacy concern. A SmartApp's ability to act as a web service expands the attack surface and potentially allows a malicious server to send dangerous commands to an app running on a user's smart devices, even though users may not expect such remote control. We observed 17 apps with this behavior. Similar to the undisclosed overprivilege, remote access does not match the user's mental model, which illustrates a gap in the current configuration and approval process.

Based on these observations, a general threat in the IoT app landscape is the ability for a malicious or compromised IoT app to steal information from sensors or home appliances or to gain unauthorized access to IoT device functionality. Even if the IoT platform itself is secure and trustworthy and previous issues of authentication and unprotected communication are patched [7, 19, 40], such issues with malicious apps may remain.

4 SmartAuth Design Overview

We next present the high-level design of SmartAuth, including design goals, architecture, and security model.

Given the unique security challenges of smart home systems, we believe that an authorization system for IoT apps should be designed to achieve the following goals.

- *Least-privilege*: The system should grant only the minimum privileges to an IoT app, just enough to support the desired functionality.
- *IoT-specific*: Compared with authorization models for mobile devices, which are designed to manage a single device, the authorization system for a smart home framework should meet the needs for multi-device, context-based, automatic operations. Permission models based on manifest permissions or runtime prompts, such as those employed in Android or iOS, either do not allow users to make context-based decisions or cannot satisfy real-time demands (e.g., approval to actuate an alarm when fire is detected).
- *Usable*: The authorization system should be human-centric, minimizing the burden on users while supporting effective authorization decisions.

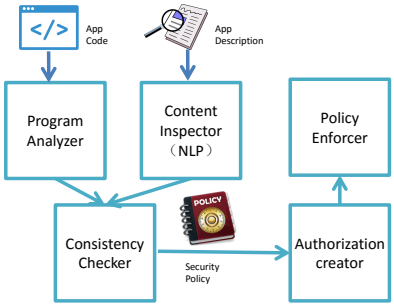

Figure 3: We provide a high-level block diagram to illustrate the design overview of our SmartAuth system.

- *Lightweight*: The authorization approach should not inhibit performance with significant overhead.
- *Compatible*: The authorization approach should be compatible with existing smart home platforms and applications without breaking app functionality.

Since authorization decisions are made by humans, providing a human-centric design to help users to make well-informed decisions is critical. Toward this goal, we design an intelligent authorization system that recovers adequate semantic information from IoT apps and presents it to users in an understandable way. We leverage semantic analysis techniques to automatically interpret the description, code, parameters, and annotations of an IoT app. We analyze the semantic meaning of these components to discovery inconsistency, then automatically generate an authorization interface explaining the findings to the user.

Based on these design principles, our SmartAuth system includes five components: a program analyzer, a content inspector, a consistency checker, an authorization creator and a policy enforcer, as illustrated in Figure 3. The code analyzer extracts the semantics of an IoT app through program analysis and NLP of app code and annotations, creating a set of *privileges* that support the app functionality. In parallel, the content inspector performs NLP on the app description to identify the required privileges explained to the user. The consistency checker compares the results of code analysis and content inspection to generate security policies and identify discrepancies between what is claimed in the description and what the app actually does. These policies and information needed to support user decisions are then presented through an authorization interface produced automatically by the authorization creator. The resulting policies are then implemented by the policy enforcer.

Our security policy model for the smart home architecture is described in the form of a triple (E, A, T). Item E represents the events, inputs, or measurements involving IoT devices and describes the *context* of the policy. Item A represents the actions triggered by elements of E, including commands such as "turn on". Item T represents the group of *targets* of the actions in A, such as a light

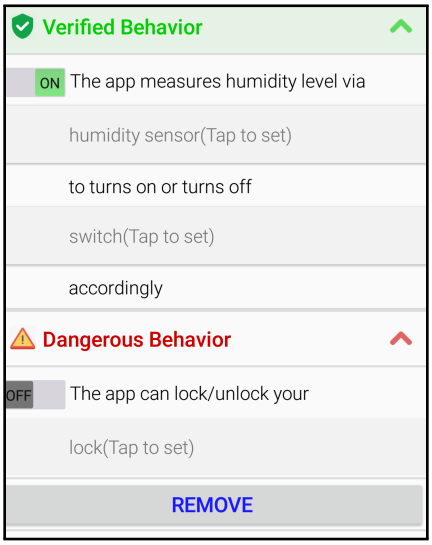

Figure 4: We illustrate the security policy generated for the HUMIDITY ALERT app, which is communicated to the user to request authorization. The indication of behavior type is discussed in more detail in Section 5.3

receiving a command, noting that an empty target implies broadcast of a message or command. This model captures typical IoT app functionality, as apps are designed to issue commands to respond to observed state changes.

This model describes not only the policy produced by the authorization process, but also the privileges both claimed in an app's description and recovered from its code. Analysis of the policy actions thus allows identification of overprivilege and presentation of conflicts or situations that require the user to make a policy decision. Figure 4 illustrates an example of such policies.

5 Design and Implementation

In this section, we detail our design and implementation of SmartAuth.

5.1 Automatically Discovering App Behaviors

To extract an app's security-critical behaviors, we perform static analysis on the app's source code and use NLP techniques on code annotations and API documents.

We collected the source code for 180 Samsung SmartThings apps from an source-level market in May 2016 [48]. This represents 100% of open-source SmartApps and 80.2% of all SmartApps at that time.

For each app collected, we parse its code and create an Abstract Syntax Tree (AST) from the code, resolving classes, static imports, and variable scope. We choose to do AST transformation for the app analysis for two reasons. First, we have access to the source code which is suitable for AST transformation. To enable further analysis, we extract the key components of method names, variable names and scope, a variety of statements. Second,

SmartThings apps are written in Groovy, which transfers method calls into reflective ones and creates challenges for existing binary analysis tools to deal with reflections. Therefore, binary analysis is not suitable.

We first extract capabilities, which are directly associated with security behaviors, by searching for the term "capability" in the preference block. We incorporate any search results into the list of requested capabilities collected from the IoT app documentation.

To further understand how the IoT app is intended to make use of the requested capabilities, we analyze the commands and attributes associated with each requested capability. To enable this analysis, we maintain a global mapping of capabilities to commands and attributes, noting that one capability may involve multiple commands and attributes. Since a SmartApp gets status updates by subscribing to events, namely as `subscribe(dev,attr,hndl)` for relevant IoT device `dev`, device attribute `attr`, and invoked method `hndl`, we use this global mapping to search the AST for relevant commands called and attributes subscribed.

We then generate the security policy, starting from the method invoked on event subscription and perform forward tracing. We first analyze the invoked function's code blocks to determine whether it contains conditional statements, which we analyze immediately. Otherwise, we trace into the called function. Within condition blocks, we look for (1) the event and (2) the object and action. The invoked function of the event subscription takes a subscription parameter that carries information about the event. Combining this information with the variables in the AST, we identify both the event and associated capability. We further identify the action triggered by the event. For example, for an app that controls a heater based on a threshold temperature setting, it is critical to distinguish whether the app turns the heater on or off. We thus search the result statement for commands that control a device. If found, we continue our analysis to match the capability through variable analysis. Otherwise, we record the event and trace into the called function.

The previous analysis covers an app's direct access to IoT devices, which we use to identify overprivilege. We also analyze whether the app has remote access to servers other than the SmartThings cloud. We consider two types of remote access: whether the app sends data to the remote server and whether the app works as a web service to take commands from the remote server. Both cases are privacy-invasive and may violate user expectations. We search the AST to match patterns including OAuth, `createAccessToken`, and `groovyx.net.http`.

We also examine clues from code annotations (e.g., comments and text strings) to gain further information about context and device state. We apply Stanford POS Tagging and analyze the nouns to determine whether they represent location or time contexts. We find that most context clues in smart homes relate to a place in the home, such as a bedroom. For example, we can extract that the humidity sensor is associated with bathroom from understanding the annotation in the following code snippet.

Listing 1: Code Snippet about Device Selection

```
section("Bathroom humidity sensor") {
  input "bathroom", "capability.
      relativeHumidityMeasurement", title:
      "Which humidity sensor?"
}
```

5.2 Analyzing App Descriptions

A key goal of our project is revealing any discrepancy between what the app claims to do and what it actually does. To find such discrepancies, we use NLP techniques to extract the security policy from the app's free-text description and program analysis to compare it with the security policy extracted from the code. We extract and correlate the behaviors in three layers: (1) entity, (2) context and action, and (3) condition.

We infer the security policy based on human-written, free-text app descriptions. To do this, we first identify the parts of speech of the words used, then analyze the typed dependencies in the description. Nouns and verbs are often related to entities; for example, movement might be related to a motion sensor. From the structure of the descriptions, we can then infer relationships between entities by identifying the typed dependencies. For instance, in the phrase "lock the door", the noun $door$ is the accusative object of the verb $lock$ (written as $dobj(lock, door)$). In the corresponding security policy, $lock$ is the $action$ and $door$ is the $target$. Most cases are more complex than this example, and our more comprehensive analysis follows.

Specifically, we use the Stanford POS Tagger to identify parts of speech and the Stanford Parser to analyze sentence structure, including typed dependencies, as illustrated in Figure 5. We follow standard NLP practices, such as removing stop words (e.g., "a," "this") [11].

We analyze noun and verb phrases to pinpoint the relevant entities, as these phrases usually describe core app functionality. However, as discussed later, analyzing a developer's description comes with non-trivial challenges. In addition, the device's context can significantly impact the implications of the entities. To overcome these difficulties, we design and implement the following process.

The most straightforward case is when the description explicitly includes the name of the entity (e.g., humidity sensors). If so, we match words directly.

Because of language diversity, the first step may not produce meaningful results. However, even when the description does not contain the device name, the description might contain contextual clues related to specific

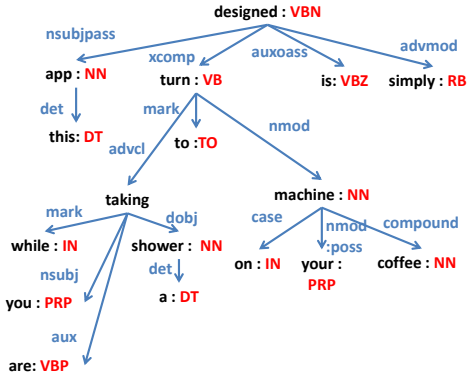

Figure 5: We illustrate example NLP analysis for the COFFEE AFTER SHOWER description: "This app is designed simply to turn on your coffee machine while you are taking a shower." Red and blue characters respectively indicate parts of speech (e.g., "VB" for verb) and typed dependencies (e.g., "advcl" for adverbial clause modifier).

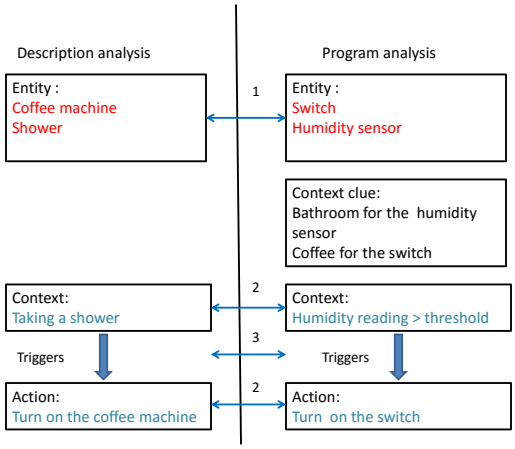

Figure 6: We illustrate the three-step policy correlation for the COFFEE AFTER SHOWER app. 1) We apply the context clues "bathroom" and "coffee" for entity correlation. 2) We use the attribute model and command model to extract and correlate the context and action. 3) We use typed-dependency analysis and causal relationship model to correlate the policies generated from the description and program analysis.

devices. For example, mention of flood detection hints to a humidity or moisture sensor. We evaluate relationships between words in the descriptions to the relevant devices through a word distance model that combines Word2Vec with a language model. Our language model includes a vocabulary of three million words and phrases, trained on roughly 100 billion words from Google News [24].

The most challenging case is when the words in the description are not directly related to the entity in the generated security policy. In this case, we compare the description to the context clues from code annotations. In the example in Figure 6, we first extract the entity "bathroom" (the context clue) from the annotation for the *humidity sensor* (`capability.relativeHumidityMeasurement`) identified through code analysis (Section 5.1). We link this entity to the entity "shower" using the semantic relation revealed using Word2Vec. In this way, we link "taking a shower" to the humidity sensor. Similarly, our technique relates "coffee machine" in the description to the *switch* device (`capability.switch`) recovered from the code.

However, simply connecting an entity in the description to a device in the code is insufficient to determine whether only expected behaviors (as specified in the description) happen. For example, "lock the door when nobody is at home" and "unlock the door when nobody is at home" have starkly different security implications. To compare the semantics of an activity in description to the operation of a device, we utilize a knowledge-based model. Specifically, we parse the API documentation of SmartThings to generate the attribute and command models, namely the keyword sets for attributes and commands that represent their semantics. We thus parse words and phrases in the description connected to the entity-related word. This can be done by going through the typed-dependency graph. For example, in Figure 6, we have identified that "coffee machine" is an entity related to the phrase "turn on". Such phrases will be compared with the keywords in the attribute and command models to find matches.

After comparing the devices used in the code to those mentioned in the description, we also need to know whether the actual control flow matches that of the policy model. The causal relationship is critical for multi-device management where devices have impacts on each other. For example, two IoT apps may both ask for access to a door lock, motion sensor on the door, and presence sensor. A benign app might unlock the door when a family member is at the door and locks it when someone other than a family member is there. A malicious app might unlock the door anytime anyone is there. These two apps use the same devices, but with different control flow.

To perform causal analysis, we analyze the typed dependencies and build knowledge-based models of causal relationships. The causal relationships model is built with sentence structures and conjunctions related to conditional relationships. We apply the initial models to the descriptions to identify which devices caused other devices to change status. For example, the sentence "turn on the light *when* motion is sensed" represents that motion status is the cause, and turn on the light is the result. At the end of this process, we obtain *verified behaviors* that match in code and descriptions and *unexpected behaviors* that exhibit a mismatch.

5.3 Authorization Interface Generation

Towards making usability a first-order concern in designing our authorization scheme, we first conduct an online

survey to understand users' mental models related to IoT app installation and the overprivilege problem. Using Amazon Mechanical Turk, we recruit adult participants who have experience using smartphones. To avoid biasing participants towards fraudulently claiming experience with SmartThings to participate in the survey, we do not require that participants have used any smart home platforms to take the survey. However, we only analyze data from the 31.6% of the survey participants who have previous experience with SmartThings. Please refer to the appendix refquestion1 for the sample survey questions.

Our survey asked about: (1) experience using IoT platforms and demographics, (2) the factors considered when installing third-party IoT apps, and (3) perspective on smart home capabilities. We received responses from 300 participants who had used SmartThings, identified by an average age of 30.8 years old (age range is 18-60) with a gender breakdown of 32% female, 67% male.

We asked participants to respond using a five-point scale (strongly care, care, neither care or not care, not care, strongly not care) about factors they consider when deciding whether or not to install a SmartThings app. App functionality (66% strongly care and 24% care) and privacy (57% strongly care and 28% care) were the factors participants stated they cared about most in their decision.

To understand participants' perspectives on smart home capabilities, we asked them to rate the sensitivity of different IoT device functions and to compare the sensitivity of SmartThings capabilities and Android/iOS permissions. To ensure that participants understood what we meant by *smart home capabilities*, we both formally defined the concept and demonstrated it using an example screen shot from a SmartThings device permission screen.

We asked participants to rate the sensitivity of eight IoT device behaviors on a four-point scale (from "not sensitive" to "very sensitive"). We find that participants have very different risk perceptions for different behaviors of the same IoT device. For example, we find the average sensitivity rating for app's ability to *unlock* their door is 3.28, whereas reading the battery level of their door is only 1.87 (Mann–Whitney $U = 21350, n1 = n2 = 300, P < 0.001$ two-tailed) [8]. These sharp distinctions highlight the importance of increasing the transparency to users about what precise behaviors an app will perform in the home, rather than considering all behaviors for a particular device monolithically. Our approach of automatically identifying discrepancies between the actual behavior of an app determined through program analysis and the free-text app descriptions that users generally rely on when considering whether to install apps [32] better supports these distinctions.

To this point, most work on app permissions focuses on smartphones. We thus asked participants to specify whether they considered Android/iOS permissions and smart home capabilities equally sensitive, Android/iOS permissions to be more sensitive, smart home capabilities to be more sensitive, or whether they were unsure. In support of our continued study, 69% of participants indicated that they considered smart home capabilities to be more sensitive than Android/iOS permissions. Participants provided a free-text explanation of why, and we performed qualitative coding on these responses by two researchers (with an agreement rate of 90.3%). The leading reason participants found IoT apps more sensitive is that they perceived the home environment to inherently present greater risks. One participant wrote:

> "Smart home compromises can inflict serious damage or injury. Imagine being locked in your house, with the heat cranked up. Or an invader monitoring your location in the house, or studying your patterns. The risk involved in a smartphone knowing your location or accessing the devices hardware, like reading contents, contacts or accessing the camera are far more limited in potential effects by an attacker."

In generating the user interface, we aim to minimize the burden on the user and provide information that matches the user's mental model of the system. We rely on a policy model that links app functionality with authorization. We first automatically summarize the security policy, removing redundant logic, and then create language models to translate the security policy into a human-understandable description. We achieve this task using state-of-the-art natural language generation tool SimpleNLG [1], a realization engine that generates and linearizes syntactic structures. The automatically generated description details what device attributes and commands are being used, and why. For example, the app monitors the temperature from the temperature sensor and whether someone is at home by the presence sensor to turn on a heater when it is cold and someone is home.

We designed our authorization approach to better align users' mental models with the actual behaviors of smart home apps, as well as to reduce user burden during the authorization process. Because many users rely on app descriptions, rather than permissions screens, to evaluate smartphone apps [32], one way of reducing user burden is to assume that a user would implicitly grant an app the permission to perform actions stated in the app description. While any assumption that a user's actions with an app perfectly follow the user's intent is necessarily flawed, prior work on smartphone permissions [32] suggests that assuming a user would permit an app to perform the behaviors described in its app description is practical. The assumption of a user would permit an app to perform the behaviors in its description is likely at least as robust as assuming that a user intended to grant the permissions

specified on a permissions screen. We therefore minimize users' burden by automatically granting the attributes inferred from the app description. For the attributes absent from the app description, we present the user with our automatically generated description of the policy model rather than potentially confusing settings.

To help users understand the potential risky behaviors, we use risk level indicators with corresponding colors and icons in the user interface, as illustrated in Figure 4. We define three indicator categories: verified behaviors that match the claimed functionality, unexpected behaviors that are not sensitive, and dangerous behaviors that are unexpected and risky. We determine these risk levels by asking security experts and average users to rate their perceived risk based on status changes and device operations. Specific parameters are further described in the Appendix.

5.4 Policy Enforcement

Once a user sets his or her policy settings through the user interface, we enforce the policy end-to-end by blocking unauthorized command and attribute access.

Our proof-of-concept implementation of the policy enforcement mechanism operates locally on the device through the use of REST APIs, mimicking the ideal integration directly into the SmartThings Cloud. We patched existing SmartApps by substituting each command or attribute function call with an equivalent REST API call to the module that includes the device handler, command or attribute name, and any additional parameters. After the module processes the request, a return value is sent back to the patched app and handed to the code that invokes this command or attribute, which is transparent to the original app. Similarly, the patched app also subscribes to events by connecting to the enforcement module. Appendix A further details how we patch existing Groovy apps to interact with our policy enforcement module.

The policy enforcement mechanism starts when the user begins to install a SmartApp. The user is directed to our enhanced interface to set up the relevant devices and policies for the app. This information is transmitted to the policy enforcement module to ensure that the app can only access what the user allows. Based on the policies, the module will make two type of decisions.

First, whenever the module receives a command or attribute request from a patched app, it will extract the device ID and actions and check the associated policies from the database for proper authorization. If allowed, the module will forward the request to the cloud service to execute and respond, after which the module will forward the response to the patched app. If denied, the request will be dropped and an error message will be returned to the patched app. We expect that SmartApps will already be designed to handle error messages, so the denial of requests should not impede normal operation. We further analyze compatibility in Section 6.3. Second, whenever there is an event reported by the SmartThings Cloud, the module will retrieve the associated app IDs and policies from the database and forward the event only to the apps that are allowed to access the event according to the app policy. The module thus blocks all unauthorized subscribe, command, and attribute requests.

6 Evaluation

We evaluate SmartAuth in several dimensions, finding SmartAuth is effective at automatically extracting security policies, significantly helps users avoid overprivileged apps, and adds minimal performance overhead when enforcing users' desired policies.

6.1 Effectiveness in Extracting Policies

We first evaluate SmartAuth's ability to accurately identify unexpected behaviors. To this end, we manually analyze the description and the code of the 180 available SmartApps and compare the results to those of the automatic analysis. In this process, we do not observe any false negatives, though we identify seven false positives (3.9%) in which SmartAuth flagged a behavior as unexpected though manual analysis of these cases suggests otherwise.

In two of these cases, the app uses a product name to represent a device, but the product name is not relevant to its functionality; for example, the Mini Hue Controller app uses the Aeon Minimote[2] device. In two other cases, the app references another app by name to explain its functionality; for example, the Keep Me Cozy Two app claims that it "works the same as Keep Me Cozy, but enables you to pick an alternative temperature sensor in a separate space from the thermostat." These cases could be eliminated using named entity analysis to identify the referred app and merge the behaviors and descriptions accordingly. In another case, the description has complicated logic spread through several sentences, causing the description to be ambiguous. In the two remaining cases, the correlation of the context is not intuitive, even for a human reader; for example, a relationship between vibration of the floor with someone waking up at night is not immediately clear.

6.2 Impact on Users

We first describe our user study to evaluate how SmartAuth impacted users' app-installation decisions, followed by additional data on the usability of SmartAuth itself.

We performed a between-subjects user study with 100 participants recruited from across our institutions. Participants completed app installation tasks in our lab using phones we provided and answered several relevant questions. We required participants to be adults who regularly use a mobile device and are knowledgeable about home automation systems. We verify participants understand

[2] http://aeotec.com/homeautomation

key concepts of smartphones and home automation using screening questions. For example, we ask them how IoT apps are installed and what purposes IoT apps serve. We also ask questions about demographics, as well as questions about their experiences installing IoT apps. The protocol takes around 20 minutes. For the 100 participants in our study, their ages ranged from 19 to 41 years with a mean age of 25.7 years, and 59% of participants reported as male and 41% as female. The participants have education backgrounds ranging from high school to graduate school. 68% of participants have a technical background (engineers or students in computer science or related field).We carefully avoid the IoT developers when we recruit in the company because they are very familiar with the system and their results might be biased.

The study's primary task is a series of selection tasks for IoT apps using the phone we provide. For five different types of IoT apps, the participant chooses between one of two similar apps. Each of the two apps in a pair has identical functionality, yet only one of the two apps in a pair is overprivileged. To prevent this difference in permissions from being the obvious variable of interest, we used apps whose titles and descriptions were roughly comparable. For example, participants choose between "Lights Off with No Motion and Presence (by Bruce Adelsman)" that will "Turn lights off when no motion and presence is detected for a set period of time" and "Darken Behind Me (by Michael Struck)" that will "Turn your lights off after a period of no motion being observed."

Each participant is randomly assigned into one of two groups, specifying whether they will see Samsung SmartThings' authorization interface or SmartAuth while completing all tasks. For each of the five app-selection tasks, participants saw the app installation page with two choices. We asked the participant to choose only one of the two apps to install, and to explain why.

For each of the five tasks, between 48% and 60% of participants who saw the current SmartThings interface chose the overprivileged app, as shown in Figure 7. Even though the current Samsung SmartThings authorization interfaces shows users a list of the devices the app can access, including potentially unexpected devices, this current interface did not help users avoid overprivileged apps.

In contrast, 84% of participants who saw the SmartAuth interface successfully avoided the overprivileged app, differing significantly from the current SmartThings interface (Holm–Bonferroni corrected χ^2, $p \leq .022$ for all five tasks). Note that for two of the tasks (A and B in Figure 7), the overprivilege was a potentially dangerous behavior (e.g., unlock a door), whereas the overprivilege for tasks C–E was potentially less risky (e.g., learn the temperature). For tasks A and B with dangerous overprivilege, only 10% and 6% of SmartAuth participants, respectively, chose the overprivileged app, compared to

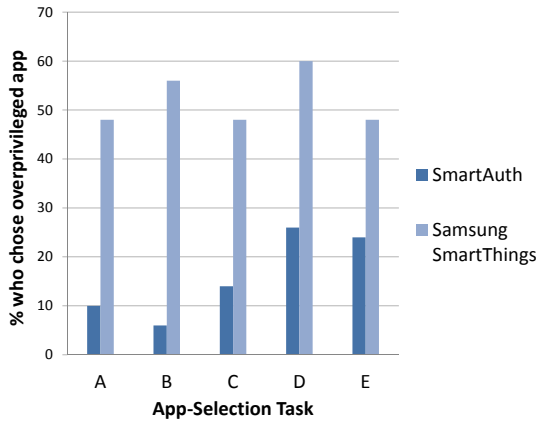

Figure 7: For 5 tasks, participants chose between two similar IoT apps, one of which was overprivileged. This graph shows the proportion of participants who chose the overprivileged app. Similar to what one would expect from random selection, around half of participants who saw the Samsung SmartThings interface chose the overprivileged app. In contrast, only between 6% and 26% of SmartAuth participants chose the overprivileged app.

48% and 56% for the current SmartThings interface. Even when they still chose the overprivileged app, we found that many SmartAuth participants were aware of the overprivilege, yet said they either did not care about the unexpected behaviors or thought the app might benefit from these behaviors in the future.

In addition to evaluating SmartAuth's impact on user behavior, we also measure the usability of SmartAuth itself. In the laboratory study, after users choose among pairs of apps and answer questions about privacy, we ask questions to elicit their perceptions of what the interface communicated to them. For some of these questions, participants respond to statements on a five-point Likert scale (from "1: strongly disagree" to "5: strongly agree").

The first statement gauges the apparent completeness of explanations ("I feel that the app interface explains thoroughly why the app can access and control these sensors/doors"); SmartAuth participants were more likely than those who used Samsung SmartThings to agree (SmartAuth mean 4.06, SmartThings mean 2.40, Mann–Whitney $U = 337.5, n1 = n2 = 50, P < 0.001$ two-tailed). The second statement measures user comfort in making decisions ("I feel confident to make a decision whether or not to install the app after reading the interface"); SmartAuth participants were significantly more confident in their decisions (SmartAuth mean 4.12, SmartThings mean 2.46, Mann–Whitney $U = 320.5, n1 = n2 = 50, P < 0.001$ two-tailed). The third statement evaluates perceived difficulty of finding information ("It is difficult to find the information from the interface"); SmartAuth participants were more likely to *disagree* with this difficulty, meaning they found it easier (SmartAuth mean 2.72, SmartThings mean 3.56, Mann–Whitney $U = 713, n1 = n2 =$

$50, P < 0.001$ two-tailed).

We also asked open-ended questions about what factors participants consider when deciding to install an app. Both SmartAuth and Samsung SmartThings participants focused on two factors in common: functionality and ease of configuration. However, SmartAuth participants also discussed privacy and unexpected or dangerous behaviors as a major factor. In comparison, *only one of the 50 Samsung SmartThings participants pointed out a mismatch between the description and the authorization screen*.

6.3 Performance and Compatibility

To evaluate the performance impact and ease of deployability for SmartAuth, we collected all 180 open-source SmartApps in the Samsung SmartThings marketplace at the time of research. In order to demonstrate that SmartAuth is both lightweight and backward compatible, we performed two performance tests: (1) pre-processing performance comprising program analysis, description analysis, behavior correlations, and policy description generation and (2) run-time performance comprising authorization interface generation and policy enforcement.

For testing the pre-processing performance, we timed the generation of the policy description for each of the 180 apps, averaging over 10 trial runs. On a 3.1 Ghz Intel Core i7 CPU with 16 GB memory, the pre-processing overhead for an app is 10.42 seconds on average. Since pre-processing is a one-time cost and can be done offline, we believe that the performance is reasonable even for vetting a large number of applications.

For the run-time performance test and compatibility test, we instrumented the SmartApp to interact with our policy server running on the Amazon EC2 cloud, which enforces the rules defined by the user. Given our purpose of evaluating the compatibility of our technique with existing SmartApps, we set the authorization policies (granting permissions to certain commands, attributes and event handlers) ourselves, instead of letting the user do that, as would happen in practice. We designed our experiments to test the technique in the worst-case scenarios. That is, we assume users would reject all unexpected and dangerous behaviors, requiring the maximum amount of policy enforcement. To enable large-scale testing without requiring the purchase of every physical SmartThings device, we used Samsung's online SmartApp simulator platform[3]. Instrumented apps are then installed on the simulator, and their functionalities are tested with simulated IoT devices.

As shown in Figure 8, we recorded the delay incurred by different command, attribute, and event handler actions. We performed 1800 experiments among the 180 SmartApps on a cloud server with 3.1 Ghz Intel Core i7 CPU and 1 GB memory. SmartAuth incurs an average delay of 35.4 msec, which is small relative to the dominant

[3] https://graph.api.smartthings.com/

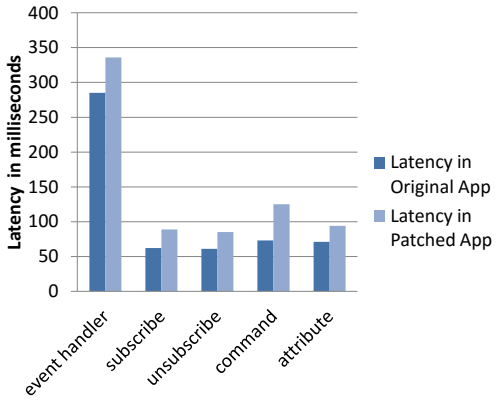

Figure 8: We plot the average delay of various functions in the SmartThings platform. The darker bar in each pair represents the delay in the unmodified platform with virtual devices, while the lighter represents the delay in our customized platform with the additional overhead introduced by SmartAuth. Event handlers incur the highest incremental overhead, while commands incur the highest proportional overhead (almost double the base case).

network latency in cloud-based IoT platforms.

Next, we test the degree to which SmartAuth policies to mitigate overprivilege and block third-party remote access impact backward compatibility with existing SmartApps. As with our performance analysis, we test the worst case of users blocking all unexpected and dangerous behaviors and all remote access. We again test patched apps on Samsung's online simulator environment. We trigger events at least five times and insert debug messages into the modified apps' source code to observe apps' behaviors while they gather data from the cloud or when events have been triggered. To evaluate backward compatibility, we both observe app behaviors and analyze the debugging messages. For tests related to overprivilege policy, we focused on the 30 apps that exhibit undisclosed overprivilege. For the interested reader, these 30 apps are listed in the Appendix as Table 2. These apps either request capabilities not mentioned in their descriptions (unexpected capability), or even worse, request capabilities that could do harm (e.g., unlocking the door). For example, the SMART SECURITY app presents a description: "alerts you when there are intruders but not when you just got up for a glass of water in the middle of the night." After scanning the source file, this app requests access to `motionSensor`, `contactSensor`, and `alarm` capabilities, satisfying the description, but also requests sensitive commands including turning on/off a switch, which is not mentioned in its description. Therefore, we mark this access as an unexpected behavior. For the remaining 150 apps, we further patch them to comply with our policy enforcement mechanism. Specifically, apps with coarse capability overprivilege and device-app binding overprivilege are also

constrained to ensure least privilege.

In our compatibility tests, **none** of the 180 apps crash after patching, even with overprivilege security rules enforced. Even after they are patched to remove overprivilege, the 180 apps behave the same as their original versions. In other words, patching does not break the functionalities claimed in the app's description.

We further test how apps function if we block all third-party remote access, an extreme case where the user denies all such requests. Of the 180 apps, only six apps suffer from a loss of valid functionality. For example, VINLI HOME CONNECT allows remote services to control IoT devices, and this functionality breaks entirely when we block remote access. We believe such examples will continue to be rather rare, especially when users are given clear information and useful options to configure the app's security policy. In addition, we envision the possibility of a cloud-based reference monitor that could check run-time remote access and filter out dangerous access, but such a design is beyond the scope of this work.

6.4 Limitations

Although SmartAuth advances user-centered IoT app authorization, our design has limitations. A malicious developer could use custom-defined methods and property names mirroring SmartThings commands and attributes to fool the program analysis. A future version of SmartAuth could better recognize this technique. Our static analysis tool is based on Groovy AST transformation. If handled correctly, the tool can detect obfuscated logic (which cannot evade AST transformation), and obfuscated dynamic variable/function names can be handled with define-use analysis citenielson2015principles. Furthermore, a malicious developer could craft app descriptions for which SmartAuth mistakenly extracts a malicious behavior from the description, even when humans would not perceive such a behavior. Future work could focus on recognizing such adversarial descriptions. External services like IFTTT could be the future work for our project. Our approach can be applied if we know the control flow information from IFTTT. External devices, if they are approved by Samsung, will be included in the capability system and covered by our project.

In addition, dynamic method invocation from remote servers is a threat that requires future investigation. However, this is less of a concern because Samsung bans dynamic method execution through code review [51].

Our user studies also have important limitations. While we did not draw attention to this fact, particularly attentive participants might have recognized that SmartAuth was a novel interface. This recognition might have biased participants to be complimentary of an interface they assumed was being tested, as well as to pay particular attention to the interface in the absence of habituation effects. Furthermore, users will not always have a choice between an overprivileged app and a less privileged variant, and it is an open question whether users might still install an overprivileged app if it is the only option. We have one assumption that users will read the app description when they decide to install apps. However, we did not run a formal user study to verify the assumption. We did observe in the lab study that most users payed attention to the app description, but it would be better to verify the assumption formally. Currently, the smart home market is still at an early stage, and most of the users are with a technical background. Many participants in our lab study have good technical background, which is representative for the current users. However, when the smart home systems get much more popular, our participants might not be representative for future users.

7 RELATED WORK

We next compare our work with previous research.

7.1 Mobile Permission Studies

Many researchers have studied permission systems for mobile devices. While some insights apply in both domains, the unique features of IoT platforms introduce new security and privacy challenges. Most similarly to SmartAuth, the Whyper system identifies Android permissions that might be used from the app's description [42]. The researchers do an extensive analysis of app descriptions and match them with permissions, but they do not evaluate the real security behaviors from the code of the applications. Even for analyzing descriptions, SmartAuth is fundamentally different because Android permissions and APIs have very specific privacy implications. In contrast, reasoning about implications in the IoT is much more context-sensitive, necessitating our further use of NLP. Zhang et al. instead analyze Android apps using static analysis, generating descriptions for the security behaviors in the applications [57]. These descriptions are helpful for users to understand the app's behavior. However, users are burdened with reading the long logs and still need to use the original Android interface to authorize. In contrast, we remove many overprivilege cases automatically and both design and test a new scheme that minimizes user burden.

Many approaches build on this prior work. AutoCog compares descriptions with permissions requested [43]. AsDroid analyzes the text in the user interface and the current behavior to see whether it is a stealthy behavior [29]. Appcontext analyzes context that triggers security behaviors and compares the context among apps to differentiate benign and malicious apps [56]. Other researchers compare app behaviors to app descriptions by clustering applications with similar functionality and finding apps that use uncommon APIs [26]. Besides mobile permissions, researchers also look into the privacy policies to

identify privacy inconsistency of the code and the privacy policy [58].

Another line of work studies users' mental model about permissions, focusing on users' perceived risks [17, 18]. For example, Egelman et al. investigate user's perceptions of sensitive data stored on their phones, including banking information and home address [13]. However, our study about users' mental model about IoT permission makes new contribution because the perceptions and requirements in IoT platfroms are different from mobile platforms. Many researchers have sought to improve mobile permissions. For example, Liu et al. propose privacy profiles to ease user burden [37]. Almuhimedi et al. propose information visualization to improve user awareness of risks [3], Harbach et al. suggest using personal examples to better explain permission requests [28], and Tan et al. suggest using developer-specified explanations for understanding [52]. Researchers have also provided general guidelines for designing permission systems [16, 44]. Users' perceptions of mobile permissions and IoT permissions share some characteristics. For instance, Wijesekera et al. observe through a field study that mobile apps sometimes violate contextual integrity by accessing unexpected resources [55]. However, due to the differing privacy and security implications for IoT platforms, SmartAuth further rethinks the design of authorization systems.

7.2 IoT Security and Privacy

IoT security and privacy is an emerging area. Previous research has largely focused on identifying security and privacy vulnerabilities. Naveed et al. discuss the security binding problems of smart devices that are external to the mobile phone [40]. Fernandes et al. run a black-box analysis of Samsung SmartThings, pinpointing the overprivilege problem [19]. We instead reconceptualize overprivilege to be more practical and user-centered. To enhance security and privacy goals in IoT and home automation systems, FlowFence [20] uses information flow control and explicitly isolates sensitive data inside sandboxes. This approach requires intensive modification to SmartApps, and the enforcement is done on Android instead of a real smart home hub. Jia et al. [30] gather information before a sensitive action is executed, and ask for user approval through frequent run-time prompts. However, in-context prompts cannot satisfy the real-time automation of IoT apps (e.g., users need to be awake to approve a permission when an emergency happens). Users will likely become habituated to approving these prompts, mistakenly approving unexpected behaviors. Furthermore, the information they provide to users is directly dumped from code, whereas we generate natural language to improve communication with users. BLE-Guardian [15] controls who can discover, scan, and connect to an IoT interface. CIDS [10] designs an anomaly-based intrusion detection system to detect in-vehicle attacks by measuring fingerprints from deployed ECUs based on clock behaviors. Sivaraman et al. [45] propose managing IoT devices through software-defined networking (SDN) based on day-to-day activities.

Beyond framework or architecture solutions, enhancing the security of smart devices is also a common countermeasure against attacks from remote or near field communication surfaces. For example, SEDA [6] proposed their attestation protocol for embedded devices. Through software attestation and showing states gathered from booting sequences, SEDA can construct a security model for swarm attestation. Similar approaches to ensure IoT or smart device integrity [2, 7, 9, 10] complement our system.

Some researchers have also examined IoT privacy from a usability perspective. For example, Egelman et al. suggest using crowdsourcing to improve IoT devices' privacy indicators [14]. Further, Ur et al. investigate parents' and teens' perspectives on smart home privacy [53] and Demiris et al. study seniors' privacy perspectives for smart homes [12]. Kim et al. study the challenges in access control management in smarthome and present usable access control policies [34, 35]. In contrast, beside understanding user's mental model about smarthome privacy. we design a new usable IoT authorization scheme.

8 Acknowledgement

We would like to thank Tadayoshi Kohno, Soteris Demetriou, and the anonymous reviewers for their invaluable feedback. The project is supported in part by NSF CNS-1223477, 1223495, 1527141 and 1618493, and ARO W911NF1610127.

9 Conclusion

In this paper, we have identified the fundamental gap between how users expect an IoT app to perform and what really takes place. We rethink the notion of authorization in IoT platforms and propose an automated and usable solution called SmartAuth to bridge the gap. SmartAuth automatically collects security-relevant information from an IoT app's code and description, and generates a user-friendly authorization interface. Through manual verification and in-lab human subject studies, we demonstrate that SmartAuth can enable users to make more well-informed authorization decisions for IoT apps compared to the current approach.

References

[1] SimpleNLG. SimpleNLG. https://github.com/simplenlg/simplenlg, 2016.

[2] Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., and Tsudik, G. C-FLAT: Control-FLow ATtestation for Embedded Systems Software. *arXiv preprint arXiv:1605.07763* (2016).

[3] Almuhimedi, H., Schaub, F., Sadeh, N., Adjerid, I., Acquisti, A., Gluck, J., Cranor, L. F., and Agarwal, Y.

Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *33rd Annual ACM Conference on Human Factors in Computing Systems* (2015), ACM, pp. 787–796.

[4] AMAZON, INC. Smart Home Skill API Reference. https://developer.amazon.com/public/solutions/alexa/alexa-skills-kit/docs/smart-home-skill-api-reference, 2017.

[5] APPLE, INC. Apple HomeKit. http://www.apple.com/ios/home/, 2016.

[6] ASOKAN, N., BRASSER, F., IBRAHIM, A., SADEGHI, A.-R., SCHUNTER, M., TSUDIK, G., AND WACHSMANN, C. SEDA: Scalable embedded device attestation. In *22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 964–975.

[7] BABAR, S., STANGO, A., PRASAD, N., SEN, J., AND PRASAD, R. Proposed embedded security framework for internet of things (IoT). In *2nd Int'l Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)* (2011), IEEE, pp. 1–5.

[8] BIRNBAUM, Z. On a use of the mann-whitney statistic. In *3rd Berkeley symposium on mathematical statistics and probability* (1956).

[9] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. TyTAN: Tiny trust anchor for tiny devices. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), IEEE, pp. 1–6.

[10] CHO, K.-T., AND SHIN, K. G. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium* (2016), USENIX Association.

[11] CHRISTOPHER D. MANNING. Dropping common terms: stop words. http://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html, 2016.

[12] DEMIRIS, G. Privacy and social implications of distinct sensing approaches to implementing smart homes for older adults. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2009), IEEE, pp. 4311–4314.

[13] EGELMAN, S., JAIN, S., PORTNOFF, R. S., LIAO, K., CONSOLVO, S., AND WAGNER, D. Are you ready to lock? In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[14] EGELMAN, S., KANNAVARA, R., AND CHOW, R. Is this thing on?: Crowdsourcing privacy indicators for ubiquitous sensing platforms. In *33rd Annual ACM Conference on Human Factors in Computing Systems* (2015), ACM, pp. 1669–1678.

[15] FAWAZ, K., KIM, K.-H., AND SHIN, K. G. Protecting privacy of BLE device users. In *25th USENIX Security Symposium* (2016), USENIX Association.

[16] FELT, A. P., EGELMAN, S., FINIFTER, M., AKHAWE, D., AND WAGNER, D. How to ask for permission. In *HotSec* (2012).

[17] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *2nd ACM workshop on Security and privacy in smartphones and mobile devices* (2012), ACM, pp. 33–44.

[18] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *8th Symposium on Usable Privacy and Security (SOUPS'12)* (2012).

[19] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *36th IEEE Symposium on Security and Privacy* (2016).

[20] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium* (2016).

[21] GABRILOVICH, E., AND MARKOVITCH, S. Computing semantic relatedness using Wikipedia-based explicit semantic analysis. In *International Joint Conference on Artificial Intelligence* (2007), pp. 1606–1611.

[22] GOLDBERG, Y., AND LEVY, O. Word2Vec explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* (2014).

[23] GOOGLE, INC. Brillo. https://developers.google.com/brillo/, 2016.

[24] GOOGLE, INC. Google News Vectors. https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM, 2016.

[25] GOOGLE, INC. Weave. https://developers.google.com/weave/, 2016.

[26] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *36th International Conference on Software Engineering* (2014), ACM, pp. 1025–1035.

[27] GROUP, S. N. The Stanford Parser: A statistical parser. http://nlp.stanford.edu/software/lex-parser.shtml, 2002.

[28] HARBACH, M., HETTIG, M., WEBER, S., AND SMITH, M. Using personal examples to improve risk communication for security and privacy decisions. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'14)* (2014).

[29] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *36th International Conference on Software Engineering* (2014), ACM, pp. 1036–1046.

[30] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., AND PRAKASH, A. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Network and Distributed System Security Symposium (NDSS'17)* (2017).

[31] JUNIPER RESEARCH. Smart Home Revenues to reach $100 Billion by 2020. https://www.juniperresearch.com/press/press-releases/smart-home-revenues-to-reach-\protect \T1\ textdollar 100-billion-by-2020, 2008–2016.

[32] KELLEY, P. G., CONSOLVO, S., CRANOR, L. F., JUNG, J., SADEH, N., AND WETHERALL, D. A conundrum of permissions: Installing applications on an Android smartphone. In *International Conference on Financial Cryptography and Data Security (FC'12)* (2012).

[33] KELLEY, P. G., CRANOR, L. F., AND SADEH, N. Privacy as part of the app decision-making process. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'13)* (2013).

[34] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Challenges in access right assignment for secure home networks. In *HotSec* (2010).

[35] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Access right assignment mechanisms for secure home networks. *Journal of Communications and Networks* (2011).

[36] LEVY, H. M. *Capability-based computer systems*. Digital Press, 2014.

[37] LIU, B., LIN, J., AND SADEH, N. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *23rd international conference on World wide web* (2014), ACM, pp. 201–212.

[38] MANNING, C. D., SURDEANU, M., BAUER, J., FINKEL, J. R., BETHARD, S., AND MCCLOSKY, D. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL (System Demonstrations)* (2014), pp. 55–60.

[39] MILLER, G. A. WordNet: a lexical database for english. *Communications of the ACM 38*, 11 (1995), 39–41.

[40] NAVEED, M., ZHOU, X.-Y., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *Network and Distributed System Security Symposium (NDSS'14)* (2014).

[41] NISSENBAUM, H. Privacy as contextual integrity. *Wash. L. Rev. 79* (2004), 119.

[42] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. Whyper: Towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium (USENIX Security 13)* (2013), pp. 527–542.

[43] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *21st ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1354–1365.

[44] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B. J., AND WANG, H. J. User-driven access control, Aug. 11 2015. US Patent 9,106,650.

[45] SIVARAMAN, V., GHARAKHEILI, H. H., VISHWANATH, A., BORELI, R., AND MEHANI, O. Network-level security and privacy control for smart-home IoT devices. In *11th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)* (2015), IEEE, pp. 163–167.

[46] SMARTAUTH. SmartAuth. http://mews.sv.cmu.edu/research/smartauth, 2017.

[47] SMARTTHINGS, INC. SmartThings Marketplace. https://support.smartthings.com/hc/en-us/articles/205379924-Marketplace, 2016.

[48] SMARTTHINGS, INC. SmartThings Public. https://github.com/SmartThingsCommunity/SmartThingsPublic, 2016.

[49] SMARTTHINGS, INC. Samsung SmartThings. https://www.smartthings.com/, 2017.

[50] SMARTTHINGS, INC. Smart Home Cloud API. http://developer.samsung.com/smart-home, 2017.

[51] SMARTTHINGS, INC. SmartThings Code Review Guidelines. http://docs.smartthings.com/en/latest/code-review-guidelines.html, 2017.

[52] TAN, J., NGUYEN, K., THEODORIDES, M., NEGRÓN-ARROYO, H., THOMPSON, C., EGELMAN, S., AND WAGNER, D. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'14)* (2014).

[53] UR, B., JUNG, J., AND SCHECHTER, S. Intruders versus intrusiveness: teens' and parents' perspectives on home-entryway surveillance. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (2014), ACM, pp. 129–139.

[54] VERA LTD. Vera: Smarter Home Control. http://getvera.com/, 2008–2016.

[55] WIJESEKERA, P., BAOKAR, A., HOSSEINI, A., EGELMAN, S., WAGNER, D., AND BEZNOSOV, K. Android permissions remystified: A field study on contextual integrity. In *USENIX Security Symposium* (2015).

[56] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *37th IEEE/ACM International Conference on Software Engineering* (2015), pp. 303–313.

[57] ZHANG, M., DUAN, Y., FENG, Q., AND YIN, H. Towards automatic generation of security-centric descriptions for android apps. In *22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 518–529.

[58] ZIMMECK, S., WANG, Z., ZOU, L., IYENGAR, R., LIU, B., SCHAUB, F., WILSON, S., SADEH, N., BELLOVIN, S., AND REIDENBERG, J. Automated analysis of privacy requirements for mobile apps. In *Network and Distributed System Security Symposium (NDSS'2017)* (2017).

A SMARTAPP PATCHING

Our patching script is written in roughly 600 lines of python code to modify the original Groovy source file by the following steps. A toy example for a patched app TURN IT ON FOR 5 MINUTES is given in Listing 2.

Listing 2: We provide a code snippet for patched IoT app TURN IT ON FOR 5 MINUTES. Text in blue indicates statements that need to be patched, and text in red indicates either new code instrumented by the script or replaced with our wrapped functions. The `appSetting` section added after the definition block is used for OAuth configuration.

```
definition(
  name: "Turn It On For 5 Minutes",
  namespace: "smartthings",
  author: "SmartThings",
  description: "When a SmartSense Multi is
      opened, a switch will be
    turned on, and then turned off after 5
        minutes.",
  category: "Safety Security",
   ... \\
) {
  appSetting "client_idFPS" // used to config
      app identifier for OAuth.
  appSetting "client_secretFPS" // used to
      config app secret for OAuth.
  appSetting "http_serverFPS" // we configure
      cloud server url here.
}
...
mappings { // act as end-points for policy
    enforcement module to deliver event data
  path("/post_event") {
    action:  [
      POST: "handleEventFromProxyServer"
    ]
  }
}
preferences {
  section("When it opens...") {
    input "contact1",
        "capability.contactSensor"
  }
  section("Turn on a switch for 5
      minutes..."){
    input "switch1", "capability.switch"
  }
}
def installed() {
  log.debug "Installed with settings:
      \${settings}"
```

```
    subscribe(contact1, "contact.open",
        contactOpenHandler)
    subscribeToServer(contact1, "contact",
        "open", contactOpenHandler)
}
def updated(settings) {
    log.debug "Updated with settings:
        \${settings}"
    unsubscribe()
    unsubscribeToServer()
    subscribe(contact1, "contact.open",
        contactOpenHandler)
    subscribeToServer(contact1, "contact",
        "open", contactOpenHandler)
}
def contactOpenHandler(evt) {
    switch1.on()
    sendCommandToProxyServer(switch1, "on", NULL,
        NULL, NULL, NULL)
    def fiveMinuteDelay = 60 * 5
    runIn(fiveMinuteDelay, turnOffSwitch)
}
def turnOffSwitch() {
    switch1.off()
    sendCommandToProxyServer(switch1, "off",
        NULL, NULL, NULL, NULL)
}
...
```

To enable authorization in the for policy enforcement module, the script automatically inserts dynamic pages and prepares a URL for the patched app to enable an OAuth authentication flow at install time. The SmartThings platform provides a trigger for an OAuth authorization flow via the URL containing an app identifier and its cloud-generated app secret. When the user navigates to the URL, they will be redirected to the SmartThings login page to enter credentials and receive an authorization token for later use.

The script next scans all devices on the SmartThings capability list[4] by parsing all input labels from the `preferences` section and its corresponding child pages, e.g., `mainPage` page section. The script builds an internal structure called DL, maintaining a pair of information (input label, device capability), for later code substitution for command or attribute statements.

The script then parses event handler subscription and unsubscription statements by scanning the keywords. A subscription statement consists of its input label, associated attributes, and the corresponding event handler function. For instance, `subscribe(motionSensors, "motion.active", motionActive)` means the app subscribes an event handler for status activity of input `motionSensors` which has motion capability, and assign function `motionActive` as callback handler. Therefore, our patching engine replaces this statement with an internal function `subscribeToServer()` to send all corresponding parameters to the policy enforcement module along with its app identifier. The module will determine whether this subscription is allowed depending on user's rules. If successful, the module will forward the event data to the registered SmartApp. Unsubscription is much easier to implement, namely by removing all subscriptions registered on the policy enforcement module.

The last step is to search all statements for possible command issuing or attribute retrieving associated with those device labels collected above. For example, the structure DL may contain an input device called **switch1** which has a `switch` capability. When the script parses a statement containing the label switch1, e.g., `switch1.on()`, the script catches the function call `on()` and checks against a capability structure defined based on the list of capabilities and their associated functions and attributes[5]. Once the script confirms the call or attribute, it replaces the original statement with the internal API call `sendCommandToProxyServer()` by sending the request to the policy enforcement module with its app identifier, device label (switch1), command label (`on()`) and any corresponding parameters.

After patching, each Groovy source file will contain around 128 new lines to provide endpoint interfaces for the policy enforcement module.

B SmartAuth Working Example

Here we use one example to show how SmartAuth works. The Flasher is an app that claimed to flash a set of lights to notify user when motion, open/close event, or switch event is detected. However, besides subscribing to motion sensor, contact sensor, and switch, the app also subscribes to the presence sensor and the acceleration sensor. To bridge the gap between what the users think the app do and the app's real behaviors, we generate the security policy from the code and from the description. We display the verified capabilities according to their functionality, and notify users about the unexpected behaviors, similar to Figure 4. On the interface, we further classify the unexpected actions into "unexpected" and "dangerous", according to the user perception measured through our crowd-sourcing result. We present the security policy and unexpected/dangerous behaviors in a usable authorization interface. After getting the response from the users, we enforce the policy so that the app only gets what it needed for the functionality and what the user understand and would like the app to access.

C Apps used in the Lab Study

We show the participants five group of apps in the SmartAuth and SmartThing interface, as shown in Table 1.Interfaces used in the experiments can be found at [46].

[4] http://docs.smartthings.com/en/latest/capabilities-reference.html

[5] http://docs.smartthings.com/en/latest/capabilities-reference.html

Table 1: Apps in the lab study

App ID	App Name	Description	Overprivileged? If so, Behavior Type
1A	SMART HUMIDIFIER	Turn on/off humidifier based on relative humidity from a sensor.	NO
1B	HUMIDITY ALERT	Notify me when the humidity rises above or falls below the given threshold. It will turn on a switch when it rises above the first threshold and off when it falls below the second threshold.	YES, Lock (Dangerous)
2A	VIRTUAL THERMOSTAT	Control a space heater or window air conditioner in conjunction with any temperature sensor, like a SmartSense Multi.	YES, Motion Sensor (Dangerous)
2B	SMART HEATER	Turn on/off the heater based on the temperature.	NO
3A	LIGHTS OFF	Turn lights off when no motion and presence is detected for a set period of time.	NO
3B	DARKEN BEHIND ME	Turn your lights off after a period of no motion being observed.	YES, Temperature Sensor (Unexpected)
4A	FLASH A NOTICE	When something happens (open/close, switch on/off, motion detected), flash lights to indicate.	NO
4B	THE FLASHER	Flashes a set of lights in response to motion, an open/close event, or a switch.	YES, Presence Sensor (Unexpected)
5A	LEFT IT OPEN	Turn lights off when no motion and presence is detected for a set period of time.	YES, Power Meter (Unexpected)
5B	SMART WINDOW	Compares two temperatures - indoor vs outdoor, - then sends an alert if windows are open (or closed). If you don't use an external temperature device, your zipcode will be used instead.	NO

D EXAMPLE SURVEY QUESTIONS

We list a few representative survey questions.

D.1 Example questions in the Mturk study

1. What factors will you consider when making decision of whether to install a third party app or not? And please indicate how much you care on each factor that you will consider. {Strongly care, care, neither care or not care, not care, Strongly not care}
   () The source / author of the app
   () The popularity of the app
   () The functionality of the app
   () The privacy aspect of the app
   () The smarthome capabilities that the app request
   () The relation of capability requests to the app's functionality
   () Others:
2. Third-party apps can access devices in the smart home after they are installed. Please rate the risk levels of the different behaviors to access devices. {Very sensitive, sensitive, a bit sensitive, not sensitive}
   () Unlock your door
   () Lock your door
   () Read the input of your door lock
   () Read the battery level
   () Read your motion sensor
   () Control your water pump
   () Turn on/off your light
   () Adjust the level of your light
3. Similar to smarthome capabilities, Android or iOS also provide permissions to third-party apps to control the access to resources in the mobile phone such as your location and contact book. Which one do you think is more sensitive?
   A) Smarthome capabilties are more sensitive
   B) Android or iOS permissions are more sensitive
   C) I think they are the same
   D) I don't know
4. Please explain your reasons for the last question:

D.2 Example questions in the in-lab study

Please choose how much you agree with the following statements. {Strongly disagree, disagree, neither agree or disagree, agree, strongly agree}.

1. I feel that the app description explains thoroughly why the app can access and control these sensors and devices.
2. I feel confident to make a decision whether or not to install the app after reading the description.
3. It is difficult to find information from the description.

E CROWDSOURCING FOR UNEXPECTED BEHAVIOR SENSITIVITY

We evaluate how sensitive the unexpected behaviors are by combining expert reviews and crowdsourcing together. In particular, we have two security experts and 100 Mturkers to look into the apps' unexpected behaviors and evaluate how sensitive the unexpected behavior is given the context of the app. We asked the participants to classify whether these unexpected behaviors are dangerous or not(dangerous is counted as 1, and not dangerous is counted as 0). From the expert and Mturk responses, we assign each security expert a weight of 0.25, and each Mturker a weight of 0.005. If the weighted sum is over 0.5, we consider the behavior as dangerous.

Table 2: Compatibility test results among 30 SmartApps exhibiting undisclosed overprivilege, meaning they contain capabilities for functionality not disclosed in the app description. Of these undisclosed overprivilege cases, we refer to the low-risk cases as unexpected capabilities and the high-risk cases as dangerous capabilities. Note that the risk levels are crowd-sourced via online surveys. We remove the access to all the unexpected and dangerous capability to test whether the apps can still perform correctly.

App	Unexpected Capability	Dangerous Capability	Compatible
ALFRED WORKFLOW	switch	lock	Not if block remote access
BRIGHT WHEN DARK AND/OR BRIGHT AFTER SUNSET	switchLevel		Yes
CAMERA POWER SCHEDULER	switch		Yes
CURLING IRON		motionSensor	Yes
FORGIVING SECURITY	contactSensor, switch	alarm, motionSensor	Yes
GOOD NIGHT		switch	Yes
JENKINS NOTIFIER	colorControl	switch	Yes
NOTIFY ME WHEN	button, contactSensor, accelerationSensor, presenceSensor, smokeDetector, waterSensor	motionSensor, switch	Yes
PHOTO BURST WHEN	accelerationSensor, contactSensor	imageCapture, motionSensor, switch, presenceSensor	Yes
PREMPOINT		imageCapture, switch, lock, garageDoorControl	Yes
RISE AND SHINE		motionSensor	Yes
SAFE WATCH	contactSensor, accelerationSensor, threeAxis, temperatureMeasurement	motionSensor, presenceSensor	Yes
SEND HAM BRIDGE COMMAND WHEN	contactSensor, accelerationSensor, switch, waterSensor, smokeDetector	motionSensor, presenceSensor	Yes
SIMPLE CONTROL	switch, lock, thermostat, doorControl, colorControl, musicPlayer, switchLevel	lock, doorControl	Not if block remote access
SMART LIGHT TIMER	contactSensor	motionSensor	Yes
SMART SECURITY		switch	Yes
SMART WINDOWS	contactSensor		Yes
SMARTBLOCK NOTIFIER		switch	Yes
SPEAKER CONTROL	contactSensor, accelerationSensor, switch, waterSensor, button	motionSensor, presenceSensor	Yes
SPEAKER MOOD MUSIC	contactSensor, accelerationSensor, button, waterSensor,musicPlayer	motionSensor, presenceSensor, switch	Yes
SPRAYER CONTROLLER 2		switch	Yes
SPRUCE SCHEDULER	contactSensor		Yes
TALKING ALARM CLOCK	switchLevel, temperatureMeasurement, thermostat, relativeHumidityMeasurement		Yes
THE FLASHER		presenceSensor	Yes
TURN IT ON FOR 5 MINUTES	contactSensor		Yes
UNDEAD EARLY WARNING	contactSensor	switch	Yes
VINLI HOME CONNECT		switch,lock	Not if block remote access
VIRTUAL THERMOSTAT		motionSensor	Yes
WEATHER WINDOWS	contactSensor		Yes
WHOLE HOUSE FAN	contactSensor		Yes

F APPS WITH UNDISCLOSED OVERPRIVILEGE

Table 2 tabulates the 30 apps that exhibit undisclosed overprivilege. These apps either request unexpected capabilities not mentioned in their descriptions or dangerous capabilities that could cause harm.