JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

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

Manar H. Alalfi, Atheer Abu Zaid, and Ali Miri
Toronto Metropolitan University, Canada

ABSTRACT Software vulnerabilities in access control models can represent a serious threat in a system. In fact, OWASP lists broken access control as number 1 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 Reverse Engineering (MDRE) in addition to static analysis. This approach allows for better coverage of privilege escalation detection than static analysis alone as it takes advantage of analyzing free-form text that carries extra permissions details. Our experimental results demonstrate high accuracy in detecting over-privilege vulnerabilities in IoT applications.

KEYWORDS Model Driven Reverse Engineering, Access Control Security vulnerabilities, Security Verification, 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 (Gubbi et al. 2013). IoT has a variety of definitions by different groups in the academia and the industry (Gubbi et al. 2013; Madakam et al. 2015; Wortmann & Flüchter 2015). 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 (Minerva et al. 2015). IoT is now widely adopted in many important domains, such as in healthcare systems, smart cities, smart homes and autonomous cars (Soumyalatha 2016) (Marosi et al. 2018). Employing IoT systems in cities has proved to be useful, such as in improving transportation by gathering data and analyzing it (Soumyalatha 2016). 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 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 (Alam et al. 2012; Alhanahnah et al. 2020). In (Zhou et al. 2019), *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 (Fernandes et al. 2016;

JOT reference format:
Manar H. Alalfi, Atheer Abu Zaid, and Ali Miri. *A Model-Driven-Reverse Engineering Approach for Detecting Privilege Escalation in IoT Systems.* Journal of Object Technology. Vol. 22, No. 1, 2023. http://dx.doi.org/10.5381/jot.2023.22.1.a1

Alhanahnah et al. 2020).

Many programmable Smart home platforms have emerged over the last few years and they have provided users with a broad set of benefits. These platforms provide third-party developers with the opportunity to distribute apps for compatible devices. Unfortunately, this flexibility to allow for development and distribution of apps by third-party developers can also introduces potential new risks and vulnerabilities in systems that use these apps. Security audit to identify and remove these potential vulnerabilities can be challenging and time consuming. The main contribution of this paper is to design and implement an automated detection tool that leverages MDE, static analysis and natural language processing to identify these vulnerabilities with a high accuracy. Unlike other previous related work, this approach allows for better coverage of privilege escalation detection than static analysis alone as it takes advantage of analyzing free-form text that carries extra permissions details.

Our approach uses Samsung's SmartThings as the platform of interest. 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* (SmartThings n.d.). 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.

```
definition
(   name: "",
    namespace: "",
    author: "",
    description: "",
    ...)
preferences
{   section("Select devices")
    { input "themotion", "capability.motionSensor", title
     : "Select a motion sensor"
      input "thelock", "capability.lock", title: "Select
     a lock" }}
def installed() { initialize() }
def updated() { unsubscribe()  initialize() }
def initialize() { subscribe themotion, "motion.active",
     activeHandler }
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 paper. 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.

This paper is divided into seven sections. The next section provides motivations for this work, including discussion on the three most common causes for over-privilege vulnerabilities in SmartApps. Section 3 describes the background on methodology and tools used. A detailed overview of our approach can be found in Section 4, followed by analysis of its performance in Section 5. Related work is discussed in Section 6, and conclusions and suggestions for future work can be found in Section 7.

2. Motivation

One of the fundamental software security principles is the least privilege principle (Stallings et al. 2012). 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" - the top security vulnerability risk in OWASP's top 10 vulnerability list[1]. OWASP top 10 vulnerabilities is a valuable resource for researchers and businesses to consider when auditing systems for security.

Design flaws in the SmartThings platform have resulted in apps acquiring access to more resources than required (Fernandes et al. 2017). 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

[1] https://owasp.org/www-project-top-ten/

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.

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.

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, as illustrated in Listings 1-3:

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

```
preferences
{ section("Select devices")
  { input "thepresence", "capability.presenceSensor",
      title: "Select a presence sensor"
    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:

```
def presenceHandler(evt)
{   if (thebattery.currentLock == "locked")
    {   thebattery.unlock()
        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.

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 (Fernandes et al. 2016). 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.

```
definition(
 name: "UnlockDoorApp",
 namespace: "",
 author: "",
 description: "Unlock the front door on my arrival",
 category: "",
 iconUrl: "",
 iconX2Url: ""
)
```

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.

```
preferences
{   section("Select devices")
    { input "thepresence", "capability.presenceSensor",
     title: "Select a presence sensor"
      input "thelock", "capability.lock", title: "Select
     a lock"
    }
}
```

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.

```
light -> off
light -> on
```

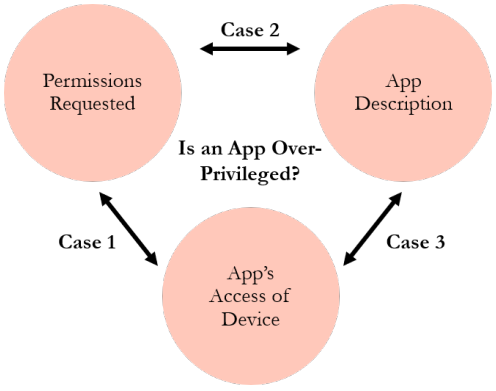


Figure 1 App Components Used in Analysis

Table 1 Causes of Over-Privilege in SmartThings

Case1: Coarse SmartApp-SmartDevice Binding	**Case2: Semantically Over-Privileged**	**Case 3: Coarse-Grained Capabilities**
1- Request set of capabilities A 2- Then, access set of capabilities B, such that A $\not\supseteq$ B	1-Infer from free-form text the needed set of capabilities A 2- Then, request set of capabilities B, such that A $\not\supseteq$ B	1- Infer the use of a set of commands attributes A, whose owner capability is properly requested 2- Then, access set of commands attributes B, such that A $\not\supseteq$ B

```
lockOnly -> lock
lock -> lock
lock -> unlock
...
powerSource -> powerSource
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.

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.

```
def presenceHandler(evt)
{   if (thelock.currentLock == "locked")
    {   thelock.unlock()
        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.

In this paper, we present an MDRE 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, MDE-ChYP [2], that employs MDE in addition to static analysis to translate multiple sources of information in the app into permission rules. Unlike other existing techniques (Zhang et al. 2018; Tian et al. 2017; Einarsson et al. 2017; Fernandes et al. 2016), 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.
- To evaluate our approach in terms of accuracy and performance, we have conducted a set of experiments over a wide range of apps.
  The analysis provided in this work requires information coming from three sources, the app source code, the app NL statements in the description, and the app user configuration specified in the preferences section. Each of those resources has its own challenge in the analysis. The NL source required a complicated analysis where we have to generate a grammar to capture permissions related information from text. The source code has different static contexts such as single flow statements and control flow statements. The configuration statements in the preferences need to be related to artifacts recovered from SmartThings documentation, more specifically, the devices, their capabilities, commands and attributes. All these artifacts need to be represented and related to each other via a strong inference engine.

Table 2 summarizes how our tool compares to related work, including our previous tool.

3. Related Work

Fernandes *et al.* (Fernandes et al. 2016) performed the first empirical security analysis of the SmartThings platform, and provided lessons for the design of smart home programming

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

Table 2 Comparison with Related Work

Paper	What it does	Techniques Employed
Security Analysis of Emerging Smart Home Applications. Fernandes et al. (2016)	Over-privilege detection: Cases 1 & 3	Static analysis Run-time analysis Manual analysis
HoMonit. Zhang et al. (2018)	Over-privilege detection: Case 3	Static analysis and NLP to extract DFA of expected behaviour. Side channel analysis to extract DFA during run-time
SmartAuth. Tian et al. (2017)	Over-privilege detection: Cases 1 & 3	NLP Program analysis
IoTCom. Alhanahnah et al. (2020)	Detects threats in IoT at the interaction level between apps	Static analysis, path-sensitive Formal analysis
SmartHomeML. Einarsson et al. (2017)	Generate smart-home applications using a DSML	MDE for fast generation of apps
MDE-ChYP (our tool)	Over-privilege detection: Cases 1,2 & 3	Static analysis to detect over-privilege. MDE for extracting expected and actual behaviors of apps Grammer inference for NLP and that to recover permission information from text

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.* (Zhang et al. 2018), 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. However, 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.* (Tian et al. 2017) 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.

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 (Zhang et al. 2018). 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 (Tian et al. 2017).

IoTCom is an approach and a tool developed to detect threats at the interaction level between IoT apps (Alhanahnah et al. 2020). 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 (Einarsson et al. 2017). 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.

4. Background

Model-Driven Engineering (MDE), an approach to software engineering where models are considered the first-class artefacts of the software engineering process. Such models are used to improve and to simplify the software development by providing high level and domain specific abstractions employed at all stages of software development including requirements, design, implementation and maintenance. One of the main benefits of MDE is automation, in the context of forward engineering, transformation languages can be used to describe and automatically transform high level models into source code, this will enable developers to focus on the important aspects of the system, neglecting technical details specific to the target platform (Pastor 2017). MDE can be applied in the context of reverse engineering, an approach used to support the comprehension of software systems, and that by recovering models of the system under analysis at a higher level of abstraction than source code. Reverse engineering can be applied in many contexts. The most common scenario is the comprehension of code, to support other tasks such as maintenance, software evolution, integration or interface with legacy systems(Rugaber & Stirewalt 2004).

The application of MDE to reverse engineering is known as model driven reverse engineering (MDRE). MDRE is defined as "producing descriptive models from existing systems that were previously produced somehow." The models generation from source code and models transformations can be automated. After the generation, the obtained models can be analyzed by domain experts or by appropriate tools, otherwise they can be used to start a model-driven development phase(Favre 2004).

Our approach uses MDRE approach to the verification of SmartThings apps on privilege escalation vulnerability. Our approach uses source transformation technology, TXL(Cordy 2012a), to automatically analyze the various artifacts of the SmartApps and that to recover a permission security model form source code. The recovered security model confirms to a permission meta-model we developed for SmartThings Apps, the details of the model recovery stage is described in section 5.1. To verify the recovered model on the three cases of privilege escalation, we codified the permission met-model in the form of prolog facts, the main goal is to use the prolog verification engine to verify the 3 cases of privilege escalation. We used the transformation technology, TXL, to transform the recovered artifacts into the permission model encoded in the form of prolog facts. Below, we provide a brief description on the technology used to develop our semi-automated approach.

TXL is 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 (Cordy 2012a).

GrammarInference is the task of learning grammars or languages from training data, for instance, by examining the sentences of an unknown language. It is a type of inductive inference, the name given to learning techniques that try to guess general rules from examples. The basic problem is to find a grammar consistent with a training set of positive examples. Grammar inference is used successfully in a variety of fields such as pattern recognition, computational biology and natural language processing.

Prolog is a logical and declarative programming language largely used for applications in natural language understanding and expert systems 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(Alalfi et al. 2012),and it proved to be efficient and scalable. It was also recently used for embedded systems verification (Flederer et al. 2017).

5. Approach Overview

Figure 2 layouts the three main components of our MDRE 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.

5.1. Extracting Permission Rules

As described in Figure 2, the first step of our approach is to extract 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 (Alhanahnah et al. 2020). In order to automate the process, we first need to define a meta-model that describes the permission model for the SmartThings platform, which we will use as a basis of recovering permissions artifacts from the SmartThings apps. A definition of such meta-model is described in subsection 5.1.1. Then, in subsection 5.1.2, we describe the process we followed to enable the extraction of the permission

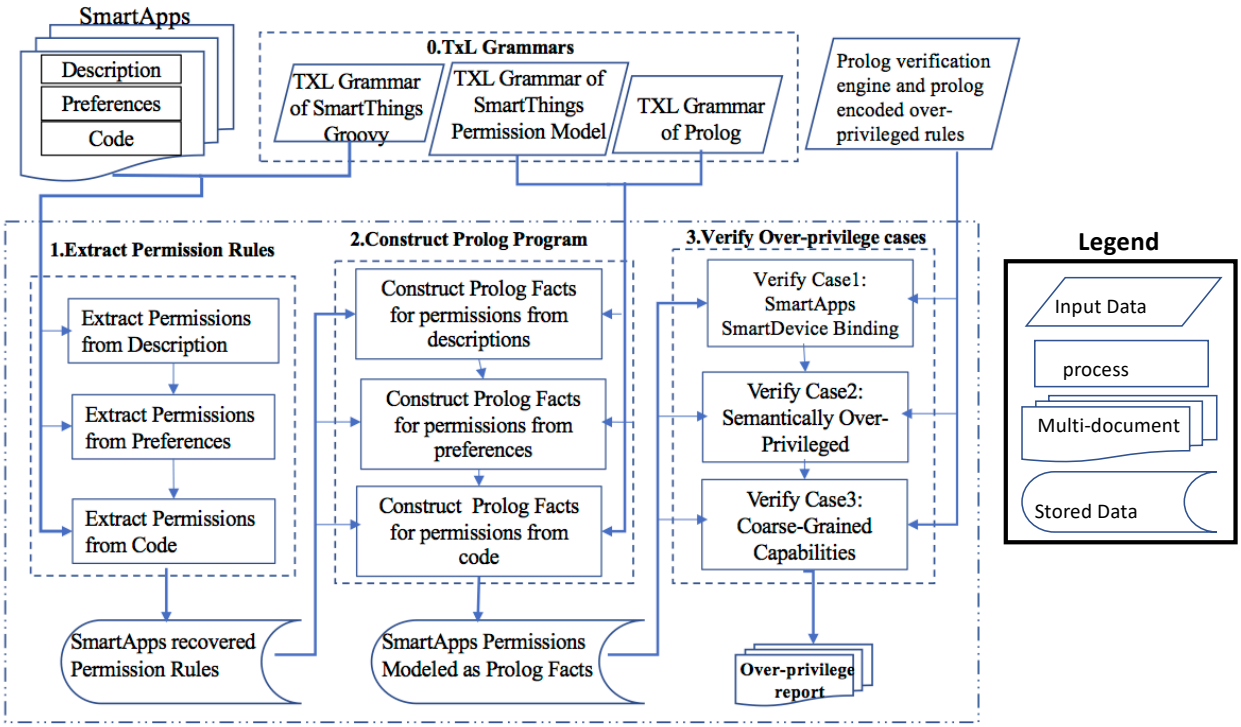


Figure 2 MDE-ChYP Approach Stages

artifacts from SmartThings apps descriptions. Since the description is expressed in NL, and in order to automate the extraction process, we have developed an automated annotation process using grammar inference. The generated grammar is used to annotate the NL SmartApps' description and it confirms to the meta-model we defined in the subsection 5.1.1

5.1.1. Smart-Home Permission Meta-Model Definition

In this section, we describe the simplified meta-model we developed which abstracts the permission model used in SmartThings platform. Figure 3 demonstrates the smart-home permission meta-model, which we adopted and revised from IoTCOM and SmartHomeML (Alhanahnah et al. 2020; Einarsson et al. 2017). 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:

- 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

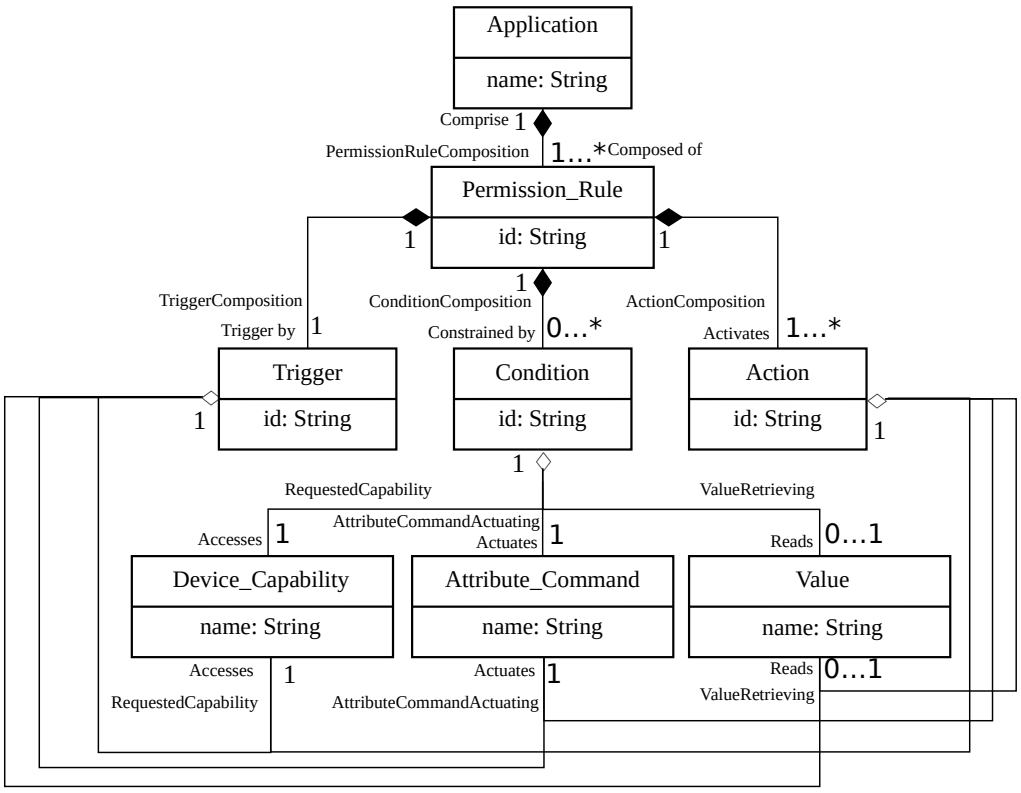


Figure 3 Smart-Home Permission Meta-Model

5.1.2. 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 (Kiyavitskaya et al. 2007).

To automate the process of semantic annotation, we use TXL (Cordy 2012a), 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. (SmartThings 2019) 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 5.1.2)

```
% Definition of all capabilities
define device
    [accelerationSensor] | [alarm] | [audioNotification]
     | [battery] | [beacon] |
    [bulb] | [button] | [carbonDioxideMeasurement] | [
     colorControl] | [colorTemperature] |
    [configuration] | [consumable] | [contactSensor] | [
     doorControl] | [energyMeter] |
    [estimatedTimeOfArrival] | [garageDoorControl] | [
     holdableButton] | [illuminanceMeasurement] |
    [imageCapture] | [indicator] | [infraredLevel] | [
     light] | [lockOnly] | [lock] |
    [mediaController] | [momentary] | [motionSensor] | [
     musicPlayer] | [notification] | [outlet] |
    [pHMeasurement] | [polling] | [powerMeter] | [
     powerSource] | [presenceSensor] | [refresh] | ...
end define

% Definition of the 'accelerationSensor' capability
define accelerationSensor
...
end define

% Definition of all commands
define command
...
end define

% Definition of all attributes
define attribute
...
end define

% Definition of all values
define value
...
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 (Parekh & Honavar 2000). 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. [3] 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 repeated 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
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

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

5.2. Extracting/Constructing Prolog Facts for SmartThings Permission-rules

As described in Figure 2, the second stage of our approach is to construct Prolog facts. Since this stage is heavily connected to the first stage of the approach, permission rules extraction, this section describes the two stages together in the form of extract/construct.

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 (Alalfi et al. 2012) and it proved to be efficient and scalable. It was also recently used for embedded systems verification (Flederer et al. 2017). 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 Section 4.1 and the corresponding Prolog facts. Each permission rule component accesses a capability and either a command or attribute and its value.

Table 3 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.

5.2.1. Extracting/Constructing Permission-Rule Facts from Description In this subsection we describe the process of extracting the permission artifacts from the SmartApps' description and how we used it to construct the corresponding Prolog facts. The inputs to this stage are:

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

Table 3 Permission Meta-Model Mapping (Figure 3) to Prolog Facts

Prolog Fact	**Mapping to Permission Meta-Model, (Figure 3)**
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.

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 3. Facts are then saved in a file for later use.

```
<program>
 <repeat_description>
  <description><repeat_permission_rule>
    <permission_rule>
     <repeat_rule_component>
      <rule_component><action><repeat_component_element>
          <component_element><device><switch>Turn</
     switch></device></component_element>
          <component_element><word><id>your</id></word></
     component_element>
          <component_element><device><colorControl>
     lights</ colorControl></device></component_element>
          <component_element><attribute><onAttribute>on</
     onAttribute></attribute></component_element>
          <empty/>
        </repeat_component_element>
       </action>
      </rule_component>
      <rule_component><trigger>
        <trigger_indicator>when</trigger_indicator>
        <repeat_component_element>
          <component_element><device><motionSensor>
     motion</ motionSensor></device></component_element>
          <component_element><word><id>is</id></word></
     component_element>
          <component_element><value><detectedValue>
     detected</ detectedValue></value></component_element
     >
...
```

Listing 13 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 13, then TXL will again translate it to Prolog facts (see Listing 14). The intermediate representation in Listing 13 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'*. In this example, we will focus on *triggerComposition*, *conditionComposition*, *actionComposition* and *device_capability* from these facts.

```
application(desc, AppName).

permission_rule(desc, AppName, rule1).
% Action facts
action(desc, rule1, action1).
device_capability(desc, action1, switch).
attribute_command(desc, action1, on).
value(desc, action1, on).
actionComposition(desc, AppName, rule1, action1, switch,
     on, on).
% Trigger facts
trigger(desc, rule1, trigger1).
device_capability(desc, trigger1, motionSensor).
attribute_command(desc, trigger1, motion).
value(desc, trigger1, detected).
triggerComposition(desc, AppName, rule1, trigger1,
     motionSensor, motion, detected).
```

Listing 14 Prolog Facts Extracted from Description

5.2.2. Extracting/Constructing 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 15 is an example of a SmartApp requesting the switch capability. This input statement is translated to a Prolog fact (as in the Listing 16), and saved to the same facts file used previously.

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

Listing 15 Capabilities Requested by SmartApp

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

Listing 16 Prolog Fact for Case 2 (Semantically Over-Privileged)

5.2.3. Extracting/Constructing Permission-Rule Facts from Code

The third step 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 17, 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 18. 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 19. In the case that we extracted either an attribute/command or a value, and this 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.

```
def installed()
{   subscribe(motion1, "motion.active",
     motionActiveHandler)}

def motionActiveHandler(evt) {
    if(switch1.currentValue("switch") == "off"){
        switch1.on()}}
```

Listing 17 Trigger and Associated Event Handler

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

Listing 18 Trigger Fact

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

Listing 19 Condition and Action Facts

5.3. Verifying Over-privilege cases

As described in Figure 2, the third step of our approach is the verification of Over-privilege cases. To do that, we create the Prolog model checker from within the TXL main program of the tool. Line 1 in Listing 20 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 4 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.

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

Listing 20 Command to Create the Prolog Model Checker

5.3.1. Verification Rules 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:

Table 4 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

– 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. We have given an illustrative example of these rules in Listing 22

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 21 Sample Facts Relevant to Over-Privilege Case 1

5.3.2. Verification Rules 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 has 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)

The 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 22 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 23 Sample Facts Relevant to Over-Privilege Case 2

5.3.3. Verification Rules 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 (Fernandes et al. 2016). 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 25).
- 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.

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

```
overprivilegedCase3(case3,AppName,RuleIdCode,
     TriggerIdCode,ConditionIdCode,ActionIdCode,
     Capability,AttributeCommand,Value):-
    /*The whole action is missing in description*/
    actionNotInDesc(AppName,RuleIdCode,ActionIdCode,
     Capability,AttributeCommand,Value);
actionNotInDesc(AppName,RuleIdCode,ActionIdCode,
     Capability,AttributeCommand,Value):-
    actionComposition(code,AppName,RuleIdCode,
     ActionIdCode,Capability,AttributeCommand,Value),
    not(actionComposition(desc,AppName,_,_,Capability,
     AttributeCommand,Value)),
    not(threeNAs(Capability,AttributeCommand,Value)),
    not(actionTwoNAs(AppName,Capability,AttributeCommand,
     Value)).
%* Capability,
threeNAs(Capability,AttributeCommand,Value):-
    Capability=na,
    AttributeCommand=na,
    Value=na.
/*  Use Case:
    Desc: "Lock the front door"
    Code: "Unlocks the door" */
actionTwoNAs(AppName,Capability,AttributeCommand,Value):-
    AttributeCommand = na,
    Value = na,
    actionComposition(desc,AppName,_,_,Capability,_,_).
```

Listing 24 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.

```
/* Facts in description */
permission_rule(desc, AppName, ruleNumber).
triggerComposition(desc, AppName, ruleNumber, trigger1,
     presenceSensor, presence, notpresent).
actionComposition(desc, AppName, ruleNumber, action1,
     switch, off, off).
/* Facts in code */
permission_rule(code, AppName, ruleNumber).
triggerComposition(code, AppName, ruleNumber, trigger1,
     presenceSensor, presence, na).
actionComposition(code, AppName, ruleNumber, action1,
     switch, on, on).
```

Listing 25 Sample Facts Relevant to Over-Privilege Case 3

6. 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 (Cordy 2012b) as a helper grammar in the transformation. We evaluate the effectiveness of our approach by answering the following research questions:

1. RQ1: Can our automated analysis, MDE-ChYP, report the 3 cases of over-privilege in SmartThings apps once applied to the available dataset? (To be addressed Section 6.2)
2. RQ2: How does privilege escalation detection performed by MDE-ChYP approach presented in this paper (Abu Zaid et al. 2019b) compare to the ChYP tool? (Abu Zaid et al. 2019b) (To be addressed in Section6.3)
3. RQ3: How effective is our approach in detecting privilege escalation in SmartThings apps when evaluated for precision and recall? (To be addressed in Section 6.4)

6.1. Dataset

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. In addition this dataset has been used by other researchers who performed experiments to detect over-privilege escalation (Fernandes et al. 2016). We also used the dataset provided by Zhang *et al.* (Zhang et al. 2018) to evaluate the detection of case 3. We refer to this dataset as "HoMonit" in this evaluation.

6.2. 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 5 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. 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

Figure 4 Analysis time of privilege escalation detection in SmartApps in relation to size of the dataset

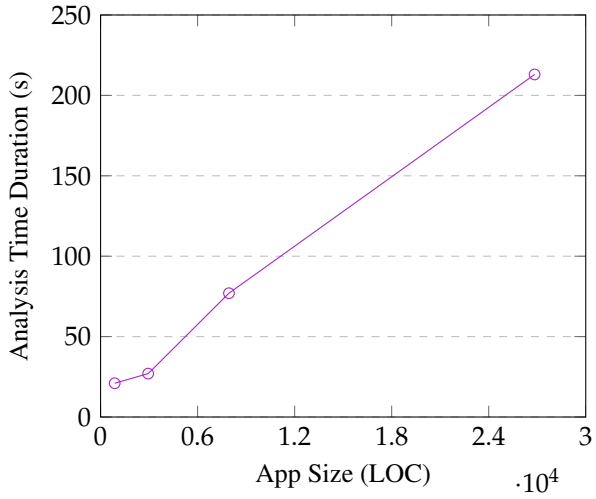
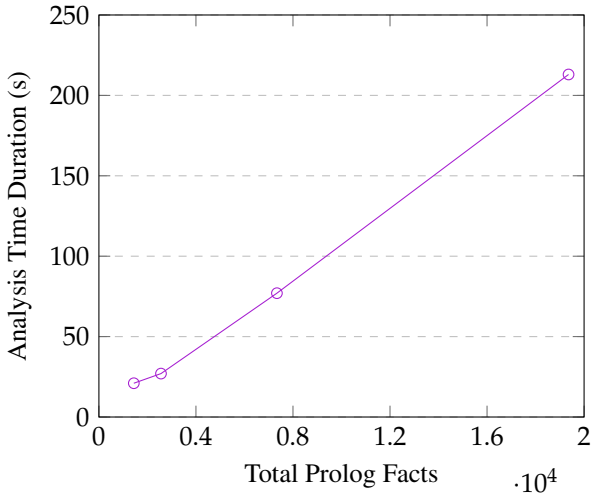


Figure 5 Analysis time of privilege escalation detection in SmartApps in relation to no. of Prolog facts



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.

To demonstrate the performance of the tool, we measure the analysis time for each dataset classification separately, as in Table 6. 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 SmartThings apps, the number of Prolog facts generated by each app and time spent in analysis. Table 6 details the performance data reported after running the evaluation experiment. The analysis shows that the approach is scaling well as it was able to process (38.6K LOC, and 30.7K prolog facts) in less than 5 minutes. The apps used in the experiment are real world apps, each app is a single file and ranges from 27 to 732 physical lines of code, with the blank and comment lines excluded. The average LOC is 118 and the median is 65.5. Figures 4 and 5 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. In this section, we have addressed RQ1 by demonstrating that MDE-ChYP is capable of reporting all 3 cases of over-privilege. We have also included the 'Analysis Time Duration' to Table 6 to illustrate a typical run time required by our solution', however, and in order to quantitatively evaluate the accuracy of our finding, it would deem difficult to manually check all the 2000 reported cases of over-privilege as such RQ2 will look into comparing MDE-ChYP to a base tool ChYP on a subset of apps that reported over-privilege case1.

6.3. 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 (Abu Zaid et al. 2019a), 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 5. The list of apps used in this experiments are displayed in table 7. 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 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 7).

Table 8 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 (Kiyavitskaya et al. 2007). Below are the formulas used to calculate the precision and recall (Zhang et al. 2018; Kiyavitskaya et al. 2007). 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).

The true negative rate TNR (also called specificity), which is the probability that an actual negative will test negative.

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

Table 5 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

Table 6 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

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

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

$$TNR = \frac{TN}{TN + FP} \tag{4}$$

F-score or F-measure is a measure of a test's accuracy. It is calculated from the precision and recall of the test. F-measure provides a single metric that weights the two ratios (precision and recall) in a balanced way, requiring both to have a higher value for the F1-score value to rise. Very small precision or recall will result in lower overall score. Thus it helps balance the two metrics.

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 26. 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 27 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 {
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 26 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 27 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 SmartThings

Table 7 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

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.

6.4. Manual Validation

In section 6.2, we present the approach results when evaluated on the whole dataset. The approach reported over-privilege

Table 8 Evaluation comparison between the MDE and the static analysis based tools

Tool	Precision	Recall	F measure
Static Analysis Tool (ChYP)	100%	60.98%	75.0%
MDE-ChYP	82.50%	80.49%	81.48%

occurrences of all three cases as shown in Table 5.

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, 6.3, 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.

6.4.1. 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.

```
definition(
 name: "Big Turn OFF",
 namespace: "smartthings",
 author: "SmartThings",
description:
     "Turn your lights off when the SmartApp is tapped",
    ...)
preferences {
 section("When I touch the app, turn off...") {
  input "switches", "capability.switch", multiple: true
 }}
def installed()
{   subscribe(location, changedLocationMode)
 subscribe(app, appTouch) }
...
def appTouch(evt) {
 log.debug "appTouch: $evt"
 switches?.off()}
```

Listing 28 App Before Injecting Over-privilege

Listing 28 provides an example of an app with no over-privileges of any case. The app is modified in listing 29 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 29).

```
definition(
    name: "Big Turn OFF",
    namespace: "smartthings",
    author: "SmartThings",
    description: "Turn your lights off when the SmartApp
     is tapped",
    ...)
preferences {
 section("When I touch the app, turn off...") {
  input "switches", "capability.switch", multiple: true
 }}
def installed()
{   subscribe(location, changedLocationMode)
 subscribe(app, appTouch) }
...
def appTouch(evt) {
 log.debug "appTouch: $evt"
 switches?.off()
 switches?.siren()}
```

Listing 29 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 28 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 30 shows some of the capabilities in the SmartThings documentation. The resulting over-privileged app is shown in listing 31, line 11 shows the app requested the capability *accelerationSensor* without mentioning it in the description in line 5.

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

Listing 30 Snippet of SmartThings List of Capabilities

```
definition(
 name: "Big Turn OFF",
 namespace: "smartthings",
 author: "SmartThings",
 description:
     "Turn your lights off when the SmartApp is tapped",
    ...)
preferences {
 section("When I touch the app, turn off...") {
  input "switches", "capability.switch", multiple: true}
  section("") {
  input "sensor", "capability.accelerationSensor"
                          , multiple: true}
}
```

Listing 31 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 28 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 32 shows the app after the injection of Case 3. We created 20 mutants in 20 apps from the dataset.

```
definition(
 name: "Big Turn OFF",
 namespace: "smartthings",
 author: "SmartThings",
 description:
     "Turn your lights off when the SmartApp is tapped",
    ...)
preferences {
 section("When I touch the app, turn off...") {
  input "switches", "capability.switch", multiple: true
 }}
def installed()
{   subscribe(location, changedLocationMode)
 subscribe(app, appTouch) }
...
def appTouch(evt) {
 log.debug "appTouch: $evt"
    switches?.off()
 switches?.on()}
```

Listing 32 App After Injecting Case 3 of Over-privilege

For further details, the complete benchmark could be found online, as well as the public datasets. [4]

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 5) 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 9, presents the statistical validation results for our analysis. Our tool could detect over-privilege vulnerabilities in malicious apps with high precision and recall. Column 1, presents the dataset type whether it is malicious or benign and which over-privileged type it covers (Case1, Case2 or Case3). The second column presents the total number of apps used for the evaluation, each app may have more than one over privilege occurrence, that is why we have included a third column to report on the number of over-privilege occurrences/absences for each case. Please note that the third row has 116 confirmed absences of over-privilege patterns since the dataset used in the evaluation is benign and covers Case2, same applies to row 5, but the benign set covers Case3 of over privilege. The intention of evaluating the tool on a dataset that is benign, is to be able to compute the (TNR), which presents the probability that an actual negative/benign will test negative/benign. F-measure value shows an overall improvement of the analysis results when compared with ChYP on the detection for case1 vulnerability. And a percentage higher than 80% for the malicious dataset (all cases).

For the benign dataset, results demonstrate 82 benign instances and 34 false negatives for case 2. It also reports 20 instances of benign-Case3, and 31 false negatives for Case3. The TNR is is relatively good for Case 2, but not for Case3, and that is evident by the large number of false negatives.

7. Threat to Validity

Threat to internal validity: We investigated the apps evaluated in section 6.4 for the reasons of false positives and false negatives and pinpointed some reasons of error and areas for improvements. A limitation of the current approach of processing natural language is that it does not take into consideration text similarity. For instance, the word 'present' in the following example is parsed as a 'presenceSensor', while it could also mean 'motionSensor' at the same time. This results in false positives being reported. The title "Motion here" is parsed as an action and in a separate rule.

```
description: "Turns on an outlet when the user is present
      and off after a period of time"
       title: "Motion here"
       title: "And (optionally) these sensors being
     present"
       section("When someone's around because of...")
     {...}
       section("Turn on these outlet(s)") {...}
```

Listing 33 App After Injecting Case 3 of Over-privilege

```
/* Facts from description: */

triggerComposition(code,zigbeeCurlingIron,rule5,trigger1,
     motionSensor,motion,active).
actionComposition(code,zigbeeCurlingIron,rule5,action1,
     switch,on,on).
actionComposition(desc,zigbeeCurlingIron,rule4,action1,
     Motion,na,na).

/* Facts from code: */
triggerComposition(desc,zigbeeCurlingIron,rule1,trigger1,
     presenceSensor,na,present).
actionComposition(desc,zigbeeCurlingIron,rule1,action1,
     switch,on,on).
```

Listing 34 Facts relevant to false positive case

```
<component_element><device><presenceSensor>present</
     presenceSensor></device></component_element>
```

Listing 35 TXL parse of the device

One reason behind this limitation is the grammar developed does not contain enough definitions to understand synonymous, for instance, 'moisture', 'water' and 'wet' are all connected and could be used interchangeably wherever needed. The grammar can be easily adapted to add such information which can be captured with more domain expert knowledge.

Another threat to validity is when descriptions appear in separate sections, it can only be decided if the descriptions belong to the same permission rule from the semantics. And so, two separate permission rules will be parsed, independent of each other, breaking the relation necessary to understand the underlying permission rule. A similar scenario happens when TXL encounters an apostrophe in the NL descriptions, this leads to cutting off the parsing of a permission rule component leading to two separate permission rule components.

To summarize, 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 punctuation is one area that could immediately improve the results. Another possible improvement is expanding the vocabulary in the tool produced using grammar inference supported and validated by domain knowledge expertise.

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.

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

Table 9 Evaluation of MDE-ChYP on the mutation dataset

Dataset	Total Apps Evaluated	Occurrences in dataset	TN	TP	FP	FN	Precision	Recall	F-measure	TNR
Malicious - Case 1	25	41	_	33	7	8	82.50%	80.49%	81.48%	_
Malicious - Case 2	19	19	_	19	8	0	70.37%	1	82.60%	_
Benign - Case 2	60	116	82	0	0	34	_	_	_	70.68%
Malicious - Case 3	20	20	_	19	8	1	70.30%	95.0%	80.85%	_
Benign - Case 3	20	51	20	0	0	31	_	_	_	37.04%

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.

Threat to external validity: Our approach uses Samsung's SmartThings as the platform of interest. This platform is similar to other programmable platforms. It is cloud-based and provides a programming framework for third-party developers (Fernandes et al. 2016). Our case study on SmartThings platform shares two other common features with other platforms: authorization and authentication through the capability model, and the support of event-driven processing. Thus, we anticipate the analysis done on this platform on privilege escalation detection can be extended to other working platforms. We also have focused our work on detecting *privilege escalation* vulnerabilities due to the risk they represent in attempts to gain unauthorized access to systems. We anticipate that our systematic approach generalizes to address other types of vulnerabilities. The generalization of our approach is due to the technologies we use in the analysis. The grammar inference approach guided by model reveres engineering and it's application to semantic annotation for NL has been used successfully in other projects and contexts (Kiyavitskaya et al. 2007), and our successful application of the approach to semantically annotate permission rules from IoT Apps demonstrates the extendability of the approach.

8. Conclusions & Future Work

This paper presents the design and development of an MDE approach and a tool, MDE-ChYP, [5] 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.

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

Appendix A. Prolog Rules for Case 1 of Over-Privilege

```
/* Prolog rule of case 1 */
overprivilegedCase1(case1,AppName,RuleId,Id,
 Capability,Resource):-
triggerComposition(code,AppName,RuleId,Id,Capability,
 Resource,_),
notAttributeCommandOfCapability(Capability,Resource),
attributeCommandOfCapability(Capability2,Resource),
not(Capability2=Capability),
not(Capability=na),
not(Resource=na);

triggerComposition(code,AppName,RuleId,Id,Capability,
 _,Resource),
not(valueOfAttributeOfCapability(Capability,Resource)
 ),
valueOfAttributeOfCapability(Capability2,Resource),
not(Capability2=Capability),
not(Capability=na),
not(Resource=na);

actionComposition(code,AppName,RuleId,Id,Capability,
 Resource,_),
notAttributeCommandOfCapability(Capability,Resource),
attributeCommandOfCapability(Capability2,Resource),
not(Capability2=Capability),
not(Capability=na),
not(Resource=na);

actionComposition(code,AppName,RuleId,Id,Capability,_
 ,Resource),
not(valueOfAttributeOfCapability(Capability,Resource)
 ),
valueOfAttributeOfCapability(Capability2,Resource),
not(Capability2=Capability),
not(Capability=na),
not(Resource=na);

conditionComposition(code,AppName,RuleId,Id,
 Capability,Resource,_),
notAttributeCommandOfCapability(Capability,Resource),
attributeCommandOfCapability(Capability2,Resource),
not(Capability2=Capability),
not(Capability=na),
not(Resource=na);
```

```
    conditionComposition(code,AppName,RuleId,Id,
     Capability,_,Resource),
    not(valueOfAttributeOfCapability(Capability,Resource)
     ),
    valueOfAttributeOfCapability(Capability2,Resource),
    not(Capability2=Capability),
    not(Capability=na),
    not(Resource=na).
    /* End of rule */

    /* Helper Sub-Rules */
    attributeCommandOfCapability(Capability,Resource):-
    capability(Capability),
    attributeCommandOf(Capability,Resource).

    notAttributeCommandOfCapability(Capability,Resource)
     :-
    capability(Capability),
    not(attributeCommandOf(Capability,Resource)).

    valueOfAttributeOfCapability(Capability,Resource):-
    capability(Capability),
    attributeCommandOf(Capability,Attribute),
    valueOf(Attribute,Resource).
```

Listing 36 Prolog Rules for Case 1

Acknowledgments

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and Toronto Metropolitan University, Faculty of Science Dean's Research Fund. We would like to thank the reviewers of this paper for their helpful comments and suggestions.

References

Abu Zaid, A., Alalfi, M., & Miri, A. (2019b). *Check your privileges (ChYP).* Retrieved from https://chyp.scs.ryerson.ca/ChYP (Last accessed on 25-1-2022)

Abu Zaid, A., Alalfi, M. H., & Miri, A. (2019a). Automated identification of over-privileged smartthings apps. In *2019 IEEE international conference on software maintenance and evolution ICSME* (pp. 247–251).

Alalfi, M. H., Cordy, J. R., & Dean, T. R. (2012). 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)* (p. 1-10). doi: 10.1109/WSE.2012.6320525

Alam, M. R., Reaz, M. B. I., & Ali, M. A. M. (2012). 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.

Alhanahnah, M., Stevens, C., & Bagheri, H. (2020). Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th acm sigsoft international symposium on software testing and analysis* (pp. 272 – 285). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3395363.3397347 doi: 10.1145/3395363.3397347

Cordy, J. R. (2012a). *The txl programming language.* Retrieved from http://txl.ca/docs/TXL106ProgLang.pdf (Last accessed on 25-1-2022)

Cordy, J. R. (2012b). *Txl world.* Retrieved from http://txl.ca/txl-resources.html (Last accessed on 25-1-2020)

Einarsson, A. F., Patreksson, P., Hamdaqa, M., & Hamou-Lhadj, A. (2017). Smarthomeml: Towards a domain-specific modeling language for creating smart home applications. In *2017 ieee international congress on internet of things (ICIOT)* (pp. 82–88).

Favre, J.-M. (2004). Foundations of model (driven)(reverse) engineering. In *Dagsthul seminar on language engineering for model driven development, drops, http://drops. dagstuhl. de/portals/04101* (Vol. 200).

Fernandes, E., Jung, J., & Prakash, A. (2016). Security analysis of emerging smart home applications. In *Proceedings of the IEEE symposium on security and privacy, 2016* (pp. 636–654).

Fernandes, E., Rahmati, A., Jung, J., & Prakash, A. (2017). Security implications of permission models in smart-home application frameworks. *IEEE Security & Privacy*, *15*(2), 24–30.

Flederer, F., Ostermayer, L., Seipel, D., & Montenegro, S. (2017). Source code verification for embedded systems using prolog. In *Proceedings 29th and 30th workshops on (constraint) logic programming and 24th international workshop on functional and (constraint) logic programming.*

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, *29*(7), 1645–1660.

Kiyavitskaya, N., Zeni, N., Mich, L., Cordy, J. R., & Mylopoulos, J. (2007). Annotating accommodation advertisements using cerno. In *Information and communication technologies in tourism 2007* (pp. 389–400). Springer.

Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of things (iot): A literature review. *Journal of Computer and Communications*, *3*(05), 164.

Marosi, A. C., Lovas, R., Kisari, Ã., & Simonyi, E. (2018). A novel iot platform for the era of connected cars. In *2018 IEEE international conference on future IoT technologies (future iot)* (p. 1-11). doi: 10.1109/FIOT.2018.8325597

Minerva, R., Biru, A., & Rotondi, D. (2015). Towards a definition of the internet of things (iot). *IEEE Internet Initiative*, *1*(1), 1–86.

Parekh, R., & Honavar, V. (2000). Grammar inference, automata induction, and language acquisition. *Handbook of natural language processing*, 727–764.

Pastor, O. (2017). Model-driven development in practice: From requirements to code. In B. Steffen, C. Baier, M. van den Brand, J. Eder, M. Hinchey, & T. Margaria (Eds.), *SOFSEM 2017: Theory and practice of computer science* (pp. 405–410). Cham: Springer International Publishing.

Rugaber, S., & Stirewalt, K. (2004). Model-driven reverse engineering. *IEEE Software*, *21*(4), 45-53. doi: 10.1109/MS.2004.23

SmartThings. (n.d.). *Anatomy and life cycle of a smartapp.* Retrieved from https://docs.smartthings.com/en/latest/smartapp-developers-guide/anatomy-and-life-cycle-of-a-smartapp.html (Last accessed on 25-1-2022)

SmartThings. (2019). *Hongtat/smartthings-capabilities: Smartthings capabilities.* Retrieved from https://github.com/

hongtat/smartthings-capabilities (Last accessed on 31-10-2021)

Soumyalatha, S. G. H. (2016). 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.*

Stallings, W., Brown, L., Bauer, M. D., & Bhattacharjee, A. K. (2012). *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA.

Tian, Y., Zhang, N., Lin, Y., Wang, X., Ur, B., Guo, X., & Tague, P. (2017). Smartauth: User-centered authorization for the internet of things. In *Proceedings of the 26th USENIX security symposium* (pp. 361–378).

Wortmann, F., & Flüchter, K. (2015). Internet of things. *Business & Information Systems Engineering*, *57*(3), 221–224.

Zhang, W., Meng, Y., Liu, Y., Zhang, X., Zhang, Y., & Zhu, H. (2018). Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM sigsac conference on computer and communications security* (pp. 1074–1088).

Zhou, W., Jia, Y., Yao, Y., Zhu, L., Guan, L., Mao, Y., … Zhang, Y. (2019). 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)* (pp. 1133–1150).

About the authors

Manar H. Alalfi is an Associate Professor at the Department of Computer Science, Toronto Metropolitan University, Toronto, and an Adjunct Assistant Professor at Queen's School of Computing, Canada. She is the director of the Creative Research in Security and Software Engineering Technology CRESSET lab. Her team conducts internationally recognized research in the area of Software Quality Assurance, Software Security and Vulnerability Analysis, Software Analytics and BigData, and Model Driven Engineering. She is a senior member of the IEEE and ACM, and a member of the Professional Engineers Ontario. You can contact the author at manar.alalfi@torontomu.ca.

Atheer Abu Zaid is an MSc graduate from the Department of Computer Science, Toronto Metropolitan University, Canada. Her current research interests are in the fields of software engineering and cyber security. Adapting software engineering techniques such as static analysis, to analyze Internet of Things applications for malicious behavior. You can contact the author at aabuzaid@torontomu.ca.

Ali Miri is a Full Professor at the Department of Computer Science, Toronto Metropolitan University, Toronto. His research interests include cloud computing and big data, computer networks, digital communication, and security and privacy technologies and their applications. He has authored and co-authored more than 200 referred articles, 6 books, and 8 patents in these fields. Dr. Miri has chaired over a dozen international conference and workshops, and had served on more than 100 technical program committees. He is a senior member of the IEEE, and a member of the Professional Engineers Ontario. You can contact the author at ali.miri@torontomu.ca.