A Model-Driven-Engineering Approach for Detecting Privilege Escalation in IoT Systems

ATHEER ABU ZAID, Ryerson University, Canada

MANAR H. ALALFI, Ryerson University, Canada

ALI MIRI, Ryerson University, Canada

Software vulnerabilities in access control models can represent a serious threat in a system. In fact, OWASP lists broken access control as number 5 in severity among the top 10 vulnerabilities. In this paper, we study the permission model of an emerging Smart-Home platform, SmartThings, and explore an approach that detects privilege escalation in its permission model. Our approach is based on Model Driven Engineering (MDE) in addition to static analysis. This approach allows for better coverage of privilege escalation detection than static analysis alone, and takes advantage of analyzing free-form text that carries extra permissions details. Our experimental results demonstrate a very high accuracy for detecting over-privilege vulnerabilities in IoT applications.

1 INTRODUCTION

IoT - term coined in 1999 by Kevin Ashton - emerged to describe when the Internet is used to connect objects as end users, rather than just people [12]. IoT has a variety of definitions by different groups in the academia and the industry [12, 14, 25]. A common understanding of IoT is the interconnection of uniquely identifiable, ubiquitous, sensing/actuating capable, programmable and self configurable things over the internet. These things share data and services without the need for human involvement [16]. IoT is now widely adopted in many important domains, such as in healthcare systems, smart cities, smart homes and autonomous cars [22] [15]. Employing IoT systems in cities has proved to be useful, such as in improving transportation by gathering data and analyzing it [22]. It is also important in smart homes to automate everyday tasks. Despite popularity of IoT systems in practice, their design and implementation maturity is still in early stages. In particular, IoT systems suffer from many security vulnerabilities, some of which have already been addressed in other types of systems. The issues of interest to us in this paper are those related to access

Authors' addresses: Atheer Abu Zaid, Ryerson University, Toronto, Canada, aabuzaid@ryerson.ca; Manar H. Alalfi, Ryerson University, Toronto, Canada, manar.alalfi@ryerson.ca; Ali Miri, Ryerson University, Toronto, Canada, Ali.Miri@ryerson.ca.

control, which is an important component of IoT systems. Access control models provide rules that administer and constrain how objects access and interact with each other. We will discuss access control models in more details later.

An important application of IoT is smart homes. A smart home is a system in a residence that allows the household to monitor and control home devices and appliances [4, 5]. In [27], *Zhou et al.* provide a concise description of the current trend in smart home platforms. They demonstrate that cloud-based smart home platforms typically have three components: the *cloud backend*, the *physical IoT devices* and the *mobile app*. The cloud is responsible for identity management, executing applications and home automation. Apps are executed in the cloud to ensure that the remote commands are sent by authenticated users. IoT devices have sensors for monitoring and actuators for executing commands. The devices can either be cloud-connected devices or hub-connected devices. The last component, the mobile app, provides the user with an interface that connects IoT devices to the smart home to specify the desired home automation.

Some of the most popular smart home platforms are: Samsung's SmartThings, Apple HomeKit, Vera Control's Vera3, Google's Weave/Brillo, AllSeen Alliance's AllJoyn, Amazon Alexa, Google Assistant and IFTTT [5, 9].

Programmable Smart home platforms have emerged over the last years and have provided users with broad benefits. These platforms provide third-party developers with the opportunity to develop apps for compatible devices. Unfortunately, this flexibility that apps can be developed by third-party developers introduced new risks and attacks into these platforms. In this paper, the platform of interest is Samsung's SmartThings. Similar to other programmable platforms, it is cloud-based and provides a programming framework for third-party developers [9]. Furthermore, SmartThings shares two other common features with other platforms: authorization and authentication through the capability model, and the support of event-driven processing. Thus, analysis done on this platform can be further extended to other similar working platforms.

2 MOTIVATION

One of the fundamental software security principles is the least privilege principle [23]. This principle indicates that an entity should be provided with the least privileges it needs to perform its function. The entity should not be able to access a resource if not given permission explicitly. Evading this principle might result in misuse of resources and other security misuse implications.

The vulnerability we are interested in this paper is over-privilege in IoT applications, which maps to the "Broken Access Control" in OWASP's top 10 IoT vulnerability list, that is listed as number 5 in importance. OWASP top 10 vulnerabilities is a valuable resource for researchers and businesses to consider when auditing systems for security [17].

Design flaws in the SmartThings platform have resulted in apps acquiring access to more resources than required [10]. Another issue in the design of the SmartThings permission model is the ability to gain elevated access to unauthorized resources in the devices. This is called privilege escalation. Detecting privilege escalation issues in software before publishing it is of great importance for protecting the users. Threats in SmartThings that allow for extra access to resources motivate us to improve the detection of vulnerabilities concerning the access control model.

In this paper, we present an MDE approach that provides a comprehensive solution to detect the different over-privilege access scenarios in SmartThings applications. To address these cases of over-privilege our paper presents the following contributions:

- We designed and developed an approach and a tool that employs MDE in addition to static analysis to translate multiple sources of information in the app into permission rules. Unlike other existing techniques [8, 9, 24, 26], our approach is comprehensive in detection and coverage of privilege escalation cases in SmartThings.
- We designed a meta-model that is used to extract permission rules in IoT apps. Once the permission rules are extracted in the form of Prolog facts, Prolog is used to check if the app conforms to the permission meta-model.
- An experiment that attempts to address the following research questions:
  (1) RQ1: What is the performance of our approach? How scalable is our analysis when applied to a large dataset of SmartThings apps? Can our automated analysis report the 3 cases of over-privilege in SmartThings apps?
  (2) RQ2: How does privilege escalation detection performed by ChYP [1] compare to the MDE approach presented in this paper?
  (3) RQ3: How effective is our tool in detecting privilege escalation in SmartThings apps when evaluated for precision and recall?

3 BACKGROUND

The SmartThings platform is a cloud-based system that provides a programming framework for third-party developers. Apps written by the developers are executed in a sand-boxed environment on the cloud to restrict the allowed operations to third-party developers. A central device in the smart-home, *the hub*, connects the apps on the cloud with the smart devices. *SmartApps* in this framework are applications that can communicate with smart devices. These communications are based on a permission model, also called the capability model.

Listing 1 demonstrates the structure of the SmartThings SmartApp. A SmartApp has four general sections: *Definition*, *Preferences*, *Predefined Callbacks* and *Event Handlers* [19]. The *Definition* section includes meta-data about the app that appear in the mobile app UI, including the description of the app. Capabilities and input requests are provided in the *Preferences* section. In listing 1, the app requests two capabilities: motionSensor and lock. Once the app is installed, the user must provide the app with access to compatible devices for each requested capability. Two common *Predefined Callbacks* are *'installed'* and *'updated'*. They are called automatically when the app is installed and updated, respectively. Event subscription is typically set up in those callbacks. It allows the app to listen to events from the device.

```
1  definition
2  ( name: "",
3      namespace: "",
4      author: "",
5      description: "",
6      ...)
7  preferences
8  { section("Select devices")
9      { input "themotion", "capability.motionSensor", title: "Select a motion sensor"
10     input "thelock", "capability.lock", title: "Select a lock" }}
11 def installed() { initialize() }
12 def updated() { unsubscribe() initialize() }
13 def initialize() { subscribe themotion, "motion.active", activeHandler }
14 def activeHandle(evt) { thelock.unlock() }
```

Listing 1. SmartApp Structure

The *Event Handler activeHandler* in Listing 1 unlocks the device when an *'active'* event is triggered. Lastly, commands and attributes are usually invoked in *Event Handlers*, such as the *'unlock'* command being invoked in *activeHandler*. A capability model represents how smart devices are capable of performing functionalities. The capability structure that

this model relies on has two components; *commands* and *attributes*. The over-privilege vulnerability in the SmartThings platform will be discussed further in this chapter. A capability can have multiple commands and attributes to support it. Attributes hold the state of the device in regard to some property, while commands are used to actuate actions on the device. A device usually has more than one capability, such as having a battery, a light, some sensors or any resource that a smart device might need to perform its job. Apps written to interact with these devices have to specify and request which of the supported capabilities they want to access. As long as a device provides all the needed capabilities of an app, the app is compatible to access this device. This is a restriction of the capability model.

On installation of the app, the end user is given the option to choose which of the compatible devices they want to be controlled by this app. Typically, the set of capabilities that the app requested of the devices are not shown to the user. This is done in the code and is not explicitly presented to the user. This may result in users not being able to detect an app being over-privileged.

3.1   Existing Access Control Models in Smart Homes

A survey by Fernandes *et al.* [10] on access control in IoT devices, and in particular smart home programming frameworks, illustrated that granularity in access control varies from all-or-nothing to very fine grained. Apple's HomeKit framework requires access at the home-level granularity (all-or-nothing), meaning iOS apps will always be over-privileged. AllJoyn framework offers access control on a fine-grained level (they call it member level), but recommends it to be at the interface level (both physical devices and software apps in AllJoyn can expose interfaces that have members). Analysis of the SmartThings framework shows that apps are given access at the level of the device (not very fine grained) as a result of a flaw in the design. This leaves apps over-privileged and increases the risk of device exploitation. Over-privilege appears to be an issue whether it was by design or as a result of a flaw in the design. Good security strategies to follow should include:

(1) **The less unnecessary access the better.**
(2) **Informing the user in case of an exploitation can help mitigate the consequences.**

It is easier for an entity to misbehave if it has more access than expected. In case this happens, it is necessary that the user is informed to mitigate the risks. It is worth to mention that mitigation actions may be visible or invisible to users. For example, a visible action could be the unlocking of a door, while an invisible one could be transfer of the information electronically on the network. Zhang *et al.* [26] proposed an intrusion detection system (IDS) to solve this problem in the SmartThings framework. It sniffs the wireless communications in the smart home and raises an alarm when an anomaly is detected during runtime. This system relies on natural language processing (NLP) techniques of an application's description. They achieved a 0.98 true positive rate. Another work by Tian *et al.* attempts to include the user in what actually happens. They analyzed an app's description, code and annotations to provide the user with better permission requests that are more inclusive of what the app actually uses. While this is useful to avoid over-privileged apps, the user could still benefit from tracking the app performance at runtime. In this paper, we consider access control models that allow access at the level of the device, such as in SmartThings Smart-home platform.

Using the code fragment Listing 1 as an example, we illustrate over-privilege in SmartApps in further details. When the user permits a SmartApp to control a capability of a device, this device can be accessed through its declared id in the input statement. For example, if we look at the preferences section in Listing 1, when the user selects a device that supports the Motion Sensor capability, this device will be bound to the id *themotion*. Using this id, commands and attributes of the capability can be accessed. However, some devices might support more than one capability. For

example, the Phillips Hue Bulb supports both the Switch capability and the ColorControl capability [21]. Due to the flaw in the design, the id is actually a reference to the device as a whole. As a result, a SmartApp that is privileged to access a device for a specific capability, can access commands and attributes of all the capabilities in the bound Device Handler. This over-privilege in SmartApps is caused by coarse SmartApp-SmartDevice binding [9].

Another over-privilege case in the SmartThings framework is caused by coarse-grained capabilities [9]. A coarse-grained capability provides the SmartApp with all the commands and attributes of the permitted capability, even if it only needs a subset of them. It is considered an over-privilege when the user permits the app to access a capability for the specified functionality, without knowing an app can control and access the entire capability.

Over-privilege in SmartApps occurs in two main scenarios: Acquiring more capabilities than permitted, and acquiring more commands and attributes than needed. For each scenario, we will provide a basic example of how over-privilege might occur in SmartApps. For fast proof-of-concept attacks, we used the web-based SmartThings IDE and simulated devices instead of physical ones.

3.1.1 Over-Privilege Caused by Coarse SmartApp-SmartDevice Binding. Once the SmartApp is authorized to access a specific capability of a device, the SmartApp can access all the capabilities implemented by the device handler (SmartDevice). To simulate this scenario, we need a device handler that supports at least two capabilities. For this example we used three capabilities: the Presence Sensor, the Lock and the Battery capabilities. We will write a SmartApp that unlocks the lock on arrival:

(1) In the preferences method, request the Presence Sensor and the Battery capabilities:

```
1 preferences
2 { section("Select devices")
3   { input "thepresence", "capability.presenceSensor", title: "Select a presence sensor"
4     input "thebattery", "capability.battery", title: "Select a battery" }}
```

Listing 2. Over-privileged SmartApp, Case 1-step1

(2) Subscribe to the presence event handler in both installed() and updated() methods.
(3) In the presence event handler, assuming *thebattery* device supports the Lock capability as well, trigger the *unlock* command and check the status of the lock through the *currentLock* attribute:

```
1 def presenceHandler(evt)
2 { if (thebattery.currentLock == "locked")
3   { thebattery.unlock()
4       log.debug "Lock status: $thebattery.currentLock" }}
```

Listing 3. Over-privileged SmartApp, Case 1-step2

We run this SmartApp in the simulator with a device handler that implements the three capabilities, which simulates a device with those capabilities. We observed that *thebattery* device successfully accessed the Lock capability of the device. This example proves that the *thebattery* id is a reference to the whole device that is bound to the Battery capability,

rather than to the Battery capability only. Thus, the SmartApp is over-privileged and has access to all the capabilities that this device supports.

3.1.2   Over-Privilege Caused by Coarse-Grained Capabilities. A SmartApp always acquires all commands and attributes implemented by a permitted capability, even if it only needs a subset of them. This could be dangerous because different commands often have different levels of risk if the SmartApp is exploited [9]. Typically, an app specifies its functionality in the description section, which often implies how it will use the capabilities.

In Listing 4, the description of the application is: "Unlock the front door on arrival". As users, we understand that the action to be taken is the unlocking of the door, and it needs to happen after the person arrives to the place. When the person arrives, this indicates a change in the state of the presence.

```
1 definition(
2  name: "UnlockDoorApp",
3  namespace: "",
4  author: "",
5  description: "Unlock the front door on my arrival",
6  category: "",
7  iconUrl: "",
8  iconX2Url: ""
9 )
```

Listing 4. SmartApp's Description in Free-Form Text

Listing 5 shows that the app requested two capabilities: the lock capability and the presenceSensor capability and their titles are shown to the user. The user can then deduce which commands and attributes are needed for this application to perform its functions. Specifically, the *presence* attribute and the *unlock* command only.

```
1 preferences
2 { section("Select devices")
3    { input "thepresence", "capability.presenceSensor", title: "Select a presence sensor"
4      input "thelock", "capability.lock", title:  "Select a lock"
5    }
6 }
```

Listing 5. SmartApp's Requested Capabilities

If we look into the permission standards table in Listing 6, we notice that the lock capability has more commands and attributes than just the unlock command.

```
1 light -> off
2 light -> on
3 lockOnly -> lock
4 lock -> lock
5 lock -> unlock
6 ...
7 powerSource -> powerSource
8 presenceSensor -> presence
```

Listing 6. Segment from the Capabilities Table

We want to make sure that the application is not taking advantage and using non implied resources. To do this we search for the accessed commands and attributes in the application. In Listing 7, we have an example of the device *thelock* actuating the *lock* command to lock the door.

Table 1. Causes of Over-Privilege in SmartThings

Case1: Coarse SmartApp-SmartDevice Binding	Case2: Semantically Over-Privileged	Case3: Coarse-Grained Capabilities
- Request set of capabilities A	- Infer from free-form text the need of set of capabilities A	- Infer the use of a set of commands & attributes A, whose owner capability is properly requested
- Access set of capabilities B, such that $A \not\supseteq B$	- Request set of capabilities B, such that $A \not\supseteq B$	- Access set of commands & attributes B, such that $A \not\supseteq B$

After running this SmartApp in the simulator, we find that the app did gain more commands than needed and could access them successfully. This proves that the app is over-privileged and this is exactly what we want to detect when analyzing the applications.

```
1  def presenceHandler(evt)
2  { if (thelock.currentLock == "locked")
3    { thelock.unlock()
4       thelock.lock() }}
```

Listing 7. In the presence event handler, after unlocking the lock, try to lock the device again

To analyze an app for this case of over-privilege, we examine the inconsistencies between two sections of the app: the app's description and the app's code that accesses the device. Figure 1 illustrates the three types of over-privilege in the SmartThings framework based on inconsistencies between an app's components. Case 1 arises upon the usage of resources that are not requested by the app. While case 2 is the result of not mentioning the need for capabilities that are requested by the app. Case 3 results when the app describes the use of some of the commands and attributes of a requested capability but actually accesses the commands and attributes that are not needed according to the description.

Causes of over-privilege are summarized in Table 1. In all three cases, if set A is not a super-set of set B (it is not inclusive of it), then it is considered to be an over-privileged app. In other words, set A has to include everything that is in set B to be considered a benign app. For example, in case 1, the app requests set of capabilities A, but some (or all) of the capabilities it accesses is from outside set A.
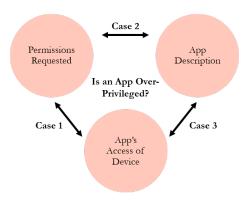

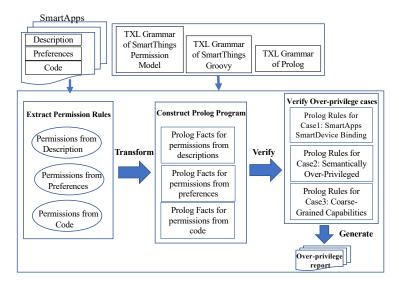Fig. 1. App Components Used in Analysis

Fig. 2. Approach Architecture

4 APPROACH OVERVIEW

Figure 2 lays the three main components of our MDE approach used to verify over-privilege access in SmartThings Apps. This approach takes as input the SmartApp, TXL grammars of SmartThings permission model and SmartThings Groovy and a Prolog program containing the rules of over-privilege in SmartThings SmartApps. The first stage recovers the permission rules from the SmartThings app from the description, the preferences and the code. The recovered permission model conforms to a meta-model we constructed for SmartThings, and presented in Figure 3.

Stage two generates the Prolog program and creates the executable from the Prolog main program which includes: the Prolog rules for each over-privilege case and the Prolog facts extracted from the SmartApp. The last step is to run the Prolog program to produce the final report with the over-privilege results. Each stage will be further discussed in the following sections.

4.1 Extracting Permission Rules

The process of extracting the permission model works by taking the smartapp as an input and extracting all the permission rules (if there is more than one). For example, in an IFTTT app there is one permission rule per app, but in a SmartThings app, it can be more complicated and include more than one [5]. In order to automate the process we first need to define a meta-model that describes the permission model for the SmartThing platform.

4.1.1 Smart-Home Permission Meta-Model Definition. In this paper, we have developed a simplified meta-model that abstracts the permission model used in SmartThigs platform. Figure 3 demonstrates the smart-home permission meta-model, which we adopted and revised from IoTCOM and SmartHomeML [5, 8]. The Permission Rule in the meta-model (Figure 3) maps to the Behavioural Rule in IoTCOM. We adjusted the triggers in their model to be 1 trigger in ours, as each trigger leads to a specific set of conditions and actions. As for the conditions, we adjusted it to be zero or more, as not all rules have conditions. At last, if there exists no actions, then the permission rule is not taken into consideration.

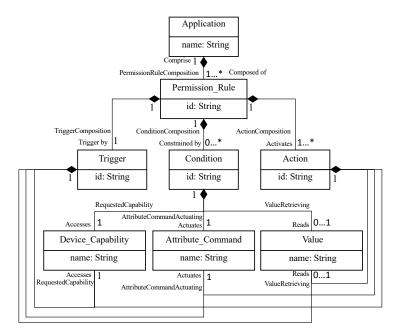We adopted from SmartHomeML the following:

Fig. 3. Smart-Home Permission Meta-Model

- The "Query Action" translates to using the "attribute/command" and "value" with one of the rule components. In SmartThings it would typically either be a trigger or a condition. This is because one cannot change the value directly, but rather by actuating the command.
- The "Control Action" translates to using the "attribute/command", which in this case means command. In SmartThings it would typically be used in an action, not a trigger or a condition.
- The "Skill (capability)" is the "device (capability)" in our model.

This concludes that our adjusted Smart-Home meta-model has the following structure: A permission rule consists of a trigger, a group of conditions (if any), and at least one action. Then, each trigger, condition and action has a device/capability, a command or an attribute and a value. In SmartThings apps, a trigger is the entry point to the application. It triggers the associated event handler. This pair of trigger and event handler is declared in a subscribe statement, the example in Listing 8 shows an app subscribed to the *'present'* event that will run the *'presence'* event handler.

```
1 def initialize() {
2  subscribe(driver, "presence.present", presence)}
```

Listing 8. Trigger and Event Handler Subscription

The conditions are translated from the If statements in the event handlers, see example in Listing 9 where the condition *If the event value is 'present'* that will be translated to a condition in the permission rule.

```
1 def presence(evt) {
2  if (evt.value == "present"){
3      ... }}
```

Listing 9. If Statement to be Translated into a Condition

The actions are anything else that is a direct access of the devices and resources. For example, in Listing 10 the app actuates the *'on'* command in the switch device.

```
1 def turnLightsOn() {
2    switches?.on()}
```

Listing 10. Non If Statement in Event Handler to be Translated into an Action

4.1.2 Semantic Annotation and Preparation for Analysis. The first challenge we encountered is how the approach will parse the natural language description presents in the app under analysis into the permission rule meta-model described in Figure 3. For that, we used semantic annotation to properly annotate free-form text according to the permission meta-model we defined in the previous section. The annotation relies on the extensive coverage of the grammar of words and phrases that map to objects of the meta-model [13].

To automate the process of semantic annotation, we use TXL [7], a hybrid functional/rule-based language with deep pattern matching that serves well for software structural analysis and source transformation. To program in TXL, a context-free grammar of the input is defined (whether it's a programming language or something else). After that, transformation rules are provided that describe how the input will be transformed. The rules must follow the grammar to allow for a correct transformation.

Generating TXL Grammar of SmartThings Permission Model

An essential part of the grammar needed for the analysis is the permission model of SmartThings. The permission model specifies the supported capabilities in SmartThings, in addition to the commands and attributes of each capability. A list of SmartThings capabilities can be found online. [20] Listing 11 Provides a snippet of the grammar built from the permission model taken from SmartThings documentation. We define the devices (capabilities) as in the Listing, as well as the commands and attributes and their values. Then, for each entity listed in the definitions, such as *'accelerationSensor'* in the Listing, we define it according to the grammar inference process. (Section 4.1.2)

```
1  % Definition of all capabilities
2  define device
3     [accelerationSensor] | [alarm] | [audioNotification] | [battery] | [beacon] |
4     [bulb] | [button] | [carbonDioxideMeasurement] | [colorControl] | [colorTemperature] |
5     [configuration] | [consumable] | [contactSensor] | [doorControl] | [energyMeter] |
6     [estimatedTimeOfArrival] | [garageDoorControl] | [holdableButton] | [illuminanceMeasurement] |
7     [imageCapture] | [indicator] | [infraredLevel] | [light] | [lockOnly] | [lock] |
8     [mediaController] | [momentary] | [motionSensor] | [musicPlayer] | [notification] | [outlet] |
9     [pHMeasurement] | [polling] | [powerMeter] | [powerSource] | [presenceSensor] | [refresh] | ...
10 end define
11
12 % Definition of the 'accelerationSensor' capability
13 define accelerationSensor
14 ...
15 end define
16
17 % Definition of all commands
18 define command
19 ...
20 end define
21
22 % Definition of all attributes
23 define attribute
24 ...
25 end define
```

```
26
27  % Definition of all values
28  define value
29  ...
30  end define
```

Listing 11. Grammar Snippet of SmartThings Permission Model

Grammar Inference

To build the needed grammar that enriches the annotations process, We opted for an approach inspired by grammar inference [18]. It is an important stage in building the grammar for this approach as it helps in capturing the semantics of the natural language statements in the apps. To perform grammar inference, we used a public dataset. [1] We started with a subset of the apps, focusing on the apps natural language statements and manually tagging each word with a suitable entity mapped to the SmartThings permission model (if possible). Based on the tagging, we inferred the grammar terminal and non-terminal constructs and their relationships. We applied the first version of the grammar to the natural language statements we manually tagged and that to ensure the grammar is capable of producing the accurate annotation to the statements once they are parsed in the form of XML representation. If the annotation was successful, we repeat the same process on a new set of natural language statements from apps under analysis. The process of grammar inference is manual and tedious process as it requires expert's knowledge with mapping NL statements words to the SmartApps permission-system meta-model. The process needed multiple iterations until we were confident that the grammar produced is capable of automatically annotating all natural language statements used in the dataset under analysis. Listing 12 provides an example of the grammar inference process and the resulting grammar. Line 1 presents the app's description in natural language: "Turn your lights on when motion is detected." We tag each word in the description, if possible. Lines 5-10 provide each word with the entity we tagged it with, some of them have multiple tagging. For example, line 16 defines the word "when" as an indicator for a trigger. Lines 14-38 explain how the tagging fits into the grammar we built.

```
1   App description: "Turn your lights on when motion is detected."
2
3   The manual tagging (the grammar inference process):
4
5   Turn > switch capability
6   lights > light capability
7   on > on attribute, command or value
8   when > indicator for trigger
9   motion > motion attribute or motionSensor capability
10  detected > active value
11
12  Snippet of the resulted grammar:
13
14  keys
15      % indicators for trigger
16      when whenever case
17      ...
18  end keys
19
20  define switch
21   'switch | 'turn
22  end define
23
```

---

[1] https://github.com/SmartThingsOverprivilege/smartthings_overprivilege_dataset

```
24 define light
25  'light | 'lights
26 end define
27
28 define motionSensor
29  'motionSensor | 'motion
30 end define
31
32 define motionAttribute
33  'motion
34 end define
35
36 define activeValue
37  'active | 'detected
38 end define
```

Listing 12. Grammar Inference Example

4.2   Constructing Prolog Facts for SmartThings Permission-rules

In order to successfully verify that the SmartApps do not have any type of over-privilege scenarios, the extracted permission artifacts needs to be transformed into a formal verification engine. Using Prolog provides a flexible, scalable and concise definition of verification goals. It enables the definition of domain-specific rules for validation. Prolog has a backtracking mechanism to test several variants of program flows for the verification goals. For our implementation, we used the established Prolog system SWI-Prolog . We have used Prolog in our previous work for access control verification of web applications [3] and it proved to be efficient and scalable. IT was also recently used for embedded systems verification [11]. In this meta-model, a SmartApp is composed of one or more permission rules. Each permission rule has the following components: a trigger, zero or more conditions and at least one action.

For our analysis, we first need to define the mappings between the meta-model we defined in previous section and the corresponding Prolog facts. Each permission rule component accesses a capability and either a command or attribute and its value.

Table 2 presents the mapping between the meta-model described in Figure 3 and their translation to Prolog facts. For example, the *'Permission_Rule()'* Prolog fact consists of the *'source'* (which is either from the code or description), the app's name and the id of the permission rule. Multiple permission rules can be extracted from the same app. Then each rule can have a trigger, conditions and actions, and each is linked through the RuleId. Each entity in the meta-model has a Prolog fact that defines its attributes. Operation *'RequestedCapability'* also has a Prolog fact to record the list of capabilities the app requested.

4.2.1   Extracting Permission-Rule Facts from Description. This is the first main stage in our approach. The inputs to this stage are:

- SmartApp to be analyzed
- TXL grammar of SmartThings permission rule
- TXL grammar of SmartThings groovy
- Extracted data-types of the devices in the SmartApp

TXL parses and annotates the descriptions in the SmartApp into the permission rule meta-model. Upon annotation, each permission rule extracted is translated into the corresponding Prolog facts from table 2. Facts are then saved in a file for later use. Listing 13 is an example of a description of a SmartApp.

Table 2. Permission Meta-Model Mapping to Prolog Facts

Prolog Fact	Description
Application(Source, AppName):	Represents the name of the app being analyzed. Source of the fact is either 'description' or 'code'.
Permission_Rule(Source, AppName, RuleId):	Represents a permission rule in an app.
Trigger(Source, RuleId, TriggerId):	Represents a trigger in a permission rule.
Action(Source, RuleId, ActionId):	Represents an action in a permission rule.
Condition(Source, RuleId, ConditionId):	Represents a condition in a permission rule.
Attribute_Command(Source, "Trigger/Condition/Action"+Id, AttributeCommandName):	Represents an attribute/command that belongs to a trigger, condition or an action in a permission rule.
Device_Capability(Source, "Trigger/Condition/Action"+Id, DeviceCapabilityName):	Represents a device/capability that belongs to a trigger, condition or an action in a permission rule.
Value(Source, "Trigger/Condition/Action"+Id, ValueName):	Represents a value that belongs to a trigger, condition or an action in a permission rule.
TriggerComposition(Source, AppName, RuleId, TriggerId, DeviceCapabilityName, AttributeCommandName, ValueName):	Represents the details of a trigger of a permission rule.
ConditionComposition(Source, AppName, RuleId, ConditionId, DeviceCapabilityName, AttributeCommandName, ValueName):	Represents the details of a condition of a permission rule.
ActionComposition(Source, AppName, RuleId, ActionId, DeviceCapabilityName, AttributeCommandName, ValueName):	Represents the details of an action of a permission rule.
Capability(DeviceCapabilityName):	Represents the name of a device/capability.
AttributeCommandOf(DeviceCapabilityName, AttributeCommandName):	Represents the name of an attribute/command that belongs to a device/capability.
ValueOf(AttributeCommandName, ValueName):	Represents the name of a value that belongs to an attribute/command.
RequestedCapability(AppName, Capability):	Represents a capability requested by an app.

```
1 description:"Turn your lights on when motion is detected.",
```

Listing 13. Description of a SmartApp

```
1  <program>
2  <repeat_description>
3   <description><repeat_permission_rule>
4    <permission_rule>
5     <repeat_rule_component>
6      <rule_component><action><repeat_component_element>
7       <component_element><device><switch>Turn</ switch></device></component_element>
8       <component_element><word><id>your</id></word></component_element>
9       <component_element><device><colorControl>lights</ colorControl></device></component_element>
10      <component_element><attribute><onAttribute>on</ onAttribute></attribute></component_element>
11      <empty/>
12     </repeat_component_element>
13    </action>
14   </rule_component>
15   <rule_component><trigger>
16    <trigger_indicator>when</trigger_indicator>
17    <repeat_component_element>
18     <component_element><device><motionSensor>motion</ motionSensor></device></component_element>
19     <component_element><word><id>is</id></word></component_element>
20     <component_element><value><detectedValue>detected</ detectedValue></value></component_element>
21 ...
```

Listing 14. Intermediate Representation of Extracting Permission Rules From Description Using XML

This description is parsed by TXL into the intermediate representation of the grammar using XML, as in Listing 14, then TXL will again translate it to Prolog facts (see listing 15). The intermediate representation in Listing 14 shows the mapping of each word in the description to its match in the grammar of SmartThings meta-model. For example, the word *'Turn'* is annotated as the device (capability) *'switch'* and the word *motion* is annotated as the device (capability) *'motionSensor'*. We mainly care about *triggerComposition*, *conditionComposition*, *actionComposition* and *device_capability* from these facts, but the rest could be useful in future work, in this paper we only mention the main facts that are actually used in the Prolog rules.

```
1  application(desc, AppName).
2
3  permission_rule(desc, AppName, rule1).
4  % Action facts
5  action(desc, rule1, action1).
6  device_capability(desc, action1, switch).
7  attribute_command(desc, action1, on).
8  value(desc, action1, on).
9  actionComposition(desc, AppName, rule1, action1, switch, on, on).
10 % Trigger facts
11 trigger(desc, rule1, trigger1).
12 device_capability(desc, trigger1, motionSensor).
13 attribute_command(desc, trigger1, motion).
14 value(desc, trigger1, detected).
15 triggerComposition(desc, AppName, rule1, trigger1, motionSensor, motion, detected).
```

Listing 15. Prolog Facts Extracted from Description

4.2.2 Extracting Permission-Rule Facts from Preferences. The input to this stage is the SmartApps and the grammars as in the first stage above. TXL analyzes the SmartApp for capabilities requested in the preferences section. Listing 16 is an example of a SmartApp requesting the switch capability. This input statement is translated to a Prolog fact (as in listing 17), and saved to the same facts file used previously.

```
1  preferences {
2   section("When I reach home, turn on the lights.") {
3    input "switches", "capability.switch", multiple: true }
4  }
```

Listing 16. Capabilities Requested by SmartApp

```
1  requestedCapability(AppName, switch).
```

Listing 17. Prolog Fact for Case 2

4.2.3 Extracting Permission-Rule Facts from Code. The third stage is to analyze the app code to extract facts for cases 1 and 3. The analysis extracts the permission rules from the code. We start with the trigger, which is extracted from the subscribe methods in the app. Each subscribe method declares the event handler (or method) that will be executed upon the trigger. The event handler is analyzed to extract the conditions (any if statements), while all other statements are considered actions.

In listing 18, the trigger is the change to the motion sensor, specifically when "motion" attribute changes its value to "active". This triggers the execution of the event handler *motionActiveHandler*. The resulted trigger fact is shown in listing 19. After that, the tool analyzes the code of the event handler for conditions and actions. The resulted two facts would typically be as in listing 20. In the case that we extracted either an attribute/command or a value, and this

Table 3. Variables Definitions

Variable	Purpose
prologExecutableFileName	The name of the output Prolog program that will be used to run the model checker
prologMainFile	The main Prolog file used to initiate the Prolog checker
prologRulesFile	The Prolog file containing the three main rules for detection of over-privilege cases
prologFactsFile	The Prolog file containing the Prolog facts extracted from the description, preferences and the code

extracted word is considered both an attribute/command and a value in the TXL grammar, we fill it in both placed in the Prolog fact. This is because when we analyze the description of the app, it is hard to infer if this word should be filled in the attribute/command or the value in the Prolog fact.

```
1  def installed()
2  { subscribe(motion1, "motion.active", motionActiveHandler)}
3
4  def motionActiveHandler(evt) {
5      if(switch1.currentValue("switch") == "off"){
6          switch1.on()}}
```

Listing 18. Trigger and Associated Event Handler

```
1  triggerComposition(code, AppName, rule1, trigger1, motionSensor, motion, active).
```

Listing 19. Trigger Fact

```
1  conditionComposition(code, AppName, rule1, condition1, switch, off, off).
2  actionComposition(code, AppName, rule1, action1, switch, on, on).
```

Listing 20. Condition and Action Facts

4.3 Verifying Over-privilege cases

We create the Prolog model checker from within the TXL main program of the tool. Line 1 in Listing 21 is the command used for the creation of the Prolog executable. Our Prolog model checker takes three sources of information to carry out the analysis. Table 3 defines the use and contents of each variable/file. The model checker first takes the main program needed to specify which rules to run. It also needs the Prolog rules file containing the three Prolog rules for checking the three cases of over-privilege explained in the following sections. Lastly, we provide the Prolog facts file containing all the facts we extracted from the app's description, preferences and code. After we have created the Prolog model checker, we pass the program's file name to the 'run' command to run the analysis and produce the analysis report.

```
1      construct createPrologExe [stringlit]
2          _ [+ "swipl --goal=main --stand_alone=true -q -o "] [+ prologExecutableFileName] [+ " -c "] [+ prologMainFile]
                [+ prologRulesFile] [+ prologFactsFile]
3      construct runPrologProgram [stringlit]
4          _ [+ "run "] [+ prologExecutableFileName] [+ ".exe"]
5      construct workHere [stringlit]
6          _ [system createPrologExe]
7            [system runPrologExe]
```

Listing 21. Command to Create the Prolog Model Checker

4.3.1   Detection of Case 1: Over-Privilege Caused by Coarse SmartApp-SmartDevice Binding.

Prolog Rules for case 1:

This case of over-privilege occurs when there is an inconsistency between the *Permissions Requested* and *App's Access of Device*. To check for occurrences of over-privilege case 1, we need to inspect if any of the devices approved for this app and has more than one capability, was misused to access a capability not intended for this app. A query to the Prolog rule needs to satisfy the following:

- a resource (a command or a value of an attribute), has been used in a trigger, a condition or an action
- this resource is not a command of the capability in this trigger, condition or action
- this resource is not a value of an attribute of the capability in this trigger, condition or action
- this resource belongs to another capability in the SmartThings capability model
- the capability and the resource are both non empty in the facts

A complete Prolog rule for case 1 can be found in Appendix A.

Prolog Facts for case 1

Given the facts in the listing below, an over-privilege of case 1 must be reported. The fact on line 13 states that the device *accelerationSensor* was used to actuate the *on* command, which is a command of another capability and not of *accelerationSensor*'s.

```
1  /* SmartThings Capability model facts */
2  capability(accelerationSensor).
3  attributeCommandOf(accelerationSensor,acceleration).
4  valueOf(acceleration,active).
5  valueOf(acceleration,inactive).
6  capability(switch).
7  attributeCommandOf(switch,switch).
8  attributeCommandOf(switch,off).
9  attributeCommandOf(switch,on).
10 valueOf(switch,off).
11 valueOf(switch,on).
12 /* Fact used to check case 1 */
13 triggerComposition(code,AppName,ruleNumber,trigger1,accelerationSensor,on,na).
```

Listing 22. Sample Facts Relevant to Over-Privilege Case 1

4.3.2   Detection of Case 2: Over-Privilege Caused Semantically.

Prolog Rules for case 2

This case of over-privilege occurs when there is an inconsistency between the *App Description* and *Permissions Requested*. To detect over-privilege case 2, we need to inspect if the application declared or indicated in the description what capabilities it will be accessing. A query to the Prolog rule needs to satisfy the following:

- the app have requested a certain capability (fact name: *requestedCapability*)
- there is no mention of this capability in the description facts (fact name: *device_capability*)
- capability has a name and is not empty (na)

This listing below is the resulting Prolog rule to detect this case of over-privilege:

```
1 overprivilegedCase2(case2,AppName,Capability):-
2    requestedCapability(AppName,Capability),
3    not(device_capability(desc,_,Capability)),
4    not(Capability=na).
```

Listing 23. Prolog Rules to Detect Case 2 of Over-Privilege in SmartApps

Prolog Facts for case 2

Given the facts in the listing below, an over-privilege of case 2 must be reported. As non of the capabilities declared in the description facts is the *switch* capability.

```
1 device_capability(desc,_,presenceSensor).
2 device_capability(desc,_,door).
3 requestedCapability(AppName, presenceSensor).
4 requestedCapability(AppName, door).
5 requestedCapability(AppName, switch).
```

Listing 24. Sample Facts Relevant to Over-Privilege Case 2

4.3.3 Detection of Case 3: Over-Privilege Caused by Coarse-Grained Capabilities.

Prolog Rules for case 3

This case of over-privilege occurs when there is an inconsistency between the *App Description* and *App's Access of Device*. It can happen as a result of coarse-grained capabilities. It means that a permitted capability is fully available to the app to access, with all its commands and attributes. This is considered an over-privilege because risk associated with giving access to different commands and attributes is not on the same level [9]. For example, giving access to turn off the oven and have it be misused might be annoying, while misusing the turn on operation is very risky.

A query to the Prolog rule needs to satisfy one of the following:

- The app performs an action in the code that is not mentioned in the description.
- There exists a combination of a trigger and an action in code, that is not mentioned in the description with any combination of the capability, attribute/command and the value. For example, suppose that there exists in the code a trigger (if presence changes), turn on lights (action), but, the description only declared the lights to turn off if you leave the house. There would be inconsistency between the facts in the description and the code, leading to reporting an over-privilege of case 3 (sample facts for this example are shown in listing 26).
- Same case as the one above, but with differences in the combination of a condition and an action, rather than a trigger and an action. An example for this would be if we found in the code a condition that if the presence changes, it would turn the lights on, while in the description, it was declared that the lights should be turned on only after 10 AM.

This listing below shows part of the resulting Prolog rule to detect this case of over-privilege:

```
1 overprivilegedCase3(case3,AppName,RuleIdCode,TriggerIdCode,ConditionIdCode,ActionIdCode,Capability,AttributeCommand,
       Value):-
2    /*The whole action is missing in description*/
3    actionNotInDesc(AppName,RuleIdCode,ActionIdCode,Capability,AttributeCommand,Value);
4 actionNotInDesc(AppName,RuleIdCode,ActionIdCode,Capability,AttributeCommand,Value):-
5    actionComposition(code,AppName,RuleIdCode,ActionIdCode,Capability,AttributeCommand,Value),
6    not(actionComposition(desc,AppName,_,_,Capability,AttributeCommand,Value)),
7    not(threeNAs(Capability,AttributeCommand,Value)),
8    not(actionTwoNAs(AppName,Capability,AttributeCommand,Value)).
9 %* Capability,
```

```
10  threeNAs(Capability,AttributeCommand,Value):-
11      Capability=na,
12      AttributeCommand=na,
13      Value=na.
14  /* Use Case:
15      Desc: "Lock the front door"
16      Code: "Unlocks the door" */
17  actionTwoNAs(AppName,Capability,AttributeCommand,Value):-
18      AttributeCommand = na,
19      Value = na,
20      actionComposition(desc,AppName,_,_,Capability,_,_).
```

Listing 25. Sample Prolog Rules to Detect Case 3 of Over-Privilege in SmartApps

Prolog Facts for case 3

Given the facts in the listing below, an over-privilege of case 3 must be reported. As there is an inconsistency between the facts of the permission rules in the description with the ones in the code.

```
1  /* Facts in description */
2  permission_rule(desc, AppName, ruleNumber).
3  triggerComposition(desc, AppName, ruleNumber, trigger1, presenceSensor, presence, notpresent).
4  actionComposition(desc, AppName, ruleNumber, action1, switch, off, off).
5  /* Facts in code */
6  permission_rule(code, AppName, ruleNumber).
7  triggerComposition(code, AppName, ruleNumber, trigger1, presenceSensor, presence, na).
8  actionComposition(code, AppName, ruleNumber, action1, switch, on, on).
```

Listing 26. Sample Facts Relevant to Over-Privilege Case 3

5 EVALUATION

In this section, we evaluate our MDE approach and tool designed to detect the three cases of over-privilege. The implementation makes use of the TXL and Prolog languages. TXL grammars are needed for this tool implementation, including grammars of: SmartThings groovy, SmartThings permission model, Prolog facts and an XML grammar from the TXL project [6] as a helper grammar in the transformation. To evaluate our approach, we obtained a dataset from publicly available sources, these include: SmartThings marketplace, SmartThings community and SmartThings forums. Those apps are developed by third-party developers and their source code is publicly available, this makes the dataset a suitable target for the evaluation. We also used the dataset provided by Zhang *et al.* [26] to evaluate the detection of case 3. We refer to this dataset as "HoMonit" in this evaluation. We evaluate the effectiveness of our approach by answering the following research questions:

(1) RQ1: What is the performance of our approach? How scalable is our analysis when applied to a large dataset of SmartThings apps? Can our automated analysis report the 3 cases of over-privilege in SmartThings apps? (Section 5.1)

(2) RQ2: How does privilege escalation detection performed by ChYP [1] compare to the MDE approach presented in this paper? (Section5.2)

(3) RQ3: How effective is our tool in detecting privilege escalation in SmartThings apps when evaluated for precision and recall? (Section 5.3)

Table 4. MDE-ChYP results on publicly available datasets

Dataset	Total Apps	Total Cases Reported	Case 1	Case 2	Case 3
Homonit	19	183	4	8	171
Forums	50	678	23	138	517
Marketplace	19	317	26	16	275
Community	142	1290	31	146	1113
Total	230	2468	84	308	2076

Fig. 4. Distribution of Over-Privilege Cases Across the Dataset
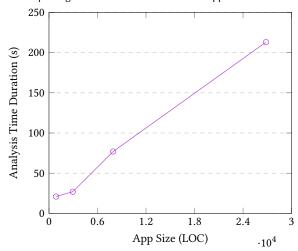

5.1 Automated Analysis Results

In this section we address RQ1. First, we run the tool on the publicly available dataset. We then observe the execution of the tool on the dataset whether it completed successfully and examine its coverage in reporting the three over-privilege cases. Table 4 details the statistical over-privilege results obtained from running our tool on the dataset. The tool ran successfully and completed all stages of the analysis. A total of 230 apps were analysed and over 2000 occurrences of over-privilege were reported across the dataset from cases 1, 2 and 3. Figure 4. illustrates the distribution of over-privilege cases in each of the dataset classifications derived from Table 4. In all four classifications, case 3 dominated the reported over-privilege occurrences. This is reasonable because it is the easiest case to fall into. In case 3, the capability is correctly requested and it is not used to access commands from another capability. What happens in case 3 is that the developer ambiguously declares how or when the commands and attributes of the capability will be accessed (whether accidentally or intentionally). So, even if for example the developer declared that the *'lock'* and *'unlock'* commands of the lock capability will be actuated on arrival and take off but did not specify which command is paired with which event, this might be suspicious to the user and should be marked as over-privilege.

5.1.1 Performance of the Tool. To answer RQ1 in regard to performance of the tool, we measure the analysis time for each dataset classification separately, as in Table 5. We ran the analysis on a PC with an Intel Core i7 2.2 GHz CPU processor and 20 GB RAM. The data we gather to calculate the performance are the lines of code (LOC) for the

Table 5. Performance Statistics

Dataset	Total Apps	Total LOC	Total Prolog Facts	Analysis Time Duration
HoMonit	19	872	1439	0m21s
Forums	50	7939	7338	1m17s
Marketplace	19	2943	2559	0m27s
Commnunity	142	26,836	19,360	3m33s

Fig. 5. Analysis time of privilege escalation detection in SmartApps in relation to size of the dataset

SmartThings apps, the number of Prolog facts generated by each app and time spent in analysis. Table 5 details the performance data reported after running the evaluation experiment. The largest bundle of apps, the Community dataset including 142 apps with more than 26,000 LOC, took a little over 3 minutes and a half to complete the analysis. Figures 5 and 6 illustrate the time growth of the analysis in regards to the LOC and Prolog facts respectively. Both figures imply that the analysis time seems to be scaling linearly with the LOC of TXL and Prolog facts.

5.2 Comparison between the MDE-based Approach (MDE-ChYP) and the Static Analysis Approach (Chyp)

To answer RQ2, we compare our MDE-based approach with our previous static analysis-based approach, ChYP [2], on two aspects: accuracy and coverage of privilege escalation detection. The static analysis approach analyzes the syntax only and performs well in tracing complex functions and loops. The MDE approach provides a combination of static analysis, text processing, and logic-based reasoning.

For this comparison, we randomly picked 25 apps that reported occurrences of case 1 when we ran the MDE-ChYP on the public datasets in table 4. The list of apps used in this experiments are displayed in table 6. We manually checked the apps for actual occurrences of case 1 and found 41 occurrences across the apps. We ran both tools on this dataset of 25 apps, then we manually validated the results reported by both tools. Through the validation of results from the static
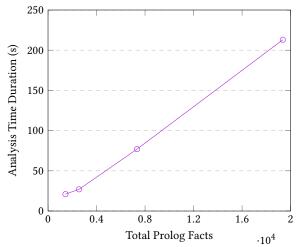
Fig. 6. Analysis time of privilege escalation detection in SmartApps in relation to no. of Prolog facts

analysis based tool, we found 25 true positives and 16 false negatives. While for MDE-ChYP tool, we found 33 true positives, 7 false positives and 8 false negatives. (see Table 6).

Table 7 displays the precision and recall of evaluating both tools on this selected dataset of 25 apps.

Precision evaluates the tool in not returning non-over-privileged cases. While recall evaluates the tool in returning the over-privileged cases if present in an app [13]. Below are the formulas used to calculate the precision and recall [13, 26]. Precision is calculated by dividing the true positives (TP) by the true positives and false positives (FP), while recall is the division of true positives over the true positives and false negatives (FN).

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

ChYP did not report any false positives, but it did report 16 false negatives, all of them vulnerabilities appearing in natural language statements, a source of information that this tool does not utilize but MDE-ChYP does. An example is provided in Listing 27. Line 3 in the listing shows that *'theAlarm'* device id is used to access the alarm capability of the device. The alarm capability in SmartThings permission model has one attribute: the *'alarm'* attribute. Lines 9 and 10 of the listing show *'theAlarm'* id subscribing to changes in the *'contact'* and *'motion'* attributes, which are not part of the alarm capability. The static analysis tool did not catch these vulnerabilities as they appeared in natural language statements.

MDE-ChYP reported 7 false positives due to ambiguities in analysing the code as free-form text. Listing 28 provides an example. This case appears when the variables are named as attributes or commands from other capabilities. Lines 3, 8 and 9 in the listing show the variable named *'alarm'* being matched with the capability *'waterSensor'* and its attribute *'water'*. This confuses MDE-ChYP and reports it as malicious. MDE-ChYP also produced 8 false negatives probably due to implementation issues that need further investigation in future work.

```
1 preferences {
```

Table 6. Detailed results of evaluation comparison between MDE-ChYP and the static analysis tool ChYP

Dataset	App	Case 1 Occurrences	ChYP			MDE-ChYP		
			TP	FP	FN	TP	FP	FN
Homonit	zigbeeFloodAlert	0	0	0	0	0	1	0
Marketplace	color_coordinator	6	6	0	0	6	0	0
	enhanced_auto_lock_door	0	0	0	0	0	1	0
	keep_me_cozy	1	1	0	0	1	0	0
Forums	alarmThing_AlertAll	2	0	0	2	2	0	0
	buffered_event_sender	1	0	0	1	1	0	0
	fireCO2Alarm	5	3	0	2	2	0	3
	garage_switch	2	1	0	1	2	0	0
	groveStreams	1	0	0	1	1	0	0
	initial_state_event_sender	1	0	0	1	1	0	0
	initialstate_smart_app_v1_2_0	1	0	0	1	1	0	0
	unbuffered_event_sender	1	0	0	1	1	0	0
	zwave_indicator_manager	4	4	0	0	4	1	0
Community	flood_alert	0	0	0	0	0	1	0
	garage_door_monitor	1	0	0	1	1	0	0
	initial_state_event_streamer	1	0	0	1	1	0	0
	keep_me_cozy_ii	1	1	0	0	1	0	0
	lock_it_when_i_leave	0	0	0	0	0	1	0
	medicine_management_contact_sensor	3	3	0	0	1	0	2
	medicine_management_temp_motion	3	3	0	0	1	0	2
	ridiculously_automated_garage_door	0	0	0	0	0	2	0
	smartblock_linker	2	0	0	2	2	0	0
	step_notifier	3	3	0	0	2	0	1
	the_big_switch	1	0	0	1	1	0	0
	Thermostats	1	0	0	1	1	0	0
Total	25 Apps	41	25	0	16	33	7	8

```
2    section("Notify me when there is any activity on this alarm:") {
3        input "theAlarm", "capability.alarm", multiple: false, required: true
4    }
5  }
6  ...
7  def initialize() {
8    log.debug "in initialize"
9    subscribe(theAlarm, "contact", contactTriggered)
10   subscribe(theAlarm, "motion", motionTriggered)
11 }
```

Listing 27. Case 1 of Over-Privilege Found in Natural Language Statements

```
1  preferences {
2   section("When there's water detected...") {
3    input "alarm", "capability.waterSensor", title: "Where?"
4   }
5  }
6
7  def installed() {
8   subscribe(alarm, "water.wet", waterWetHandler)
9   subscribe(alarm, "water.dry", waterWetHandler)
```

```
10 }
```

Listing 28. Example of a False Positive Reported by the MDE based tool

The second aspect in this comparison is the tools' coverage in detecting the three cases of over-privilege in Smart-Things apps. Due to the nature of MDE-ChYP tool and approach, it can detect over-privileges resulted from confusions in the semantics. The static analysis based tool can only detect over-privileges exposed from analysing the syntax. This is why the MDE approach can detect cases 1, 2 and 3 of over-privilege in SmartThings apps, while the static analysis approach only detects case 1.

Table 7. Evaluation comparison between the MDE and the static analysis based tools

Tool	Precision	Recall
Static Analysis Tool(ChYP)	100%	60.98%
MDE-ChYP	82.50%	80.49%

5.3 Manual Validation

In section 5.1, we present the approach results when evaluated on the whole dataset. The approach reported over-privilege occurrences of all three cases as shown in Table 4.

In order to quantitatively validate our findings and to compute precision and recall, we need to compare our results with a base tool, however, and since we did not have access to any tool that provides an analysis for the three types of over-privilege, we need to either manually validate all our results or randomly select random samples from the results and compute precision and recall. Validating all the results manually is time consuming, and selecting random subset from the results does not always guarantee the selected sample has all the over-privilege cases we aim to evaluate. We have already observed that from the previous section, 5.2, when we compared the MDE approach with ChYP for case1 of over-privilege. For this reason, we needed to create a benchmark for the evaluation using mutation analysis.

5.3.1 Benchmark. We have selected a sample benign dataset, and we manually confirmed the dataset is benign, then in each app in the dataset, we injected one case of over-privilege. Mutants created for case 1 required actuating commands and attributes that do not belong to the requested capability.

```
1 definition(
2  name: "Big Turn OFF",
3  namespace: "smartthings",
4  author: "SmartThings",
5 description: "Turn your lights off when the SmartApp is tapped",
6    ...)
7 preferences {
8  section("When I touch the app, turn off...") {
9   input "switches", "capability.switch", multiple: true
10 }}
11 def installed()
12 { subscribe(location, changedLocationMode)
13  subscribe(app, appTouch) }
14 ...
15 def appTouch(evt) {
16  log.debug "appTouch: $evt"
17  switches?.off()}
```

Listing 29. App Before Injecting Over-privilege

Listing 29 provides an example of an app with no over-privileges of any case. The app is modified in listing 30 to show an example of an app with over-privilege of case 1. The app used the *'switch'* device to access the command *'siren'*, which belongs to another capability (line 18 of listing 30).

```
 1 definition(
 2    name: "Big Turn OFF",
 3    namespace: "smartthings",
 4    author: "SmartThings",
 5    description: "Turn your lights off when the SmartApp is tapped",
 6    ...)
 7 preferences {
 8  section("When I touch the app, turn off...") {
 9   input "switches", "capability.switch", multiple: true
10  }}
11 def installed()
12 { subscribe(location, changedLocationMode)
13  subscribe(app, appTouch) }
14 ...
15 def appTouch(evt) {
16  log.debug "appTouch: $evt"
17  switches?.off()
18  switches?.siren()}
```

Listing 30. App After Injecting Case 1 of Over-privilege

To create the mutant for case 2, we randomly choose any one of the capabilities that are not mentioned in the description of the app, and we request it by the application.

   The app in listing 29 only described the use of the switch capability, which was requested by the app correctly. To inject case 2 of over-privilege, we find a capability from the SmartThings documentation that is not supposed to be requested by the app based on its description. Then, we request this capability in the app without informing the user in the description. Listing 31 shows some of the capabilities in the SmartThings documentation. The resulting over-privileged app is shown in listing 32, line 11 shows the app requested the capability *accelerationSensor* without mentioning it in the description in line 5.

```
 1 Capabilities: accelerationSensor, alarm, audioNotification, battery, beacon, bulb, button,carbonDioxideMeasurement,
       colorControl, colorTemperature, ...
```

Listing 31. Snippet of SmartThings List of Capabilities

```
 1 definition(
 2  name: "Big Turn OFF",
 3  namespace: "smartthings",
 4  author: "SmartThings",
 5  description: "Turn your lights off when the SmartApp is tapped",
 6    ...)
 7 preferences {
 8  section("When I touch the app, turn off...") {
 9   input "switches", "capability.switch", multiple: true}
10   section("") {
11   input "sensor", "capability.accelerationSensor"
12                    , multiple: true}
13 }
```

Listing 32. App After Injecting Case 2 of Over-privilege

We created 19 mutants for case 2 in 19 apps from the dataset. As for case 3 of over-privilege, we picked apps from the dataset that have requested at least one capability, and described the use of part of it only in the app's description.

Listing 29 shows a compatible app for injecting case 3. This app describes the use of the command "off" from the switch capability. To inject case 3 of over-privilege, we replace the actuating of the "off" command to the "on" command that is part of the switch capability. Listing 33 shows the app after the injection of case 3. We created 20 mutants in 20 apps from the dataset.

```
1  definition(
2   name: "Big Turn OFF",
3   namespace: "smartthings",
4   author: "SmartThings",
5   description: "Turn your lights off when the SmartApp is tapped",
6      ...)
7  preferences {
8   section("When I touch the app, turn off...") {
9    input "switches", "capability.switch", multiple: true
10  }}
11 def installed()
12 { subscribe(location, changedLocationMode)
13  subscribe(app, appTouch) }
14 ...
15 def appTouch(evt) {
16  log.debug "appTouch: $evt"
17     switches?.off()
18 switches?.on()}
```

Listing 33. App After Injecting Case 3 of Over-privilege

For further details, the complete benchmark could be found online, as well as the public datasets. [2]

To answer RQ3, we run the tool on a selected subset of the dataset and the benchmark we created. For case 1 of over-privilege, we are interested in detecting the use of commands and attributes by capabilities that are not in possession of them. We randomly picked 25 over-privileged apps from the public datasets (listed in Table 4) and validated the reports of case 1 of over-privilege, a detailed analysis for case1 was discussed in the previous section. For case 2, we evaluate the tool in reporting the use of extra unneeded capabilities and in the use of the expected capabilities. We achieve this by evaluating the results obtained from running the tool on malicious and benign datasets. For case 3, we are also interested in evaluating the tool on malicious and benign datasets. The use of a command or attribute from the owning capability should only be reported in case it is unnecessary. Finally, to evaluate the accuracy of the tool, we manually check the results obtained from the automated detection and calculate the precision and the recall.

Table 8 presents the statistical validation results for our analysis. Our tool could detect over-privilege vulnerabilities in malicious apps with high precision and recall. The lowest recall percentage appears in analyzing benign apps for case 3, where high numbers of false negatives are being reported.

We investigated the apps evaluated for the reasons of false positives and false negatives and pinpointed some reasons of error and areas for improvements. Fist, our NLP approach could be improved by exploring better tools and techniques in order to better understand the context and in turn produce a better match. NLP pre-processing techniques involving

---

[2]github.com/SmartThingsOverprivilege/smartthings_overprivilege_dataset

Table 8. Evaluation of MDE-ChYP on the mutation dataset

Dataset	Total Apps Evaluated	Case Occurrences in dataset	TP	FP	FN	Precision	Recall
Malicious - Case 1	25	41	33	7	8	82.50%	80.49%
Malicious - Case 2	19	19	19	8	0	92.66%	74.81%
Benign - Case 2	60	0	82	0	34		
Malicious - Case 3	20	20	19	8	1	82.98%	54.93%
Benign - Case 3	20	0	20	0	31		

punctuation is one area that could immediately improve the results. Another possible improvement is expanding the vocabulary in the tool produced using grammar inference.

-Nevertheless, even if we achieve all the desired improvements in the tool, the description could be written poorly and ambiguously that even the best tools could not comprehend the necessary privileges. Developers are strongly advised to write clear and comprehensive descriptions. This is one way we can reduce over-privileges in apps.

Another way of mitigation is designing the access control component in the programming framework to be the most fine-grained, and ask the user for approval as straightforward as it should be. Of course, an obvious mitigation is to resolve design flaws that result in over-privileges, like case 1 in SmartThings.

6 RELATED WORK

Fernandes *et al.* [9] performed the first empirical security analysis of the SmartThings platform, and provided lessons for the design of smart home programming frameworks. They presented in detail the different types of vulnerabilities found in the SmartThings platform, and conducted proof-of-concept attacks by exploiting the design flaws in the platform. This was a starting point for us to understand how over-privilege results in a SmartApp.

Fernandes *et al.* developed a static analysis tool that detects over-privilege in SmartThings, in addition to run-time analysis and manual analysis where the tool fails to complete the analysis. The tool is developed to detect cases 1 & 3 of over-privilege in SmartThings. They used the tool to better understand the extent to which SmartApps are over-privileged. However, they did not evaluate the effectiveness of the developed tool, nor did they provide an easy way to evaluate it by researchers.

HoMonit, a system designed and developed by Zhang *et al.* [26], detects two types of vulnerabilities in the SmartThings platform: over-privilege in SmartApps and event eavesdropping and spoofing. HoMonit first extracts the expected behavior of a SmartApp either by static analysis of open-source SmartApps or by NLP techniques on closed-source SmartApps. Then, it performs side-channel analysis to monitor the size and interval of the encrypted packets. Changes in the sniffed packets between benign and malicious apps indicate a change in the DFA state. From this change, it can be inferred, with high probability, that the app is not behaving as expected.

Zhang *et al.* evaluated HoMonit and received a 0.98 rate of correctly labeling misbehaving SmartApps based on over-privilege access. Although, over-privilege detection in HoMonit only targets case 3 (caused by coarse-grained capabilities) and requires executing the apps in the benign and malicious states to compare between them.

Tian *et al.* [24] presented a technique named *SmartAuth* that provides a new solution to the over-privilege problem in IoT. SmartAuth proposes a user-centric authorization through the generation of new authorization user interfaces based on what the app actually performs. SmartAuth uses NLP and program analysis to analyze an app's description, code and annotations to detect over-privilege in the app.

Table 9. Comparison with Related Work

Paper	What it does	Techniques Employed
Security Analysis of Emerging Smart Home Applications. Fernandes et al. [9]	Over-privilege detection: Cases 1 & 3	Static analysis Runtime analysis Manual analysis
HoMonit. Zhang et al. [26]	Over-privilege detection: Case 3	Static analysis and NLP to extract DFA of expected behaviour. Side channel analysis to extract DFA during runtime
SmartAuth. Tian et al. [24]	Over-privilege detection: Cases 1 & 3	NLP Program analysis
IoTCom. Alhanahnah et al. [5]	Detects threats in IoT at the interaction level between apps	Static analysis, path-sensitive Formal analysis
SmartHomeML. Einarsson et al. [8]	Generate smart-home applications using a DSML	MDE for fast generation of apps
MDE-ChYP (our tool)[3]	Over-privilege detection: Cases 1,2 & 3	Static analysis to detect over-privilege. MDE for extracting expected and actual behaviors of apps

To conduct the analysis, SmartAuth first analyzes the description using NLP and then analyzes the app code using NLP and program analysis. The extracted privileges from the description are compared with the privileges extracted from the app code to decide which privileges are not properly requested by the app. SmartAuth detects cases 1 & 3 of over-privilege, then patches the malicious app at run-time. The unavailability of SmartAuth tool publicly and the dataset used makes it hard to compare the efficiency of SmartAuth with our tool.

Our approach targets the security auditors of the platform, or the third-party app developers. Where access to the source code is available and analysis can be done statically instead of both statically and dynamically as in [26]. Once over-privilege is detected in our solution, the developers can modify the app's source code instead of patching the app by our tool, as in [24].

IoTCom is an approach and a tool developed to detect threats at the interaction level between IoT apps [5]. In this study, they present a smart-home automation model that defines an app as a set of one or more rules. Each rule can have a set of triggers, conditions and actions. This model is used to extract the apps' behaviours using static analysis, after that they perform formal analysis to detect threats in IoT apps interactions. The static analysis performed is path-sensitive which accounts for conditions in the extracted rules.

SmartHomeML is a domain specific modelling language (DSML) used to generate applications for smart-home platforms, specifically for Alexa and SmartThings [8]. SmartHomeML adopts Model Driven Engineering (MDE) techniques for the rapid generation of the apps. In this study, they designed a meta-model that abstracts the structure of applications in different platforms, and used it to generate those applications automatically.

Our MDE approach extract permission models from the applications, our experiment finds MDE to be a great option for decreasing the complexity of extracting the expected permissions and behaviour of an application and that by analysing various aspects of SmartApps, this includes code, free-form text and user preferences. Unlike the above techniques, Our approach can detect the three cases of SmarApps over-privilege.

Table 6 summarizes how our tool compares to related work, including our previous tool.

7 CONCLUSIONS & FUTURE WORK

This paper presents the design and development of an MDE approach and a tool [4] for the detection of over-privilege in SmartThings. MDE is used to abstract the system and extract models that represent it at a high level. Those models and constraints allow for the process of model checking. We chose Prolog for the model checking, through applying Prolog rules and queries against Prolog facts to conduct the detection of over-privilege. We demonstrate how our approach is capable of combining multiple sources of information for better understanding of permissions granted to the software. This approach takes into consideration the semantics of the free-form text in the software, which allows for better understanding of the intended permission model in the software, which in turn gives better coverage of over-privilege detection. Experimental results give 78.12% precision and 92.59% recall for case 1, 88.24% precision and 78.95 % recall for case 2, 70.37% precision and 95% recall for case 3. Our plans for future work include working on decreasing the false positives in the detection of over-privilege. We intend to do this by working on the following aspects: investigating how to fix limitations introduced from the chosen programming paradigm of TXL's and improving the natural language understanding in our approach.

8 APPENDIX A: PROLOG RULES FOR CASE 1 OF OVER-PRIVILEGE

```
1   /* Prolog rule of case 1 */
2   overprivilegedCase1(case1,AppName,RuleId,Id,Capability,Resource):-
3   triggerComposition(code,AppName,RuleId,Id,Capability,Resource,_),
4   notAttributeCommandOfCapability(Capability,Resource),
5   attributeCommandOfCapability(Capability2,Resource),
6   not(Capability2=Capability),
7   not(Capability=na),
8   not(Resource=na);
9
10  triggerComposition(code,AppName,RuleId,Id,Capability,_,Resource),
11  not(valueOfAttributeOfCapability(Capability,Resource)),
12  valueOfAttributeOfCapability(Capability2,Resource),
13  not(Capability2=Capability),
14  not(Capability=na),
15  not(Resource=na);
16
17  actionComposition(code,AppName,RuleId,Id,Capability,Resource,_),
18  notAttributeCommandOfCapability(Capability,Resource),
19  attributeCommandOfCapability(Capability2,Resource),
20  not(Capability2=Capability),
21  not(Capability=na),
22  not(Resource=na);
23
24  actionComposition(code,AppName,RuleId,Id,Capability,_,Resource),
25  not(valueOfAttributeOfCapability(Capability,Resource)),
26  valueOfAttributeOfCapability(Capability2,Resource),
27  not(Capability2=Capability),
28  not(Capability=na),
29  not(Resource=na);
30
31  conditionComposition(code,AppName,RuleId,Id,Capability,Resource,_),
32  notAttributeCommandOfCapability(Capability,Resource),
33  attributeCommandOfCapability(Capability2,Resource),
34  not(Capability2=Capability),
35  not(Capability=na),
36  not(Resource=na);
37
```

---

[4]https://cresset.scs.ryerson.ca/ChYP

```
38    conditionComposition(code,AppName,RuleId,Id,Capability,_,Resource),
39    not(valueOfAttributeOfCapability(Capability,Resource)),
40    valueOfAttributeOfCapability(Capability2,Resource),
41    not(Capability2=Capability),
42    not(Capability=na),
43    not(Resource=na).
44    /* End of rule */
45
46    /* Helper Sub-Rules */
47    attributeCommandOfCapability(Capability,Resource):-
48    capability(Capability),
49    attributeCommandOf(Capability,Resource).
50
51    notAttributeCommandOfCapability(Capability,Resource):-
52    capability(Capability),
53    not(attributeCommandOf(Capability,Resource)).
54
55    valueOfAttributeOfCapability(Capability,Resource):-
56    capability(Capability),
57    attributeCommandOf(Capability,Attribute),
58    valueOf(Attribute,Resource).
```

Listing 34. Prolog Rules for Case 1

REFERENCES

[1] Atheer Abu Zaid, Manar Alalfi, and Ali Miri. Check your privileges (ChYP), 2019. URL https://chyp.scs.ryerson.ca/ChYP. Last accessed on 25-1-2020.

[2] Atheer Abu Zaid, Manar H Alalfi, and Ali Miri. Automated identification of over-privileged smartthings apps. In *2019 IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 247–251. IEEE, 2019.

[3] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automated verification of role-based access control security models recovered from dynamic web applications. In *2012 14th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 1–10, 2012. doi: 10.1109/WSE.2012.6320525.

[4] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. A review of smart homes—past, present, and future. *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 42(6):1190–1203, 2012.

[5] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 272–285, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397347. URL https://doi.org/10.1145/3395363.3397347.

[6] James R. Cordy. Txl world, 2012. URL http://txl.ca/txl-resources.html. Last accessed on 25-1-2020.

[7] James R. Cordy. The txl programming language, 2012. URL http://txl.ca/docs/TXL106ProgLang.pdf. Last accessed on 25-1-2020.

[8] Atli F Einarsson, Patrekur Patreksson, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj. Smarthomeml: Towards a domain-specific modeling language for creating smart home applications. In *2017 IEEE International Congress on Internet of Things (ICIOT)*, pages 82–88. IEEE, 2017.

[9] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *Proceedings of the IEEE Symposium on Security and Privacy, 2016*, pages 636–654, 2016.

[10] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Security implications of permission models in smart-home application frameworks. *IEEE Security & Privacy*, 15(2):24–30, 2017.

[11] Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, and Sergio Montenegro. Source code verification for embedded systems using prolog. In *WLP / WFLP*, 2017.

[12] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[13] Nadzeya Kiyavitskaya, Nicola Zeni, Luisa Mich, James R Cordy, and John Mylopoulos. Annotating accommodation advertisements using cerno. In *Information and Communication Technologies in Tourism 2007*, pages 389–400. Springer, 2007.

[14] Somayya Madakam, R. Ramaswamy, and Siddharth Tripathi. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3(05):164, 2015.

[15] Attila Csaba Marosi, Róbert Lovas, Ádám Kisari, and Ernö Simonyi. A novel iot platform for the era of connected cars. In *2018 IEEE International Conference on Future IoT Technologies (Future IoT)*, pages 1–11, 2018. doi: 10.1109/FIOT.2018.8325597.

[16] Roberto Minerva, Abyi Biru, and Domenico Rotondi. Towards a definition of the internet of things (iot). *IEEE Internet Initiative*, 1(1):1–86, 2015.

[17] Owasp. Owasp top ten, 2018. URL https://owasp.org/www-project-top-ten/. Last accessed on 25-1-2020.

[18] Rajesh Parekh and Vasant Honavar. Grammar inference, automata induction, and language acquisition. *Handbook of natural language processing*, pages 727–764, 2000.

[19] SmartThings. Anatomy and life cycle of a smartapp, 2018. URL https://docs.smartthings.com/en/latest/smartapp-developers-guide/anatomy-and-life-cycle-of-a-smartapp.html. Last accessed on 25-1-2020.

[20] SmartThings. Hongtat/smartthings-capabilities: Smartthings capabilities, 2019. URL https://github.com/hongtat/smartthings-capabilities. Last accessed on 31-10-2021.

[21] SmartThings. Writing your first smartapp, 2019. URL https://docs.smartthings.com/en/latest/getting-started/first-smartapp.html. Last accessed on 25-1-2020.

[22] Shruti G Hegde Soumyalatha. Study of iot: understanding iot architecture, applications, issues and challenges. In *1st International Conference on Innovations in Computing & Net-working (ICICN16), CSE, RRCE. International Journal of Advanced Networking & Applications*, 2016.

[23] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer security: principles and practice.* Pearson Education Upper Saddle River, NJ, USA, 2012.

[24] Yuan Tian, Nan Zhang, Yue-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *Proceedings of the 26th USENIX Security Symposium*, pages 361–378, 2017.

[25] Felix Wortmann and Kristina Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 2015.

[26] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1074–1088. ACM, 2018.

[27] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1133–1150, 2019.