A GQM Approach to Evaluation of the Quality of SmartThings Applications Using Static Analysis

Byeong-Mo Chang¹, Janine Cassandra Son¹ and Kwanghoon Choi²*
¹ Sookmyung Women’s University
Seoul, 04310, Republic of Korea
[e-mail: chang@sookmyung.ac.kr, janineson.it@gmail.com]
² Chonnam National University
Gwangju, 61186, Republic of Korea
[e-mail: kwanghoon.choi@jnu.ac.kr]
*Corresponding author: Kwanghoon Choi

Received March 10, 2019; revised February 24, 2020; revised March 25, 2020; accepted April 5, 2020; published June 30, 2020

Abstract

SmartThings is one of the most popular open platforms for home automation IoT solutions that allows users to create their own applications called SmartApps for personal use or for public distribution. The nature of openness demands high standards on the quality of SmartApps, but there have been few studies that have evaluated this thoroughly yet. As part of software quality practice, code reviews are responsible for detecting violations of coding standards and ensuring that best practices are followed. The purpose of this research is to propose systematically designed quality metrics under the well-known Goal/Question/Metric methodology and to evaluate the quality of SmartApps through automatic code reviews using a static analysis. We first organize our static analysis rules by following the GQM methodology, and then we apply the rules to real-world SmartApps to analyze and evaluate them. A study of 105 officially published and 74 community-created real-world SmartApps found a high ratio of violations in both types of SmartApps, and of all violations, security violations were most common. Our static analysis tool can effectively inspect reliability, maintainability, and security violations. The results of the automatic code review indicate the common violations among SmartApps.

Keywords: Quality, Evaluation, Static analysis, SmartThings, IoT applications

This work was supported by the National Research Foundation of Korea (NRF) grant funded by MSIP (No. 2017R1A2B4005138) and MoE (No. 2019R1H1A3A01058608).

http://doi.org/10.3837/tiis.2020.06.003

ISSN: 1976-7277
1. Introduction

There is a growing interest in the Internet of Things (IoT) application development for smart homes [1]. SmartThings is one of the most popular open platforms for home automation IoT where programmers have developed many applications, which are called SmartApps written in Groovy, allowing users to automate their homes controlling smart devices [2][3].

Modern IoT apps use many features including ones that may threaten users with security concerns. It has been reported that most SmartApps currently contain sensitive data flows [4] and violate security properties [5]. To allow the sharing of SmartApps on the open platform with no security concerns, it is important to maintain the quality of SmartApps at a high standard.

However, SmartApps used in our daily lives do not guarantee the SmartThings’ standards because they are not officially endorsed. There is no official quality review process about the software quality characteristics of SmartApps. Although there have been some analysis methods [2][4][5][6], they have focused only on particular security properties of SmartApps.

In this paper, we proposed a quality review method using static analysis for SmartApps on the home automation platform, SmartThings. First, the review method evaluates software quality characteristics defined by a widely accepted standard, such as ISO/IEC 25010. In particular, reliability, maintainability, and security are important characteristics in evaluating the qualities of SmartApps, as will be explained in detail later. Second, we aim at an automatic code review method because it is more efficient way of reviewing source code for potential vulnerabilities and non-compliance than manual code review. Automating the process through static analysis can significantly cut down efforts in the code review [7][8].

In the design of our review method, we adopted the well-known mechanism called Goal/Question/Metric (GQM) methodology [9][10] to justify the rationale of the review method by systematically defining and interpreting software measurements. First of all, following the GQM methodology, we set three goals of evaluating the three software quality characteristics of the ISO/IEC 25010 standard, developed a set of 12 questions defining and quantifying the specific goals, and set up 59 metrics collected from the code guidelines and best practices in the SmartThings developer documentation and from those for the Groovy programming language to answer the developed questions about the three goals.

Secondly, we automated the SmartApps code review using static analysis and used the analysis output to evaluate the quality of SmartApps in terms of the number of violations per lines of code called code defect density. Static analysis checks whether source code complies with the code review guidelines without running the programs. The 59 metrics collected in our GQM-based methodology were implemented as rules in CodeNarc [11], a rule-based static analysis tool for Groovy [12].

Finally, we evaluated the quality of SmartApps with 105 official and 74 community-created SmartApps from GitHub projects using our tool. A high ratio of violations was found in both the official and community-created SmartApps. This study shows that security defects, such as declaring unspecific subscriptions and web services-related violations, contributed the highest for both types of SmartApps. The maintainability defects found in community SmartApps are evidently higher than those found in official apps.

This article revises and extends earlier versions of this work [13][14] with the design of quality metrics under the GQM based methodology and the interpretation of the measurements.
As far as we know, this is the first systematically designed and automatic software review method for SmartApps that evaluates the three software qualities of ISO/IEC 25010 standard. Our contributions are summarized as follows.

- Software metrics for SmartApps are systematically designed by the GQM paradigm.
- A static analysis tool is implemented for automatically measuring the metrics.
- The official and community-created SmartApps are evaluated using this tool.

In Section 2, we review related work regarding static analysis and code review. Following an overview of SmartApps and static analysis in Section 3, we present a GQM-based methodology, including the research questions, in Section 4. We explain how to implement the collected metrics using CodeNarc in Section 5. We report the results of the analysis tool, relevant findings, and limitations of the study in Section 6. Finally, in Section 7, we conclude the study.

2. Related Work

In this section, we discuss previous studies regarding the use of static analysis tools automatically checking integrity of source code in software development. In addition, we discuss the role of static analysis in code review.

**Static analysis tools:** Various static analysis tools have been popular in software development. Checkstyle [15], PMD [16], and FindBugs [17] are notable among the existing popular Java static analysis tools. Checkstyle focuses on improving readability by supporting Google Java style guide and Sun code conventions, while PMD finds common programming mistakes, such as unused variables and unnecessary object creation, in the source code [18]. FindBugs is a static analysis tool for Java to catch vulnerabilities and bad practices. Fernandes et al. [2] developed a static analysis tool for SmartApps to uncover that 55% of SmartApps in the SmartThings marketplace are overprivileged because they declare event handlers to have more capabilities than necessary. Moreover, they found a security leak that grants a SmartApp full access to a device even if the app declares only limited access to the device. Our study used CodeNarc, which is customizable as its Java counterparts. CodeNarc is specific to the Groovy programming language. Panichella et al. [18] studied the use of the existing tools with default configurations as code reviews. Meanwhile, our research explored the custom rule capabilities of an existing static analysis tool to fit the research goal, which is to analyze SmartApps.

**Using static analysis for code review:** Code review is a systematic analysis of program code, and it corrects mistakes to improve the overall quality of software [18]. Developers do code review by manual review involving human code auditors or by an automated tool [19]. In both cases, code review requires a set of rules or the types of errors to look out for before the review can be performed [20]. The rules can be based on coding guidelines, if available, or design documents.

For popular programming languages, such as Java and PHP, there are vulnerable samples and applications available. We can use them as baseline corpora, which serve as a guide in understanding coding practices and patterns in those languages [21]. However, there are no formal secure coding guidelines specific to SmartThings for now. So, we collected the rules from the SmartThings guidelines and applicable rules from CodeNarc to implement a custom tool.
Researchers have studied how to facilitate the code review process by static analysis tools, which have been popular in software development to detect defects in the program. Panichella et al. [18] investigated how to exclude warnings detected by static analysis tools in the scope of code reviews on Java open-source projects. Results indicated that a high percentage of warnings were removed by analyzing specific categories of warnings. Projects using static analysis tools have been found to have fixed a higher percentage of warnings than other projects. It is relevant to our study because it suggests that static analysis tools can be used to support developers during code reviews. The study has proven that removing warnings during development can produce higher quality apps.

Other studies have focused on how static analysis tools reduce code review effort. Singh et al. [8] concluded that static analysis tools such as PMD reduce the workload of code reviewers by demonstrating that the warnings generated by the tool matched those suggested by the reviewer. Gomes et al. [20] explored the use of automated static analysis tools. They discussed how errors such as security vulnerabilities could be complicated and exist in hard-to-reach states. Static analysis tools can investigate these errors more easily because they do not require the code to be run. In our study, we also tried to capture potential security defects as well as other qualities such as reliability and maintainability.

Security analysis: For research to study security analysis on smart home applications based on SmartThings, Fernandes et al. [2] presented an in-depth empirical security analysis of the SmartThings platform to report two security flaws. First, in SmartThings’ privilege separation model, the flaws lead to significant overprivilege in SmartApps. Second, the event system does not provide sufficient protection for events that carry sensitive information. Cilik et al. [4] developed a static taint analysis tool called SAINT for IoT applications, such as SmartThings. Using it, they reported many sensitive data flows uncovered in SmartThings apps. Celik et al. [5] designed and implemented a system called SOTERIA for model-checking IoT applications. Given SmartApps and a property of interest, the system can automatically extract a state model and can apply model checking to find property violations. Manandhar et al. [22] proposed the notion of natural home automation scenarios for security analysis and designed an automatic method of generating such scenarios.

3. Background

3.1 SmartThings and SmartApps

![Fig. 1. SmartThings architecture](image-url)
The SmartThings architecture in Fig. 1 consists of the SmartThings cloud, hubs, devices, and the companion mobile app. The SmartThings cloud provides a secure execution environment for Groovy-based SmartThings applications called SmartApps. Hubs are connected via Wi-Fi or Ethernet to the SmartThings cloud, and devices are connected to the hubs at home via Z-Wave or ZigBee, which are low-powered wireless protocols. Sensor information in the devices is sent to the cloud via the hubs so that SmartApps can access the sensor values, and the SmartApps in the cloud can control the devices by sending commands via the hubs. The SmartThings mobile app provides users functions to install SmartApps on the cloud and to bind home devices to the device names declared in the SmartApps.

To introduce SmartApps, this section explains the features of SmartApps with an example. The source code structure of IoT applications is different from that of the standard application source code [23].

//definition
definition(
    name: "Lights-off-with-door-close App",
    namespace: "demo",
    author: "Demo User",
    description: "Turn a light on when a door opens and off when it closes.",
    category: "Utility"
)

//preferences
preferences {
    section("Select devices") {
        input "contact1", "capability.contactSensor", title: "Select contact sensor"
        input "light1", "capability.switch", title: "Select a light"
        input "lock1", "capability.lock", title: "Select a lock"
    }
}

//predefined callbacks
def installed() { initialize() }
def updated() { unsubscribe(); initialize() }
def initialize() {
    subscribe (contact1, "contact.open", openHandler);
    subscribe (contact1, "contact.closed", closedHandler)
}

event handlers
def openHandler(evt){ light1.on(); lock1.unlock() }
def closedHandler(evt){ light1.off() }

Listing 1. Typical SmartApp structure

SmartApp structure. The SmartApp in Listing 1 turns a light on when a door opens and turns it off when it closes. Listing 1 shows a typical SmartApp structure with four sections: definition, preferences, predefined callbacks, and event handlers [3]. The definition section has metadata about SmartApp itself. The preferences section defines pages, which are to be displayed on the companion mobile application, where users choose devices at install time for the SmartApp to use. SmartApps declare predefined callbacks installed and updated, which are invoked on installation and updates of the SmartApps. When the SmartApp in Listing 1 is installed, users are asked to bind real devices to the referred device names contact1, light1, and lock1.
Subscription model. It is through events that SmartApps interact with the devices on the SmartThings platform. SmartApps register event handlers for a device and receive from the device a stream of events. It operates using a subscription model that allows devices to subscribe to an event and takes action when the event happens. The predefined callbacks declare subscriptions with devices names, event types, and event handlers. In Listing 1, the event handlers `openHandler` and `closedHandler` are subscribed in `initialize()` to the device `contact1`. They are invoked on the events `contact.open` and `contact.close`, and they control the devices `light1` and `lock1`.

Sandboxed Groovy environment. SmartApps are developed using Groovy, a dynamically typed language on the Java platform [12] in the restricted SmartThings environment. Functions necessary for developing SmartApps are provided but some features are forbidden due to security and performance reasons. For example, SmartApps are not allowed to create new classes and to call certain methods [3].

External system access (Web service and other APIs). SmartApps may access web services hosted outside the SmartThings cloud. The SmartThings platform provides libraries for the purpose. In the opposite direction, SmartApps may serve as Web service endpoints to response to external requests. Third-party applications can do REST calls on the SmartApps to make use of the SmartApp functions, such as retrieving the device status [3].

The SmartThings web site provides a guide on how to develop SmartApps securely for public use through Code Review Guidelines and Best Practices [3]. It can be used as one of the criteria to evaluate user-submitted SmartApps. However, at the time of writing, the SmartThings web site states that there is no plan to review submissions for public distribution [3].

In the SmartThings guideline, there is a rule called Avoid Chained RunIn Call suggesting to avoid the use of the `runIn` method in scheduling a job. Listing 2 shows an example of violating this guide rule. This violation involves the use of the `runIn()` method, which executes a specified handler after a given number of seconds has elapsed. According to the guideline, chaining `runIn()` calls are vulnerable to failure because the failure of a scheduled execution in `handler()` breaks the whole chain. The guideline suggests to use a predefined scheduling function, such as `runEvery5Minutes()`, for a recurring schedule [3].

```groovy
def initialize() { runIn(60, handler) }
def handler() {
    // do something here
    // schedule to run again in one minute - this is an anti-pattern!
    runIn(60, handler)
}
```

Listing 2. Example of rule violation

3.2 Static Analysis Using CodeNarc

Static analysis examines source code without actual runs of the program, which is different from dynamic analysis testing software by running programs [24]. Static analysis is an effective method in code review or inspection. The term often refers to the analysis employed by an automated software tool [20]. Static analysis tools do not always result in real defects but they can report warnings to mark that a piece of code is vulnerable [25]. For example, these tools can detect issues, such as empty catch blocks, duplicate catch blocks, and uncaught exceptions [18].
When programmers make mistakes during software development, the compiler often detects them, and once the programmer fixes them, we continue the development process. However, it may be hard to discover some errors, especially those related to security, early in the development process. Defects can be discovered later after production, and are more expensive to correct [20]. This can be even more serious when security is breached, especially in smart home IoT applications. An advantage of the static analysis approach is that it can detect potential defects as early as possible in the development process. In contrast, dynamic analysis, such as testing, applies to code execution, so it can be done only in the late development phase. Thus a static analysis approach makes software development more stable and less vulnerable to errors in future tests [20].

Rule-based static analysis tools report warnings or violations in the source code whenever a portion of the source code matches a specific rule or pattern. Rules can be added or deleted easily according to the purpose of the analysis. It cannot, however, detect an error that does not match a fixed set of patterns or rules [26]. This is a disadvantage because if critical violations were missed, the tool would have no way to catch them.

Static analysis can automatically detect software problems before the code is released for public use. This study uses CodeNarc [11], a static analysis tool for analyzing Groovy code, which checks violations based on over 300 rules. The tool includes a set of preconfigured patterns or rules, but one can customize the tool for a specific project. For example, CodeNarc has the Basic category rules. These rules are designed to report unreachable code warnings, such as Empty else block and Dead code. CodeNarc provides mechanisms to detect code inconsistencies, helping developers to produce cohesive and maintainable code in a good design [27].

In this work, we have chosen CodeNarc because it is highly extensible. This is similar to the popular Java static analysis and bug finding tools, such as Checkstyle [15], PMD [16], and Find Bugs [17]. Also, many IDEs provide plugins for CodeNarc. For example, this research uses IntelliJ IDEA [28] and its CodeNarc plugin for evaluating the SmartApps after we add new rules to CodeNarc to check violations based on the SmartThings documentation.

4. Methodology

4.1 A GQM-Based Approach

The main purpose of this study is to design a method for evaluating the quality of smart home IoT applications on the SmartThings platform using static analysis. To accomplish this, we adopted the GQM paradigm [9] [10] in our research. The GQM paradigm is a well-known mechanism for defining and interpreting software measurements. Under the GQM paradigm, we first define a goal of interest. Second, we rephrase this goal into a form of questions. Third, we set up metrics that can answer these questions. Lastly, we collect data over the metrics to answer the questions and, subsequently, to see whether the goals are achieved.

4.1.1 Goal

Our goal is to evaluate three software qualities on SmartApps, reliability, maintainability, and security, with a particular emphasis on addressing the following research questions in the evaluation of the three software qualities of SmartApps:

- RQ1: What are the common violations found in SmartApps?
• RQ2: Are there any significant differences in quality between community-created SmartApps and official SmartApps?

The three software qualities were chosen from the standard system and software quality models called ISO/IEC 25010:2011 SQuaRE (Systems and Software Quality Requirements and Evaluation). The ISO/IEC SQuaRE is a standard evaluation model for software quality to formally identify quality characteristics related to software system requirements, design objectives, testing objectives, quality assurance, and so on [29]. They are categorized as the eight qualities, as shown in Table 1.

Table 1. Software qualities in ISO/IEC 25010:2011 and their inclusions for evaluation

| Software Quality | Description | Inclusion |
|------------------|-------------|-----------|
| Functional Suitability | Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions | Excluded |
| Performance Efficiency | Performance relative to the amount of resources used under stated conditions | Excluded |
| Compatibility | Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment | Excluded |
| Usability | Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use | Excluded |
| Reliability | Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time | Included |
| Security | Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization | Included |
| Maintainability | Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers | Included |
| Portability | Degree of effectiveness and efficiency with which a product or system can be transferred from one hardware, software or other operational or usage environment to another | Excluded |

However, because it is intended for general software, not all the qualities fit the purpose of evaluating SmartApp source code. Not all quality characteristics can be measured simply by analyzing the source code. We chose three quality characteristics matched with the key characteristics for SmartApps: reliability, security, and maintainability. Reliability is included for evaluating SmartApps because IoT applications should run well for a long time after deployment. Security is an important quality characteristic to protect information available at IoT environments such as in the home. Maintainability is a good criterion to see if SmartApps are easy to customize. Each home will have a different set of connected devices, so it must be easy to modify SmartApps to adapt them to each user's home.

We exclude the rest of the eight software quality characteristics from our goal for the reasons explained in the following. Functional suitability is excluded because the specification of each SmartApp under evaluation is unavailable, so it is hard to quantify this quality characteristic. Performance efficiency has never received much attention in the application domain of SmartApps because users typically use SmartApps connected with less than a dozen devices at home, so we decided not to use it for our evaluation. As to compatibility and portability, there is no sense in evaluating these characteristics because SmartApps run on a single platform, SmartThings. Usability is an important characteristic, but we excluded it from our list because what is good or bad for usability in the area of IoT applications has not yet been thoroughly discussed, which is contrary to usability in mobile...
applications. Moreover, it is not easy to create an automatic method to evaluate this characteristic.

4.1.2 Question

The goal to evaluate the reliability, maintainability, and security of SmartApps is now refined into two levels of questions, as shown in Table 2. Three questions at the high level are associated with the three software qualities. The description of the qualities in the ISO/IEC25010 standard in Table 1 leads us to make the three high-level questions. First, the frequency of faults in SmartApps will determine the degree to which the SmartApps perform their functions. Second, the ease of identifying styles, structure, and parts directly connects with how SmartApps can be modified effectively and efficiently for maintenance. Third, the possibility of vulnerabilities and attacks will affect how well SmartApps protect information and data. Our goal is thus refined to the three high-level questions in Table 2.

| Goal       | Question                                                                 | Sub-Questions                                                                 |
|------------|--------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Reliability| What is the frequency of faults?                                          | R1. Are null values handled properly?                                         |
|            |                                                                         | R2. Are there any potential mistakes/typos?                                  |
|            |                                                                         | R3. Are there busy loops?                                                    |
|            |                                                                         | R4. Are there missing cases?                                                  |
|            |                                                                         | R5. Are there any inconsistencies?                                           |
|            |                                                                         | R6. Are there any unused values/codes?                                       |
| Maintainability| How easy is it to identify styles, structure, and parts for maintenance? | M1. Is the logic too complex?                                                |
|            |                                                                         | M2. Is the code hard to read?                                                |
| Security   | What are the possibility of vulnerabilities and attacks?                 | S1. Are there hard-coded values?                                             |
|            |                                                                         | S2. Are there external HTTP requests?                                        |
|            |                                                                         | S3. Is it not easy to predict the behavior?                                  |
|            |                                                                         | S4. Are there any uses of unrestricted things?                               |

The three high-level questions are too general, so we need to make them specific to the context of SmartApps on the SmartThings platform. We have, therefore, derived low-level sub-questions from each of the high-level questions, as shown in the sub-questions column of Table 2. The derivation is largely based on experiences with SmartApps and software qualities by the authors. The first high-level question on the frequency of faults to evaluate the reliability of SmartApps is refined into six sub-questions. They are numbered with a prefix $R$ for reference.

$R1$ is a sub-question on whether or not null values are handled properly. SmartApps are written in Groovy, which is a programming language for the Java platform, and NullPointerExceptions are one of the most frequently occurring exceptions on the SmartThings platform. $R2$ asks if there are any potential mistakes or typos. For example, when an assignment operator is used in a conditional expression that is likely to be mistaken for an equality operator. $R3$ is a question about the presence of any busy loops. Event handlers, which are the main structure of SmartApps, are called by the SmartThings platform, and so a busy loop can be rewritten with an event handler. $R4$ checks if there are any potential faults due to missing cases, such as missing event handlers. SmartApps are vulnerable if an undefined event handler is called. $R5$ is an investigation of any inconsistencies. For example, Groovy is a dynamically typed programming language that does not require a method
signature, but the method should be written to return a single type of data. R6 attempts to identify unused objects, unused arrays, dead code, and so on.

For the second high-level question on evaluating the ease of identifying styles, structure, and parts for maintenance, we refine it into two sub-questions, as shown in Table 2. M1 asks if the logic implemented in SmartApps is complex, for example, due to high cyclomatic complexity, which counts the number of independent paths in the source code where each path has at least one edge not in one of the other paths. M2 is a sub-question on readability, such as how big the size of a method is, how many methods are in a class, and so on.

For the third high-level question on the potential vulnerabilities and attacks, we list four sub-questions in Table 2. S1 examines issues related to security because hard-coded values, such as phone numbers, can be dangerous to the system if users cannot update a wrong value through the SmartApps using the mobile app. This suggests a safe and proper SmartApp implementation with contact inputs, so they can be validated and updated. S2 is a sub-question on potential vulnerabilities caused by the HTTP protocol. SmartApps can make HTTP requests to outside services, and they can run as an HTTP endpoint to serve requests from the outside. S3 examines whether the execution flow of SmartApps should be made explicit. For example, the flow of the event from devices to handlers is made obvious by explicit method names to subscribe to an event. Also, using dynamic method execution should be avoided as much as possible. S4 is a sub-question on the use of unrestricted things. Although SmartApps are written in Groovy, there are some restrictions in the use of full features of the programming language.

4.1.3 Metric

Now it remains to define metrics to answer all (sub-)questions in Table 2. Our strategy starts with the well-established code review guidelines for SmartApps and the Groovy programming language. Then we collected some review guidelines that can be used to answer one of the 12 (sub-)questions explained previously. A guideline can be found in the developer documentation at the official SmartThings website [3]. Some examples of the guidelines include Avoid recurring short schedules and Do not use dynamic method execution. We chose them as our metrics because they can be used to answer the questions R3 and S3, respectively. Other guidelines, such as Code should be readable and Comment appropriately, are also part of the code review. However, the guidelines are too general and require more information about what readable code should be and what is an appropriate comment. Without concrete patterns to check in the source code, it can be hard to implement with static analysis; thus, they are not chosen as our metrics and are simply ignored.

The other guidelines come from CodeNarc, an open-source static analysis tool for Groovy [11]. Because SmartApps are written in the Groovy programming language, we refer to the rules implemented in CodeNarc to look for our metrics. It consists of 357 rules targeted to general Groovy source code. The categories include basic, convention, design, security, formatting rules, and more. Due to the restricted, sandboxed nature of SmartApps, most of the rules in CodeNarc do not apply to the 12 (sub-)questions. So, our selection incorporates only those that are applicable to SmartThings and those that can answer the questions in the GQM-based methodology.

To summarize, 59 metrics in total were collected to answer all the 12 (sub-)questions for our GQM-based methodology. 20 out of 30 guidelines from the SmartThings code review guidelines were chosen as new metrics. Meanwhile, 38 out of 357 default CodeNarc rules were selected, as they are applicable to SmartApps. An additional metric was also included to
count the lines of code (LOC). Table 3 shows a list of metrics and their association with the goals and sub-questions. The table also shows the source of the metrics. The sources spread from SmartThings best practices to various guidelines, such as basic or general rules, conventions, security considerations, size, and documentation. Refer to a full list and detailed descriptions of all the metrics in the extended version of this paper [31].

After we introduce metrics for each of the sub-questions, a measurement experiment will count the frequency of the faults, complexity, and vulnerabilities to answer all the associated sub-questions immediately. Eventually, it will answer the three high-level questions to achieve our goal of evaluating SmartApps in terms of reliability, maintainability, and security.

Although the GQM-based approach is thus advantageous for organizing the measurement from a top-down perspective, there is still a problem with how to interpret the measurement results related to the software quality of SmartApps. It is not easy to justify setting an absolute software quality index to determine whether or not the software quality of a SmartApp is good. It is also not possible to make a relative comparison between a new measurement result and an old one. This is because SmartThings is a new home IoT platform, and as yet, it has no previously accumulated results of the measurements.

Table 3. Metrics and their static analysis rules (excerpt from the extended version [31])

| Goal       | Q#  | Metric                          | Rule         | Source          |
|------------|-----|---------------------------------|--------------|-----------------|
| Reliability| R1  | Handle null values              | Custom rule  | Best practices  |
| Reliability| R2  | Assignment in conditional       | Default rule | Basic           |
| Reliability| R3  | Avoid chained runIn() calls     | Custom rule  | Best practices  |
| Reliability| R4  | Missing event handler           | Custom rule  | Basic           |
| Reliability| R5  | Use consistent return values    | Custom rule  | Best practices  |
| Reliability| R6  | Dead code                       | Default rule | Basic           |
| Maintainability| M1  | Confusing ternary               | Default rule | Convention      |
| Maintainability| M1  | If statement could be ternary   | Default rule | Convention      |
| Maintainability| M1  | Cyclomatic complexity           | Default rule | Size            |
| Maintainability| M2  | For loop should be while loop   | Default rule | Basic           |
| Maintainability| M2  | Could be Elvis                  | Default rule | Convention      |
| Maintainability| M2  | Nested block depth              | Default rule | Size            |
| Maintainability| M2  | Separate parent and child apps  | Custom rule  | Best practices  |
| Security    | S1  | Do not hard-code SMS messages   | Custom rule  | Security considerations |
| Security    | S2  | Document external HTTP requests  | Custom rule  | Documentation   |
| Security    | S2  | Document exposed endpoints      | Custom rule  | Documentation   |
| Security    | S3  | Do not use dynamic method exec. | Custom rule  | Security considerations |
| Security    | S3  | Subscriptions should be clear   | Custom rule  | Security considerations |
| Security    | S3  | Subscriptions should be specific| Custom rule  | Security considerations |
| Security    | S4  | Do not use restricted method calls | Custom rule  | Security considerations |

What we can rely on at the moment is to have two groups of SmartApps, one group of official ones and another group of community-created ones, and then to interpret the results of the measurements relatively by comparing the two groups based on common characteristics and different ones.

The purpose of the two research questions (RQ1 and RQ2) in our goal is for this proper interpretation of the measured results from the aforementioned GQM-based methodology. The first research question intends to draw the common characteristics between the official
SmartApps and the community-created SmartApps. The second research question is to contrast the two kinds of SmartApps.

4.2 Static Analysis for GQM-Based Measurement

To measure the metrics in our GQM-based methodology automatically, a code review tool was developed to detect violations and evaluate the qualities of SmartApps. We used CodeNarc, an open-source static analysis tool for Groovy, because SmartApps are written in the Groovy programming language. The tool can be customized where users can select or remove rules according to the specific context. In addition, users can write their own rules, which will be done in this study. We selected CodeNarc because of its extensibility and its capability of analyzing multiple apps at once to automate and streamline the code review process.

We have set up 59 static analysis rules in such a way that every rule corresponds directly to one of the 59 metrics in the GQM-based methodology. Of all the rules, we have implemented 21 static analysis rules in the static analysis tool for the 20 guidelines from the SmartThings code review guidelines, and one more extra rule for counting LOC. The rest of the metrics come from CodeNarc rules, and we have adopted the 38 static analysis rules for our measurements. In Table 3, those metrics marked by ‘custom rule’ are what we have implemented, and the other metrics marked by ‘default rule’ are what we have adopted from CodeNarc.

Ideally, everything susceptible to review guidelines (code, design, specifications) that is within the capabilities of static analysis must be included to help to improve the quality of the program as much as possible. However, the static analysis tool cannot always adopt all the guidelines. For example, a rule named Document external HTTP requests is partially supported by the tool. This is because although it is possible for static analysis to detect HTTP external calls, we cannot make sure that the detected calls are a real threat. In such a situation, human reviewers are recommended to do code inspection manually because uncovering the intention or reason that the external request is made is beyond the capability of static analysis. Based on the guideline only, we do not know whether a given endpoint is safe. This limitation can be overcome if there is more information available about safe or unsafe endpoints [21]. We do not consider such an enhancement and leave it to future work.

Fig. 2 shows a proposed flow of the automatic code review and evaluation under the GQM-based metrics for quality of SmartApps. First, we gathered the SmartApps and configured the tool according to the code review rules. Then, analysis was performed through CodeNarc generating measurements in the CodeNarc report. Finally, we analyzed the summary of the defects found in the CodeNarc report. Our tool displayed defect density score for each official SmartApp, and identified the most common violation among the SmartApps.
The tool repeats the same flow for community-created SmartApps. Then, we compared the two reports to analyze their similarities and differences in quality characteristics.

5. Implementation

In this section, we describe the implementation of our static analysis tool for measurement and evaluation on the SmartApp qualities. Our tool is available as open-source software that can be freely downloaded.

- The new custom static analysis rules: https://github.com/janineson/CodenarcPluginFiles
- The default static analysis rules from CodeNarc Ver. 0.20: https://github.com/janineson/Codenarc

5.1 Writing New Static Analysis Rules

In this research, we use both the CodeNarc default rules and new custom rules. We employed 59 rules in total to evaluate the three SmartApp quality attributes under our GQM-based methodology (see Table 3). We adopted 38 static analysis rules available at CodeNarc and developed 21 new static analysis rules based on the code review guidelines. Table 3 displays some static analysis rules for SmartApps. The other rules are available in the extended version [31]. Some static analysis rules, such as Cyclomatic Complexity, require thresholds to be defined. The default values set in CodeNarc were adopted, and they can be found in the repository of our tool. The detailed description of the metrics in the extended version [31] specifies the default values.

In CodeNarc, it is easy to create new rules. We briefly explain a procedure of creating a new rule. Once we define a new rule name using the command-line program, CodeNarc will create Groovy rule files automatically. We edit the generated rule files to add some code that analyzes if there is a violation. CodeNarc also provides a convenient framework to test if the rule is working or not. We can write unit test files for that particular rule.

CodeNarc uses abstract syntax tree (AST) traversal for static analysis to examine the code structure and to check violations without running the program. As an example, Listing 3 shows a piece of code that analyzes if the subscription is clear. An unclear subscription may use a string variable instead of explicitly stating the attribute (e.g., 'contact.open'). The example code overrides the AST visitor method visitMethodCallExpression to visit all method calls with the method name 'subscribe'. The second argument of the subscribe method call is the attribute of event type that the user wants to subscribe to. So, it is a violation if a variable instance appears as the second argument of the method.

```java
class ClearSubscriptionAstVisitor extends AbstractAstVisitor {
    @Override
    void visitMethodCallExpression(MethodCallExpression call){
        if(AstUtil.isMethodNamed(call, 'subscribe', 3)) {
            def attributeName = call.arguments.expressions[1]
            if (!(attributeName instanceof ConstantExpression))
                addViolation(call, 'Subscriptions should be clear.')
        }
        super.visitMethodCallExpression(call)
    }
}
```

Listing 3. CodeNarc custom rule ClearSubscription for the metric 'Subscriptions should be clear'
5.2 Code Defect Density

After writing the new rules, the tool must be set up to include the new rule set. The new rules written under the SmartThings guidelines are named as custom rules. The inclusion of the new rules can allow the static analysis tool to detect SmartApp specific violations more precisely [32].

Once CodeNarc retrofitted with the new rules analyzes SmartApps, it produces a detailed defect report. Developers refer to the report to improve the SmartApp source code.

The number of defects or the violation count found by the rules in CodeNarc is divided over the LOC, then multiplied by 1000, because we want to display output based on KLOC. The industry average is known to be about 15 – 50 defects per KLOC, which are the number of errors per 1000 lines, of delivered code [33]. However, this research will adopt the defect density, which is computed as:

\[
\text{code defect density} = \frac{\# \text{ of defects}}{\text{lines of code}} \times 1000
\]

where the number of defects is the number of violations discovered by the tool.

We can evaluate the quality of SmartApps based on the total code defect density for each of reliability, maintainability, and security. By the defect density score, we can now quantify the quality of apps. So, we can compare two SmartApps in terms of the three types of quality, and we can analyze which type of quality is better than the others within a SmartApp.

6. Analysis of Results

In this section, we report the analysis results of the study to answer the two research questions in Section 4. The raw data from our analysis and evaluation can be freely downloaded from

- https://github.com/janineson/SmartThings_AutomaticCodeReviewEvaluationTool

At the website, the static analysis reports and the evaluation results are as follows:

- Analysis report: CodeNarcAntReport-official.html, CodeNarcAntReport-cc.html
- Evaluation result: codereviewout-Official.txt and codereviewout-cc.txt

To analyze SmartApps through static analysis, it is necessary to have the source code of the apps. We gathered all SmartApps (referred to as population) from a public GitHub repository [34], which consists of 105 official and 74 community-created apps. In total, 179 apps were analyzed. They were all SmartApps available at the SmartThings developer site at the time of our experiment. A snapshot of all the SmartApps used for our analysis and evaluation is copied into the following repository for reproduction.

- SmartApps source code: https://github.com/janineson/SmartThingsPublic

In this study, we differentiated the SmartApps into two types: official and community-created. The official SmartApps are already in production and available in the SmartThings marketplace. The community-created SmartApps are publicly distributed apps that are shared among users. However, the community-created SmartApps are not guaranteed to be in high quality because we are not sure if they pass any code review or inspection, such as one performed by SmartThings. Therefore, it can be dangerous to use these apps.

In the following, we will explain the analysis report and evaluation results over all the official and community-created SmartApps in detail.
6.1 RQ1: What are the Common Violations found in SmartApps?

Table 4 gives a summary of the analyzed SmartApps, where 44.8% of official and 77.0% of community-created apps have violations of the rules defined in the static analysis tool. The number of violated rules was 25 for both sets.

Table 4. Summary of analyzed SmartApps

| SmartApp          | # Total Analyzed | # With Violations(%) | # Violated Rules |
|-------------------|------------------|----------------------|------------------|
| Official          | 105              | 47 (44.8%)           | 25               |
| Community-created | 74               | 57 (77.0%)           | 25               |

Table 5 reports the rule violations that the static analysis tool detected among 105 official and 74 community-created SmartApps. We calculated the percentage of SmartApps with common rule violations for each set. The most common violation for all SmartApps was *Subscriptions should be specific*. It was found in 42% of community-created apps and was also found in 23% of official apps.

Table 5. Rule violations showing the percentage of occurrences in both sets of apps

| Rule                           | Quality Attribute | % Community-created | % Official |
|--------------------------------|-------------------|---------------------|-----------|
| Subscriptions should be specific | Security         | 42                  | 23        |
| Document exposed endpoints     | Security         | 24                  | 10        |
| Inverted if-else               | Maintainability   | 20                  | 11        |
| Document external HTTP requests| Security         | 20                  | 10        |
| Cyclomatic complexity          | Maintainability   | 18                  | 4         |
| Method count                   | Reliability       | 11                  | 10        |
| Could be Elvis                  | Maintainability   | 11                  | 8         |
| Use consistent return values   | Reliability       | 3                   | 10        |
| Missing switch default         | Reliability       | 9                   | 5         |
| Empty method                   | Reliability       | 9                   | 5         |
| Method size                    | Reliability       | 9                   | 1         |
| Avoid recurring short schedules| Reliability       | 9                   | 1         |
| Correct use of atomic state    | Reliability       | 8                   | 1         |
| Handle null values             | Reliability       | 7                   | 3         |
| Nested block depth             | Maintainability   | 7                   | 2         |
| Ahe metric                     | Maintainability   | 5                   | 3         |
| If statement could be ternary  | Maintainability   | 4                   | 4         |
| Verify array index             | Reliability       | 4                   | 4         |
| Parameter reassignment         | Reliability       | 3                   | 4         |
| Confusing ternary              | Maintainability   | 4                   | 2         |
| Correct use of atomic state update | Reliability   | 4                   | 1         |
| Subscriptions should be clear  | Security         | 1                   | 3         |
| Constant if expression         | Reliability       | 0                   | 2         |
| Empty else block               | Reliability       | 1                   | 1         |
| Empty if statement             | Reliability       | 1                   | 1         |
| Dead code                      | Reliability       | 1                   | 1         |
| Empty catch block              | Reliability       | 0                   | 1         |

Results show that most of the violations appeared in both sets of SmartApps albeit in a slightly different order. Based on the results of the population (all apps) used in this study, the
SmartApp violations affected all three quality attributes used in the study, with security having the highest impact. For the majority of the rules, the percentage of official apps that violated them was less than for community-created apps, as shown in Fig. 3. This is because the SmartThings team did reviews and tests before the apps were published in the marketplace. On the other hand, *Use consistent return values* appeared more frequently in official apps. It was also observed that *Constant if expression* and *Empty catch block* violations were found only in official apps.

The top five most common violations we found were concerned with the security and maintainability of the code. It is interesting to find that violations were still detected from official published apps that had gone through the SmartThings code review. The reviewers may have missed or may have disregarded the violations.

We list the five most common violations with descriptions in the following:

*Subscriptions should be specific.* A SmartApp that is subscribed to many event types will execute excessively. Extensive subscriptions can lead to security issues in that they can inadvertently allow to perform unnecessary tasks. The best practice is to subscribe to only the event of interest. An example violating this rule is an app that subscribes to a presence sensor but uses only one of its events in the sensor device. The entire event types of *presence* is specified in the subscription instead of the specific event *presence.present*.

*Document exposed endpoints.* Every endpoint should have a proper documentation to describe the APIs and the data accessible through those APIs. The automatic code review tool reports this because of potential security risks. This involves remote access where the SmartApp acts as a web service to receive requests from the outside. The use of a SmartApp as a web service expands the attack surface through which malicious attackers can exploit the user's home devices [35].

*Inverted if-else.* This belongs to a CodeNarc default convention rule related to the readability of the source code. Violations of rules in this category do not always mean that there is a problem in the code. Instead, it suggests improvement in terms of code maintenance in order to make the code easier to understand by following coding conventions [18].

*Document external HTTP requests.* When SmartApps have HTTP requests to external services, there should be a documentation on their purpose. The automatic code review tool reports this because of possible security issues. The HTTP requests are a remote access that
can send private data to a remote server. Privacy can be violated especially if the user does not know the destination of data or how the data is used there [35].

Cyclomatic complexity. CodeNarc reports this size-related metric. Although to judge whether the source code is too complex requires a certain standard, it is difficult to understand code with high cyclomatic complexity, indicating that it is more likely to have defects.

6.2 RQ2: Are There Any Significant Differences in Quality between Community-Created SmartApps and Official SmartApps?

Table 6 reports on the analysis results. As explained previously, the report also revealed that security-related issues were the top contributors to defects in SmartApps. To compare the difference of the average code defect densities of both sets, we used a statistical test called Welch's t-test. This is introduced to test the hypothesis that two populations from official SmartApps and community-created SmartApps have equal means. This statistical test is known to be more reliable for two samples with unequal variances or with unequal sample sizes. Fig. 4 shows a two-sample Welch's t-test using T distribution with the sample mean, the sample standard deviation, and the sample size of both sets of SmartApps. The T distribution graph for each quality attribute is shown. The graph indicates that the null hypothesis for this test, which states that both sets have the same average code defect density, is rejected. This means that the two sets of apps are found to be different with regard to reliability, maintainability, and security.

Table 6. Code defect density (defects per KLOC)

| SmartApp      | Measure       | Security | Reliability | Maintainability |
|---------------|---------------|----------|-------------|----------------|
| Official      | Mean          | 6.83     | 2.21        | 1.75           |
|               | Standard Deviation | 13.90   | 5.05        | 4.26           |
| Community-created | Mean          | 11.78    | 7.95        | 4.14           |
|               | Standard Deviation | 15.49   | 17.46       | 8.24           |

Fig. 4. Two-sample Welch's T-test (from left to right: reliability, maintainability, security)

Table 7 shows the code defect densities of official apps, and Table 8 shows that of community-created apps, along with the LOC and the number of inputs and subscriptions. Although SmartApps typically declare devices and other constants as input, we observed that some SmartApps declare no inputs and subscribe no event handlers (refer to Table 7, wattrvision-manager.groovy example). These SmartApps provide web services exposing an endpoint, to which external entities can request API calls to retrieve information and control the devices. Both tables show that the number of input devices tends to be greater than the number of subscriptions. This is because some of the devices have event handler subscriptions while others only respond to action commands. However, it is also observed that the number of input devices is less than the number of subscriptions. For an input device that supports
multiple event types, more than one event handler subscription can be associated with a single device.

Official SmartApps can be considered to be mature projects because they are already deployed and used in operation. The automatic code review tool produced a consistent report that official SmartApps have higher quality in maintainability and reliability. This is because both the maintainability and reliability defect density score of the top 20 official apps in Table 7 are significantly lower than those of the top 20 community-created apps in Table 8. Note that the defect density scores of all the apps are available in the extended version [31]. This implies that official apps are most likely following the Groovy convention rules on readability of code. Meanwhile, we found that most of the community-created apps are not likely following the convention rules very much; therefore, it is hard to read and maintain the code.

There are noticeable gaps between security, reliability, and maintainability defect density scores in community-created SmartApps. Even though the ranks of the qualities are the same for both official SmartApps and community-created apps, the defect density scores in community-created apps are greater in all the three aspects. The community-created apps need to improve the quality, especially in terms of the security and reliability of the source code.

The high defect density score results from several factors. The formula for calculating the score is the ratio of the defects against LOC. SmartApps with fewer LOC tend to have fewer defects than longer SmartApps. Based on the set of SmartApps we analyzed, the typical size of a SmartApp code base is less than 200 lines. Ideally, the defect density score should be close to zero, which means there are no defects found. Some official SmartApps were observed to only contain security violations greatly influencing its ranking as the highest among the three qualities.

One SmartApp with a high total defect density in Table 7, only showed security violations, but because of its low LOC count, the net defect density score was very high. Another factor may be due to the frequency of the same type of error in an app. The same type of error is found multiple times in the app, and it can greatly affect the defect density.

In this study, we define defects as non-compliance with the SmartThings guidelines and best practices. They can be warnings of vulnerabilities that can potentially arise if the guidelines and universal programming standards are not followed. The reviews and tests conducted by SmartThings imply that officially published apps function normally in an expected way. However, according to our results in RQ1, which showed that 23% of official apps incurred security defects related to unspecific subscriptions, official apps may show unintended behavior that the apps do not explicitly declare, causing problems. In a more complex system with many IoT devices interconnected with another, the failure or malfunction probability will rise if there are defects in the code.

| SmartApp                        | LOC | Input | Subscr. | Reli. | Main. | Secu. | Total  |
|---------------------------------|-----|-------|---------|-------|-------|-------|--------|
| presence-change-push.groovy     | 28  | 1     | 1       | 0     | 0     | 71.43 | 71.43  |
| let-there-be-light.groovy       | 32  | 2     | 1       | 0     | 0     | 62.5  | 62.5   |
| sleepy-time.groovy              | 56  | 3     | 1       | 17.86 | 0     | 35.71 | 53.57  |
| turn-it-on-when-im-here.groovy  | 42  | 2     | 1       | 0     | 0     | 47.62 | 47.62  |
| presence-change-text.groovy     | 43  | 2     | 1       | 0     | 0     | 46.51 | 46.51  |
| wattvision-manager.groovy       | 312 | 5     | 1       | 27.4  | 13.7  | 0     | 41.1   |
| energy-alerts.groovy           | 73  | 5     | 1       | 25.64 | 9.62  | 9.62  | 44.87  |
| turn-on-if-i-arrive-after-sunset.groovy | 50  | 2     | 1       | 0     | 0     | 40    | 40     |
| light-follows-me.groovy        | 52  | 3     | 1       | 0     | 0     | 38.46 | 38.46  |
The results also indicate that the SmartApp community developers can improve the quality of code if they followed the SmartThings guidelines and the coding conventions before they publish their code. Indeed, the open development community is very beneficial to the problem of identifying problematic code vulnerable to threats, so the review will not miss the evaluation of both kinds of SmartApps, not all defects should be manually checked. The tool claimed in previous studies, [20][32]. Nonetheless, the automatic static analysis tool must be a good pre-stage to a review because, judging from the high defect density scores detected in the SmartThings ecosystem, but certain countermeasures, such as automatic code review, must be supported for publishing high-quality apps. Also, this study suggests that static code analysis is not a replacement for human-led code reviews, but rather a supplement to them, as has been identified in previous studies. [20][32]. Nonetheless, the automatic static analysis tool must be a good pre-stage to a review because, judging from the high defect density scores detected in the evaluation of both kinds of SmartApps, not all defects should be manually checked. The tool can identify problematic code vulnerable to a threat, so the review will not miss the problematic code [25].

Table 8. Community-created SmartApp code defect densities (excerpt from the extended version [31])

| SmartApp                          | LOC | Input | Subscr. | Reli. | Main. | Secu. | Total |
|-----------------------------------|-----|-------|---------|-------|-------|-------|-------|
| smart-energy-service.groovy       | 523 | 5     | 0       | 99.43 | 17.21 | 5.74  | 122.37|
| spruce-scheduler.groovy           | 2139| 30    | 16      | 66.39 | 18.23 | 1.87  | 86.49 |
| initial-state-event-streamer.groovy| 304 | 26    | 38      | 6.58  | 6.58  | 69.08 | 82.24 |
| lights-off-with-no-motion-and-presence.groovy| 66 | 4 | 2 | 0 | 15.15 | 60.61 | 75.76 |
| let-there-be-dark.groovy          | 34  | 2     | 1       | 0     | 0     | 58.82 | 58.82 |
| smart-alarm.groovy                | 1496| 58    | 10      | 54.14 | 1.34  | 0.67  | 56.82 |
| tcp-bulbs-connect.groovy          | 461 | 3     | 0       | 47.72 | 4.34  | 2.17  | 54.23 |
| turn-off-with-motion.groovy       | 58  | 3     | 1       | 0     | 17.24 | 34.48 | 51.72 |
| gideon.groovy                    | 199 | 6     | 2       | 0     | 40.2  | 10.05 | 50.25 |
| obything-music-connect.groovy     | 40  | 1     | 0       | 50    | 0     | 0     | 50    |
| curb-control.groovy               | 102 | 1     | 0       | 19.61 | 19.61 | 9.8   | 49.02 |
| plantlink-connector.groovy        | 366 | 1     | 2       | 2.73  | 21.86 | 21.86 | 46.45 |
| switch-activates-home-phrase.groovy| 44 | 3    | 1       | 0     | 0     | 45.45 | 45.45 |
| smart-light-timer-x-minutes-unless-already-on.groovy| 110 | 4 | 3 | 2.73 | 21.86 | 21.86 | 46.45 |
| door-lock-code-distress-message.groovy| 44 | 5 | 1 | 0 | 0 | 45.45 | 45.45 |
| shabbat-and-holiday-modes.groovy  | 134 | 4     | 0       | 37.31 | 0     | 7.46  | 44.77 |
| netatmo-connect.groovy            | 452 | 2     | 0       | 28.76 | 6.64  | 8.85  | 44.25 |
| switch-changes-mode.groovy        | 46  | 1     | 1       | 0     | 0     | 43.48 | 43.48 |
| smart-windows.groovy              | 116 | 8     | 1       | 0     | 34.48 | 0     | 34.48 |
| lighting-director.groovy          | 1074| 64    | 28      | 5.59  | 12.1  | 14.9  | 32.59 |
6.3 Limitations

The automatic code review tool used in the study treats all external HTTP calls as violations because they can be a threat to the system. However, not all HTTP requests and exposed endpoints are harmful. The current design may lead to false positives in the results. The number of web service SmartApps in the population of the analyzed apps has significantly affected the code review evaluation outcome. SmartThings has provided security measures in the form of OAuth to authenticate these requests. However, this study is limited to implementing the guidelines into rules to automate the code review and thus, all instances of HTTP requests and endpoint exposures are tagged. We plan to explore this limitation in future work by incorporating other analysis techniques [4][5][36][37] focused more on security.

7. Conclusion

This study proposed an automatic code review tool using static analysis with quality evaluation metrics systematically designed under GQM methodology, evaluating the three software quality characteristics of the ISO/IEC52060 standard for SmartApps. To the best of our knowledge, this is the first automatic code review tool specific for applications on the SmartThings platform. Throughout this study, we found a high ratio of violations in both the official and community-created SmartApps. This study also indicated that security defects are the most common one for both types of SmartApps. Maintainability defects were noticeably more prevalent in community-created SmartApps than in official SmartApps; therefore, they need to improve readability of the source code more.

The proposed tool will be useful for an official code review process to endorse SmartApps before users deploy them at homes, and it guarantees the quality of SmartApps in the SmartThings’ standards helping users to mitigate security threats. The tool can also reduce the review cost by automatically precluding low-quality SmartApps from the manual review.

As future work, we plan to perform an in-depth analysis on how to evaluate the qualities of SmartApps with external services. Certain SmartApps make use of external web services that involve sharing of data to third-party domains to which the current tool can only flag warnings that may pose security threats without confirming.

References

[1] Y. Yang, L. Wu, G. Yin, L. Li, H. Zhao, “A survey on security and privacy issues in Internet-of-Things,” IEEE Internet of Things Journal, vol. 4, no. 5, pp. 1250-1258, October, 2017. Article (CrossRef Link).
[2] E. Fernandes, J. Jung, A. Prakash, “Security analysis of emerging smart home applications,” in Proc. of IEEE Symposium on Security and Privacy, pp. 636-654, May 22-26, 2016. Article (CrossRef Link).
[3] “Smartthings developer documentation”, 2017. [Online]. Available: http://docs.smartthings.com
[4] Z. B. Celik, L. Babun, A. K. Sikder, H. AkSU, G. Tan, P. D. McDaniel, A. S. Uluagac, “Sensitive information tracking in commodity iot,” in Proc. of 27th USENIX Security Symposium, Baltimore, MD, USA, pp. 1687-1704, August 15-17, 2018. Article (CrossRef Link).
[5] Z. B. Celik, P. D. McDaniel, G. Tan, “Soteria: automated iot safety and security analysis,” in Proc. of 2018 USENIX Annual Technical Conference, Baltimore, MD, USA, July, pp. 147-158, 2018. Article (CrossRef Link).
[6] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, Patrick McDaniel, “Program analysis of commodity iot applications for security and privacy: challenges and opportunities,” ACM Computing Survey, vol.52, no.4, article 74, August, 2019. Article (CrossRef Link).

[7] R. M. Hartog, “Octopull: Integrating static analysis with code reviews,” Master’s thesis, Delft University of Technology, the Netherlands, December 16, 2015. Article (CrossRef Link).

[8] D. Singh, V. R. Sekar, K. T. Stolee, B. Johnson, “Evaluating how static analysis tools can reduce code review effort,” in Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 101-105, October 11-14, 2017. Article (CrossRef Link).

[9] V. R. Basili, “Software modeling and measurement: the goal/question/metric paradigm,” Tech. report, College Park, MD, USA, September, 1992. Article (CrossRef Link).

[10] R. van Solingen, E. Berghout, “Integrating goal-oriented measurement in industrial software engineering: industrial experiences with and additions to the goal/question/metric method (gqm),” in Proc. of 7th Int’l Software Metrics Symposium, pp. 246-258, April 4-6, 2001. Article (CrossRef Link).

[11] CodeNarc, 2018. [Online]. Available: http://codenarc.sourceforge.net

[12] Groovy, 2003. [Online]. Available: http://groovy-lang.org

[13] Janine Cassandra Son, “Automatic code review for SmartThings applications using static analysis,” Master thesis, Sookmyung Women’s University, June, 2018. Article (CrossRef Link).

[14] Janine Casandra Son, Byeong-Mo Chang, Kwanghoon Choi, “Automatic code review for SmartThings application using static analysis,” in Proc. of Korea Software Congress (KSC2017), Busan, pp.513-515, December 20-22, 2017. Article (CrossRef Link).

[15] Checkstyle, 2001. [Online] Available: http://checkstyle.sourceforge.net

[16] PMD, 2017. [Online] Available: https://pmd.github.io

[17] FindBugs, 2015. [Online] Available: http://findbugs.sourceforge.net

[18] S. Panichella, V. Arnaoudova, M. D. Penta, G. Antoniol, “Would static analysis tools help developers with code reviews?,” in Proc. of IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 161-170, March 2-6, 2015. Article (CrossRef Link).

[19] J.-S. Oh, H.-J. Choi, “A reflective practice of automated and manual code reviews for a studio project,” in Proc. of Fourth Annual ACIS International Conference on Computer and Information Science (ICIS’05), pp. 37-42, July 14-16, 2005. Article (CrossRef Link).

[20] I. Gomes, P. Morgado, T. Gomes, R. Moreira, “An overview on the static code analysis approach in software development,” Faculdade de Engenharia da Universidade do Porto, Portugal, 2009. Article (CrossRef Link).

[21] A. Costin, “Lua-code: security overview and practical approaches to static analysis,” in Proc. of IEEE Security and Privacy Workshops (SPW), May 25, 2017. Article (CrossRef Link).

[22] Sunil Manandhar, Kevin Moran, Kaushal Kafle, Ruhaa Tang, Denys Poshivanyk, Adwait Nadkarni, “Towards a natural perspective of smart homes for practical security and safety analyses,” in Proc. of 41st IEEE Symposium on Security and Privacy, San Francisco, CA, USA, pp.1-18, May 18-20, 2020. Article (CrossRef Link).

[23] M. Kim, J. H. Park, N. Y. Lee, “A quality model for IoT service,” in Proc. of Advances in Computer Science and Ubiquitous Computing, J. H. Park, Y. Pan, G. Yi, V. Loia (Eds.), Springer Singapore, Singapore, pp. 497-504, 2016. Article (CrossRef Link).

[24] D. Evans, D. Larochelle, “Improving security using extensible lightweight static analysis,” IEEE Software, vol. 19, no. 1, pp.42-51, August 7, 2002. Article (CrossRef Link).

[25] S. Wagner, J. Jürjens, C. Koller, P. Trischberger, “Comparing bug finding tools with reviews and tests,” in Proc. of Testing of Communicating Systems, F. Khendek, R. Dssouli (Eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 40-55, May 31-June 2, 2005. Article (CrossRef Link).

[26] B. Chess, G. McGraw, “Static analysis for security,” IEEE Security & Privacy, vol. 2, no. 6, pp.76-79, November, 2004. Article (CrossRef Link).

[27] D. Insa, J. Silva, “Automatic assessment of Java code,” Computer Languages, Systems & Structures, vol.53, pp.59-72, September, 2018. Article (CrossRef Link).

[28] JetBrains, IntelliJ IDEA. [Online]. Available: https://www.jetbrains.com/idea
[29] ISO/IEC 25010 – Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – systems and software quality models, Technical report, 2010.

[30] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harda, H. Watanabe, “A framework for measuring and evaluating program source code quality,” in Proc. of Product-Focused Software Process Improvement, J. Münch, P. Abrahamsson (Eds.), pp.284-299, July 2-4, 2007. Article (CrossRef Link).

[31] Byeong-Mo Chang, Janine Cassandra Son, Kwanghoon Choi, “An evaluation of the quality of IoT applications on the SmartThings platform using static analysis (extended version),” Preprint submitted to KSII Trans. on Internet and Information Systems(The Extended Version), pp. 1-30, 2019. Article (CrossRef Link).

[32] B. Chess, J. West, Secure programming with static analysis, 1st Edition, Addison-Wesley Professional, 2007.

[33] S. McConnell, Code complete: a practical handbook of software construction, Code Series, Microsoft Press, 1993.

[34] SmartThings community, Smartthings open-source DeviceTypeHandlers and SmartApps code, 2015. [Online]. Available: https://github.com/SmartThingsCommunity/SmartThingsPublic

[35] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, P. Tague, “Smartauth: user-centered authorization for the internet of things,” in Proc. of 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, pp.361-378, August 16-18, 2017. Article (CrossRef Link).

[36] Kashif Iqbal, Muhammad Adnan Khan, Sagheer Abbas, Zahid Hasan, Areej Fatima, “Intelligent transportation system (ITS) for smart-cities using mamdani fuzzy inference system,” International Journal of Advanced Computer Science and Applications (IJACSA), vol.9, No.2, pp.94-105, 2018. Article (CrossRef Link).

[37] Ayesha Atta, Sagheer Abbas, M. Adnan Khan, Gulzar Ahmed, Umber Farooq, “An adaptive approach: smart traffic congestion control system,” Journal of King Saud University-Computer and Information Sciences, 2018. Article (CrossRef Link).
Byeong-Mo Chang received his Ph.D degree from KAIST in 1988. He is currently a professor at Sookmyung Women’s University. His research areas are programming languages, program analysis and verification, and software security.

Janine Cassandra Son received his MS degree from Sookmyung Women’s University in 2018. Her research areas include programming languages, static analysis, and Internet of Things.

Kwanghoon Choi received his Ph.D degree from KAIST in 2003. He is currently an associate professor at Chonnam National University. His research areas include programming languages, type systems, compilers, and software engineering.