# An Evaluation of the Quality of IoT Applications on the SmartThings Platform Using Static Analysis

Byeong-Mo Chang[a], Janine Cassandra Son[a], Kwanghoon Choi[b,*]

[a] *Sookmyung Women's University, Seoul, Republic of Korea*
[b] *Chonnam National University, Gwangju, Republic of Korea*

**Abstract**

SmartThings is one of the most widely used open developer platform for smart home IoT solutions that allows users to create their own applications called SmartApp for personal use or for public distribution. The nature of openness demands high standards on the quality of SmartApps, but there have been few researches yet to evaluate it thoroughly. As part of software quality practice, code review is responsible for detecting violations of coding standards and ensures that best practices are followed. The aim of this research is to evaluate the quality of SmartApps through code review using an automatic static analysis. We first organize our static analysis rules by following the well-known Goal/Question/Metric paradigm, and then we apply the rules to real-world SmartApps in order to analysis and evaluate them. A study of 105 official published and 74 community-created real-world SmartApps found a high ratio of violations in both type of SmartApps, and of all violations, security violations contribute the highest in the defect density of SmartApps. Our static analysis tool can inspect reliability and maintainability effectively as well as security violations. Using the results of the automatic code review, we identify the common violations among SmartApps.

*Keywords:* Quality, Evaluation, Static Analysis, SmartThings, IoT Applications

## 1. Introduction

Interest in the Internet of Things (IoT) application development is continuously growing with the development of smart homes, devices, and other automation [1]. SmartThings is currently one of the most widely-used open developer platforms for Internet of Things with a high number of applications [2]. It also supports a wide range of smart devices which makes it attractive for users. The smart devices work through a central hub and an IoT application called SmartApp, which is a small Groovy-based program that allows users to automate their homes using the capabilities of the devices [3]. Since SmartThings focuses on home automation, it is critical that the program code used to run the user's home automation systems must be reliable, maintainable, and secure.

In SmartThings, users can install SmartApps directly from the marketplace where the official published apps can be found. On the other hand, users can also create their own app and can publish for themselves through custom code installation. SmartThings has a developer community that allows sharing of apps among users. They are called community-created SmartApps and anyone can use the publicly available apps for personal use. However, due to its nature as part of an open development platform, they are not guaranteed to pass SmartThings' standards since they are not officially endorsed. Modern IoT apps included, incorporate various capabilities and rich features but they could also open doors for various security concerns [4].

*Corresponding author
*Email addresses:* chang@sookmyung.ac.kr (Byeong-Mo Chang), janineson.it@gmail.com (Janine Cassandra Son), kwanghoon.choi@jnu.ac.kr (Kwanghoon Choi )

[1]This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2017R1A2B4005138).

More pervasive are IoT applications in our daily lives, more efforts must be put into keeping the qualities of the IoT applications. Every IoT application should be obliged to get through a form of review process on the software qualities before its deployment. The review process should evaluate software quality characteristics defined by a widely accepted standard for evaluating software quality 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.

To make sustainable a process of evaluating the qualities of IoT applications, it should be automated. Traditional code review process is usually done manually by a reviewer or a team who are knowledgeable about the standards set by the company [5]. Studies conducted have shown that automating the process through static analysis can greatly reduce code review effort [6, 7]. Although it cannot fully automate the process because not all rules can be applied through static analysis, it facilitates a more efficient way of reviewing source codes for potential vulnerabilities and non-compliance than manual code review.

In this paper, we aim to evaluate SmartApps on reliability, maintainability, and security of the official and community-created SmartApps. We choose to follow the Goal/Question/Metric (GQM) paradigm [8, 9] to organize a methodology used in our research. First, our goal is to evaluate how reliable, maintainable, and secure SmartApps are. Second, we develop a set of quantifiable questions that attempt to define and quantify the specific goal through two levels. In the high-level, three questions are made up more related to the general notion of the three software qualities. In the low-level, we refine them into 12 questions, which are more specific to SmartApps. This makes it easy to generate the relevant questions from the goals. Third, we set up 59 metrics collected from the code guidelines and best practices in SmartThings developer documentation and from the coding standards and best practices for the Groovy programming language, in order to answer the developed questions.

We also aim to automate the code review of SmartApps using static analysis and use the analysis output to evaluate the quality of SmartApps in terms of code defect density. Static analysis of source code checks them for compliance with the code review guidelines without executing the programs. The 59 metrics collected in our GQM-based methodology are implemented as rules in CodeNarc [10], a rule-based static analysis tool for Groovy [11], in the way that every rule implemented in CodeNarc one-to-one corresponds to a metric. The evaluation is based on violations of rules that are implemented within static analysis. SmartApp code quality can be evaluated by classifying the violations of rules into metrics, which are taken from the output of the analysis tool. We investigate the results after evaluating real-world SmartApps which source codes are available in a GitHub public repository.

The number of rule-violations for each SmartApp is measured and analyzed in terms of code defect density, which is used to evaluate the quality of SmartApp. A defect is defined as deviation from the guidelines and best practices, which are implemented as analysis rules. Violations of these rules do not necessarily cripple the system. However, they serve as warnings to prevent unwanted errors and most importantly to produce a maintainable code [12]. This automatic code review system can be divided into two parts: static analysis and evaluation tool. The static analysis tool generates the analysis report, which contains information about the code review violations. This report will be the input to the evaluation tool.

We investigated the quality of SmartApps through the analysis of 105 official and 74 community-created SmartApps from GitHub projects using this tool, which are all SmartApps available at the time of our investigation. We found high ratio of violations in both of the analyzed official and community-created SmartApps. This study indicates that security defects contributed the highest for both types of SmartApps. In the security violations, declaring unspecific subscriptions turned out to be the most common, and web services-related violations, which can pose a security threat if not reviewed properly, are the next. The maintainability defects of community-created SmartApps are noticeably higher compared to official apps, therefore they need more improvement in terms of readability or complexity of the source code.

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

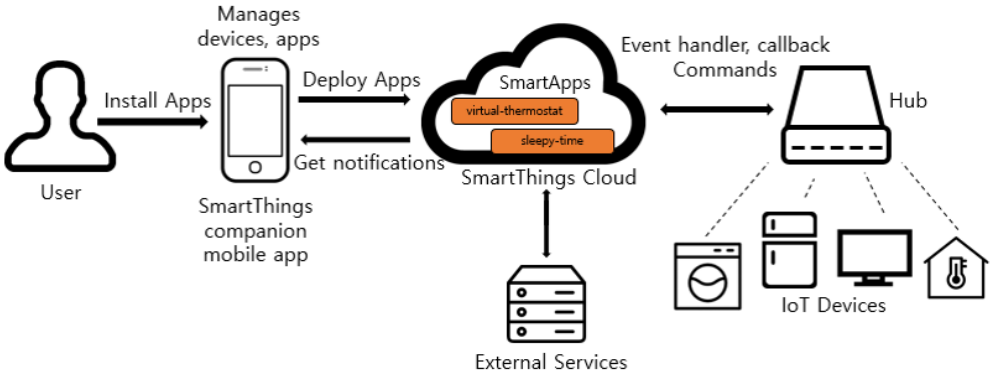


Figure 1: SmartThings architecture

## 2. Background

### *2.1. SmartThings and SmartApps*

As shown in Figure 1, the SmartThings architecture consists of four major components: the SmartThings cloud, hubs, devices, and the companion mobile app. The SmartThings cloud provides sandbox execution environment for SmartApps, which are Groovy-based SmartThings applications. Each hub is connected to the SmartThings cloud via WiFi or Ethernet, and it is also connected to devices at home via low-powered wireless protocols such as Z-Wave or ZigBee. Sensor information is sent from the devices to the hub so that SmartApp can access it, and the hub also has a role to control the devices by commands from SmartApps in the cloud. The SmartThings companion mobile app helps users to install SmartApps on the cloud and to bind one's own devices to the device names in the SmartApps.

As a background to SmartApps, this section provides information regarding their characteristics and functions. The source code of IoT applications is unlike standard application source code in certain ways [13]. SmartApps, in particular, have a different program structure compared to regular mobile applications.

```
//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()
```

```
28      initialize()
29    }
30
31    def initialize()
32    {
33     subscribe contact1, "contact.open", openHandler
34     subscribe contact1, "contact.closed", closedHandler
35    }
36
37    //event handlers
38    def openHandler(evt){
39     light1.on()
40     lock1.unlock()
41    }
42
43    def closedHandler(evt){
44     light1.off()
45    }
```

Listing 1: A typical SmartApp structure

*SmartApp structure.* The SmartApp in Listing 1 turns a light on when a door opens and off when it closes. Listing 1 shows a typical SmartApp structure [3]. It is composed of 4 main sections: *definition*, *preferences*, *predefined callbacks*, and *event handlers*. Every SmartApp must have a definition method, which provides metadata about the SmartApp itself. The preferences method defines a screen or page, where users can choose devices that the SmartApp will use at install time. Every SmartApp must define predefined callback methods named 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 they have for the referred device names `contact1`, `light1`, and `lock1`.

*Subscription model.* Like most home automation platforms, SmartApps also interact with the devices through events. SmartThings allows apps to register callbacks for a given event stream generated by a device [2]. It operates using a subscription model which allows devices to subscribe to an event and takes action when the event happens [3]. Subscriptions are declared in the predefined callbacks section and they invoke the 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 in a restricted form of Groovy, an optionally-typed and dynamic language for the Java platform [11]. Hence, the creation of new classes and calling certain methods are not allowed, among other restrictions, for performance and security purposes [3]. The restricted SmartThings environment conveniently provides predefined functions necessary for developing a smart application.

*External system access (Web service and other APIs).* SmartApps may also have access to external web services. APIs are available to make these requests possible. Web service SmartApps allow exposure to Web service endpoints and requests from external applications using an authentication service [2]. It exposes endpoints that allow third parties to make REST calls, and do the usual SmartApp functions like getting the device status and sending a response back to the calling client [3].

SmartThings has provided a comprehensive documentation and it includes *Code Review Guidelines and Best Practices* [3]. It consists of rules on how to develop SmartApps correctly for personal use or for public distribution. It is also one of the criteria used by SmartThings to evaluate user-submitted SmartApps for publishing. However, at the time of writing, their official website states that they are not reviewing submissions for public distribution [3].

One use case for smart applications is to schedule a job to run on a specific schedule. Listing 2 shows an example of a violation called *Avoid Chained RunIn Call*. Avoid chained *runIn()* calls involves the use of

*runIn()* method which executes a specified handler after a given number of seconds have elapsed. According to the guideline, chaining *runIn* calls should be avoided since it is prone to failure. When a scheduled execution in *handler()* fails, it will not be able to reschedule itself, thus, breaking the whole chain. Instead, a predefined scheduling function such as *runEvery5Minutes()*, should be used to specify a recurring schedule [3].

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

Listing 2: An example of rule violation

### *2.2. Static Analysis using CodeNarc*

Static analysis is closely related to code review or inspection. It examines source code without actual execution of the program as opposed to dynamic analysis (testing software by executing programs) [14]. The term is mostly associated with the analysis performed by an automated software tool [15]. The results of static analysis tools are not always real defects but can be warnings to mark that a piece of code is critical in some way [16]. Examples of issues that can be detected by these tools are empty catch blocks, unused variables, and uncaught exceptions [17].

In software development, programmers may encounter errors or make mistakes. A typical scenario would involve the compiler noting the error, the programmer fixing it, and then the development is continued. However, some errors especially those related to security may not be easily discovered early in the development process. A defect may be discovered later after production where it will be more expensive to fix [15]. In the case of IoT applications and smart homes, this can be critical especially when security is compromised. An advantage of static analysis approach is it can detect anomalies early in the development phase. In contrast, testing or dynamic analysis is being done late in the development process since it needs to be applied to code execution. During development, static analysis approach makes the code more reliable, readable, and less prone to errors in future tests [15].

Rule-based static analysis tools work by detecting warnings or violations defined based on a particular rule or pattern. Rules can be added or deleted easily according to the purpose of analysis. It, however, cannot detect an error that has not matched with a fixed set of patterns or rules [18]. This could be a disadvantage because if critical violations were missed, then the tool will have no way to catch them.

Static analysis can automatically detect software problems before it is released for public use. This study utilizes CodeNarc [10], a static analysis tool for analyzing Groovy code, that checks violations based on over 300 rules. The tool comes with a set of predefined configurations on a set of applicable patterns or rules, but it can also be customized for a particular project. For example, the *Basic* category of CodeNarc contains rules that report violations or warnings such as *Empty else block* and *Dead code*, which is code that is unreachable. CodeNarc provides mechanisms to detect code inconsistencies in order to produce a good design that ensures cohesive and maintainable code [19].

In this work, we consider CodeNarc since it is open source and configurable. It is highly extensible, similar to the more widely used Java static analysis and bug finding tools such as Checkstyle and PMD [7]. It also has available plugins supported by major IDEs, such as IntelliJ IDEA [20], which is the IDE used in this research for evaluating the SmartApps. In order to run CodeNarc in the context of SmartApps, new rules must be added. As a result, it becomes suitable for checking rule violations based on the documentation by SmartThings.

Table 1: Software Qualities in ISO/IEC25010 and Their Inclusions for Evaluation of SmartApps

| 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 system, product or component can be transferred from one hardware, software or other operational or usage environment to another | Excluded |

## 3. Methodology

### *3.1. A GQM-based Approach*

The main purpose of this paper is to evaluate the quality of smart home IoT applications on the SmartThings platform using static analysis. To accomplish this, we adopt the Goal/Question/Metric (GQM) paradigm [8, 9] in our research. The GQM paradigm is a well-known mechanism for defining and interpreting software measurement. Under the GQM paradigm, a certain goal is defined, this goal is refined into questions, and, subsequently, metrics are defined that should provide the information to answer these questions. By answering the questions, the measured data can be analyzed to identify if the goals are attained.

#### *3.1.1. Goal*

Our goal is set to evaluate three software qualities on SmartApps, reliability, maintainability, and security, particularly emphasizing to address the following research questions in the evaluation of the three software qualities of SmartApps:

- **RQ1:** What are the common violations found in SmartApps?
- **RQ2:** How do community-created SmartApps differ from official SmartApps in terms of quality?

The three software qualities are chosen from *ISO Systems and software Quality Requirements and Evaluation (SQuaRE)* - system and software quality models. The ISO software quality characteristics are widely used as a standard for evaluating software quality. They are established to formally identify quality characteristics related to software system requirements, objectives and quality assurance [21]. They are categorized as 8 qualities as shown in Table 1.

However, since it is for general software, all of the qualities do not fit the purpose of evaluating IoT source code quality particularly in terms of evaluating SmartApps. Some quality characteristics cannot be easily measured by analyzing the source code only [22]. We choose 3 quality characteristics matched with the key characteristics for SmartApps: *reliability*, *security*, and *maintainability*. *Reliability* is included for

evaluation of 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 home. *Maintainability* is a good criterion to see if SmartApps are developed easy to customize. Each home will have a different set of connected devices, and so it must be easy to modify SmartApps to adapt it for the user's home.

We exclude the rest of the 8 software quality characteristics from our goal by the reasons explained in the following. *Functional suitability* is excluded because the specification of each SmartApp under evaluation is unavailable and so it is hard to quantify this quality characteristic. *Performance efficiency* has never gotten user's attention much in the application domain of SmartApps because users typically use SmartApps connected with less than a dozen of devices at home. So, we decided not to use it for our evaluation. As to *compatibility* and *portability*, there is no sense to evaluate on these characteristics because SmartApps run on a single platform, SmartThings. *Usability* is an important characteristic but we excluded it from our list by the reason that it has not been thoroughly discussed yet what is good or bad for usability in the area of IoT applications, which is contrary to usability in mobile applications. Moreover, it is not easy to have an automatic method to evaluate in terms of this characteristic.

### *3.1.2. Question*

Table 2: Goals and Questions in the GQM Paradigm

| Goal | Question | No. Sub-Questions |
|---|---|---|
| Reliability | Evaluate the frequency of faults | R1. Are null values handled properly?<br>R2. Are there any potential mistakes/typos?<br>R3. Are there busy loops?<br>R4. Are there missing cases?<br>R5. Are there any inconsistencies?<br>R6. Are there any unused values/codes? |
| Maintainability | Evaluate the easiness of identifying styles, structure, and parts for maintenance | M1. Is the logic too complex?<br>M2. Is the code hard to read? |
| Security | Evaluate the possibility of vulnerabilities and attacks | S1. Are there hard-coded values?<br>S2. Are there external HTTP requests?<br>S3. Not easy to predict the behavior?<br>S4. Are there any uses of unrestricted things? |

The goal to evaluate reliability, maintainability, and security of SmartApps is now refined into two-level questions as shown in Table 2. Three questions in the high level are associated with the three software qualities, respectively. Each description of the qualities by 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 degree to which the SmartApps perform their functions. Second, the easiness of identifying styles, structure, and parts directly connect 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.

The three high-level questions are a bit general, and so we need to make them specific to the context of SmartApps on the SmartThings platform. We have therefore derived more 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 on 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 specific questions. Note that they get 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 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 handling. R4 checks if there are any potential faults due to missing cases such as missing event handlers. SmartApss are prone to faults if a certain event handler is called but was not defined at all. 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 easiness 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 is the number of linearly independent paths through the source of SmartApps. M2 is a subquestion 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 up four sub-questions in Table 2. S1 examines issues related to security since hard-coded phone numbers can pose a risk to the system if the value is wrong and cannot be updated using the SmartApp preferences and settings. This suggests a safe and proper implementation by using the contact input so it will be subjected to validation and can be updated. S2 is a subquestion on potential vulnerabilities caused by using the HTTP protocol. SmartApps can make HTTP request to the outside services and they can run as a HTTP endpoint to serve requests from the outside. S3 tries to examine if 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 name to subscribe to an event. Also it should be avoided to use dynamic method execution as much as possible. S4 is a subquestion on the uses of unrestricted things. Although SmartApps are written in Groovy, there are some restrictions in the use of full features of the programming language.

#### *3.1.3. 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 collect some review guidelines that can be used to answer one of the 12 (sub-)questions explained previously. A guideline can be found at the SmartThings official website developer documentation [3]. Some examples of the guidelines include *Avoid recurring short schedules* and *Do not use dynamic method execution*. We choose them as our metrics because they can be used to answer the questions R3 and S3, respectively. Another guidelines such as *Code should be readable* and *Comment appropriately*, are also part of the code review. However, the guidelines are too general and requires more information as to what a 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 [10]. Since SmartApps are written in 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 is done to incorporate 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 are collected to answer all the 12 (sub-) questions for our GQM-based methodology. 20 out of 30 guidelines from the SmartThings code review guidelines are chosen as new metrics. Meanwhile, 38 out of 357 default CodeNarc rules are selected, as they are applicable to SmartApps. An additional metric is also included to count the 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 to indicate where they come from. The sources spread from SmartThings best practices to various guidelines such as basic or general

Table 3: Metrics and Their Static Analysis Rules (excerpt from Appendix A)

| Goal | Question | 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 | R5 | *Verify array index* | 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 execution* | 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 |

rules, convention, security considerations, size, and documentation. Refer to Table A.9 in the appendix for a full list. Also refer to Table C.12 in the appendix for detailed descriptions for all the metrics.

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 subquestions 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 to organize the measurement from top-down perspective, there remains a problem on how to interpret the result of the measurement in terms of the software qualities of SmartApps. It is not so easy to justify how to set an absolute software quality index on determining whether or not the software quality of a SmartApp is good. Besides, it is not possible to do a relative comparison between a new measurement result with old one. This is because SmartThings is a new home IoT platform and it has no such a thing as the previous accumulated result of the measurement yet.

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

The purpose of the two research questions in our goal are 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.

### *3.2. A Static Analysis for the GQM-based Measurement*

In order to measure the metrics in our GQM-based methodology automatically, a code review tool is developed to detect violations and evaluate the qualities of SmartApps. We utilize CodeNarc, an open source static analysis tool for Groovy since SmartApps are written in 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.

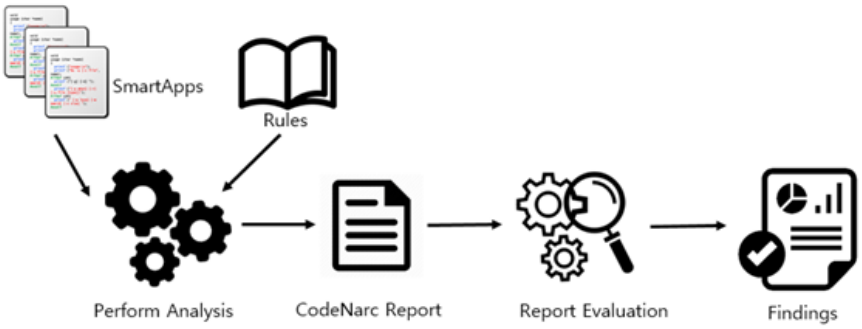


Figure 2: Running the tool and evaluating the results

We have set up 59 static analysis rules in the way that every rule one-to-one corresponds to one of all 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 measurement. 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 guideline (code, design, specifications) that is within the capabilities of static analysis must be included. This is to ensure a valid and meaningful output that will help improve the quality of the program. However, not all of the guidelines can be implemented in the static analysis tool. An example of a rule that can only partially be implemented is *Document external HTTP requests*. Static analysis can detect HTTP external calls but it cannot fully confirm the threat. Hence, manual inspection from human code reviewers is recommended since static analysis is not capable of determining the reason why the external request must be made. Based on the guideline only, we do not know which endpoint is safe or not. This limitation can be overcome if we are able to get more information about safe or unsafe endpoints [23]. We leave the evaluation and implementations of such enhancements in the future work.

Figure 2 displays the flow of the automatic code review and evaluation process. To get started, we gather the SmartApps and configure the tool according to the code review rules. Then, analysis is performed through CodeNarc. This phase will generate a set of raw results from the CodeNarc report. Finally, we analyze the summary of the defects found in the CodeNarc report. Each official SmartApp defect density score is displayed and the most frequently occurring violation among the SmartApps is also identified. The same was done for community-created SmartApps. Then, the two reports are compared to investigate their similarities and differences in the quality characteristics.

## 4. Implementation

In this section, we explain how to implement our static analysis tool for measurement and evaluation. Our tool is available as an 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

### *4.1. Writing New Static Analysis Rules*

In this research, we make use of both CodeNarc default rules and new custom rules. A total of 59 rules were used to evaluate SmartApp quality attributes such as reliability, security, and maintainability 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 a list of static analysis rules for SmartApps (See Appendix A for all the rules). For some static analysis rules that require thresholds such as *Cyclomatic Complexity*, we used the default values used in CodeNarc. The detailed description of the metrics in Appendix C.12 specifies the default values.

CodeNarc has provided most of the libraries to make it easier to create new rules. After defining a new rule name using the CodeNarc command line program, it will automatically create the related Groovy rule files. We write in the generated Groovy rule file the condition for checking if a certain piece of code is a violation. To test if the rule is working, CodeNarc also automatically creates a unit test file for that particular rule where success and violation test cases can be written.

Since CodeNarc uses static analysis, it relies heavily on Abstract Syntax Tree (AST) traversal to inspect the code structure and to check violations without running the program [2]. Listing 3 illustrates an example of code that checks if the subscription is clear. An unclear subscription involves the use of string variable instead of explicitly stating the attribute (e.g., *'contact.open'*). The code works using the AST visitor method *visitMethodCallExpression* where it visits all method calls with the name *'subscribe'*. The second argument of the subscribe method call contains the attribute that the user wants to subscribe. If it is detected to be a variable instance, then it is a violation.

```
1  class ClearSubscriptionAstVisitor extends AbstractAstVisitor {
2   @Override
3   void visitMethodCallExpression(MethodCallExpression call){
4     if(AstUtil.isMethodNamed(call, 'subscribe', 3)) {
5       def attributeName = call.arguments.expressions[1]
6       if (!(attributeName instanceof ConstantExpression))
7       addViolation(call, 'Subscriptions should be clear.')
8     }
9     super.visitMethodCallExpression(call)
10  }
11 }
```

Listing 3: CodeNarc custom rule *ClearSubscription* for the metric 'Subscriptions should be clear' (excerpt)

### *4.2. Configuring and Running the Tool*

After writing the additional rules, the tool must be configured to refer to the new rule set. Rules based on SmartThings guidelines are encoded as custom rules. Customizing it for the SmartThings environment can improve the quality of the static analysis tool's result since it can now detect violations in the context of the SmartApp code [24].

CodeNarc can run multiple files at once. It performs well in analyzing multiple SmartApps since they are composed of small programs. Each file has an average of 200 lines, therefore, it does not take a lot of time to analyze multiple SmartApps.

As with all code review tools and even in manual code inspection, CodeNarc also generates a report that shows the defects in detail for the purpose of correction and improvement of the source code. It can generate an HTML or XML report containing information regarding the violations such as line number and description of the issues.

Figure 3 shows the output of the plugin when run using the IDE. Particularly it shows a screen shot of the CodeNarc for IntelliJ IDEA plugin using SmartThings custom rules. We customized this tool to count the LOC for every file for the purpose of calculating the defect density metrics. After scanning all the files in the source folder, it returns the number of items detected including the violations and total number of files scanned. Then, it shows a breakdown of the items per app. For example, the figure displays four items detected under *beacon-control.groovy*, the first is the LOC and the rest are violations. The file has two kinds of violations and one of them, *Use consistent return values*, was detected in two different lines. The next file, *big-turn-off.groovy* only has one item, which is the LOC. There are no violations detected for that file.

The number of defects or violation count is divided over the lines of code (LOC), then multiplied by 1000, since we want to display an output based on KLOC. According to McConnell, the industry average is about 15-50 errors per 1000 lines of delivered code, also known as defects per KLOC [25]. However, for this

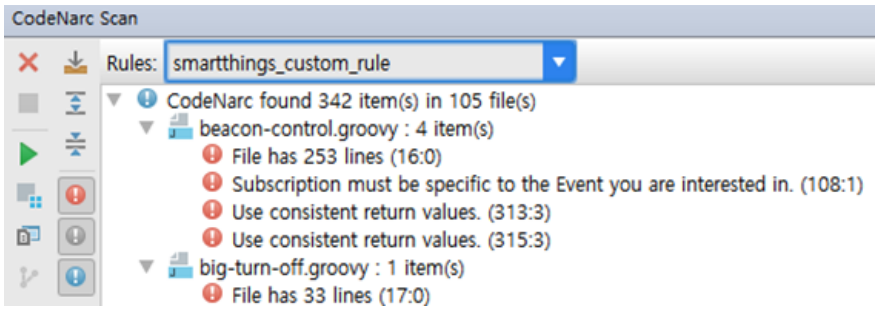


Figure 3: A screenshot of the CodeNarc for IntelliJ IDEA plugin using SmartThings custom rules.

research, we will follow the official SmartApp defect density mean standard described, which is computed as:

$$code\ defect\ density = \frac{\#\ of\ defects}{lines\ of\ code} \times 1000 \tag{1}$$

where the number of defects is the number of violations discovered by the tool. We calculate the total code defect density and the densities for each of reliability, maintainability, and security.

Figure 4 displays an excerpt of the output of the evaluation tool where it shows details of the 3 violations including the source and line numbers where the violations were detected. The computed defect density and breakdown of violations and other metrics are also displayed. In the report, *routine-director.groovy* has a reliability defect density of 7.66. This value was computed based on the occurrence of a rule violation under reliability, *Avoid recurring short schedules*. This violation involves a scheduling function that executes more frequently than every five minutes. Unless there is a good reason for executing short periodic functions, this kind of design must be avoided according to the guidelines. In the example, the violation appeared twice as seen in the breakdown of violations, making the defect density for reliability higher. Using this report, we can evaluate the quality of SmartApps based on their defect density score. We can now determine which apps have high or low quality and also find out what types of violations are common in SmartApps.

## 5. Analysis of the Results

In this section, we report results of the study by answering the research questions in Section 3. The raw data from our analysis and evaluation can be freely downloaded from

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

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

- Static analysis report: CodeNarcAntReport-official.html, CodeNarcAntReport-cc.html
- Evaluation result: codereviewout-Official.txt codereviewout-cc.txt

In order to analyze SmartApps through static analysis, we need to have access to the source code of the apps. We gather all SmartApps (referred to as population) from a public GitHub repository [26] which consists of 105 official and 74 community-created apps. In total, 179 apps are analyzed. They are all SmartApps available in the SmartThings developer site at the time of our experiment. A snapshot of all these SmartApps used for our analysis and evaluation is copied into a repository for reproduction.

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

In this study, we differentiate the SmartApps into two types: official and community-created. Official SmartApps are already used in production and they are available in the SmartThings marketplace, an app store. Community-created SmartApps are also publicly distributed apps shared among users. However,

```
FILENAME : routine-director.groovy
rule name : SpecificSubscription line : 103
source line/ message : [SRC]subscribe(people, "presence", presence)
[MSG]Subscription must be specific to the Event you are interested in.
rule name : AvoidRecurringShortSchedules line : 118
source line/ message : [SRC]runIn(60,"setSunrise")
[MSG]Avoid recurring short schedules unless there is a good reason for it.
rule name : AvoidRecurringShortSchedules line : 122
source line/ message : [SRC]runIn(60,"setSunset")
[MSG]Avoid recurring short schedules unless there is a good reason for it.
---Defect Density Metrics (KLOC)---
Reliability - 7.66
Security - 3.83
Maintainability - 0.0
Total Defect Density - 11.49
---Breakdown of Violations and Other Metrics---
Lines of Code : 261
No. of Input : 11
No. of Subscriptions : 3
AvoidRecurringShortSchedules : 2
SpecificSubscription : 1
Total Violations : 3
```

Figure 4: SmartApp code review evaluation report (excerpt)

they are not guaranteed to pass any code review or inspection since they are not part of the official apps approved by SmartThings. Therefore, using these apps could lead to risks.

In the following, we will explain the analysis report and evaluation result over all the official and community-created SmartApps in detail.

### *5.1. 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. It also shows that the number of violated rules for both sets are 25.

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 on the violations found among 105 official and 74 community-created SmartApps. The percentage of SmartApps that share common rule violations was calculated for each set. *Subscriptions should be specific* is revealed to be the most common violation for all SmartApps where it was found in 42% of community-created apps and in 23% of official apps.

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 of the three quality attributes used in the study, with security having the highest impact. In majority of the rules, the percentage of official apps which violated them are less compared to community-created apps as shown in Figure 5. This can be attributed to reviews and tests done by the SmartThings team prior to publishing the apps in the Marketplace. On the other hand, *Use consistent return values* appeared more frequently in official apps. It could also be observed that *Constant if expression* and *Empty catch block* violations were found only in official apps.

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

| Rule | Quality Attr. | % 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 |
| *Abc 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 |

The top 5 common violations found are concerned on the security and maintainability aspects of the code. It is worth noting that even though official published apps have passed the SmartThings code review, the automatic code review tool still detected violations which the reviewers may have missed or may have disregarded.

The following describes the top 5 common violations in detail:

*Subscriptions should be specific.* A SmartApp that is subscribed to every event will execute excessively and is rarely necessary. Broad subscriptions may even lead to security issues, in that they could allow unintended or unnecessary operations to be executed. The best practice is to create subscriptions specific to the *event* you are interested in. An example is an app that is subscribed to a presence sensor but only used one of its events. A violation occurs when the overall event *presence* is used as attribute in the subscription instead of a specific event *presence.present*.

*Document exposed endpoints.* Exposed endpoints should be documented to disclose the purpose of the API and what data those APIs may access. The automatic code review tool flags this because of security risks. This involves remote access where the SmartApp acts as a web service to take commands from the remote server. The ability of a SmartApp to act as a web service may expand the attack surface and could allow malicious server to send dangerous commands to a SmartApp running on the user's devices [27].

*Document external HTTP requests.* HTTP requests to external services should include documentation on its purpose and usage. This is also flagged due to possible security issues that may arise if ignored. This remote access type involves sending of data to a remote server. This can be privacy-invasive especially if the user who installed the app does not know where it sends data and how it will be used [27].

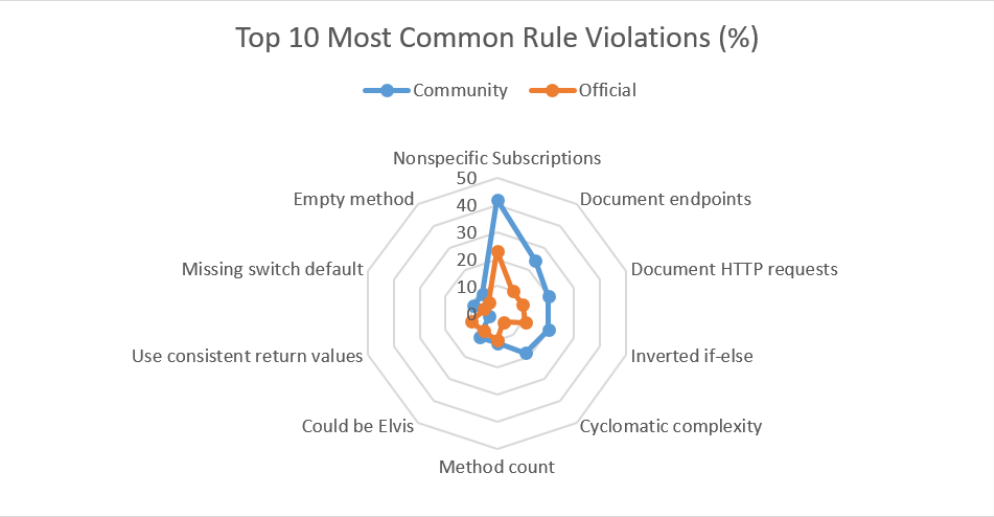


Figure 5: Top 10 most common rule violations

*Inverted if-else.* This is a CodeNarc default convention rule focused on the readability of the source code. Violations of this category do not necessarily mean that the code will not work. Instead, it suggests improvement in the maintainability aspect of the code by following coding conventions to make the code easier to understand [17].

*Cyclomatic complexity.* This is a size-related metric included in CodeNarc that requires a certain threshold to tell whether a source code is too complex or not. Code with high cyclomatic complexity could be difficult to understand and could indicate a higher probability of having defects.

### *5.2. RQ2: How do community-created SmartApps differ from official SmartApps in terms of quality?*

Table 6 reports on the results of the analysis where it reflects the findings in RQ1, which revealed that security-related issues are the top contributors to defects in SmartApps. In order to compare the difference of the average code defect densities of both sets, we use a statistical test called Welch's t-test. This is introduced to test the hypothesis that two populations have equal means, and this is known to be more reliable when the two samples have unequal variances and/or unequal sample sizes. Figure 6 shows a two-sample Welch's t-test using T distribution based on the mean and standard deviation 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 there is a significant difference found in both set of apps with regards to the reliability, maintainability, and security based on the automatic code review evaluation.

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 |

Tables 7 and 8 show the code defect densities of both official and community-created apps, along with the lines of code and the number of input and subscriptions. SmartApps typically consist of devices and other constants declared using input. Cases where there are no explicit declaration of input and subscriptions

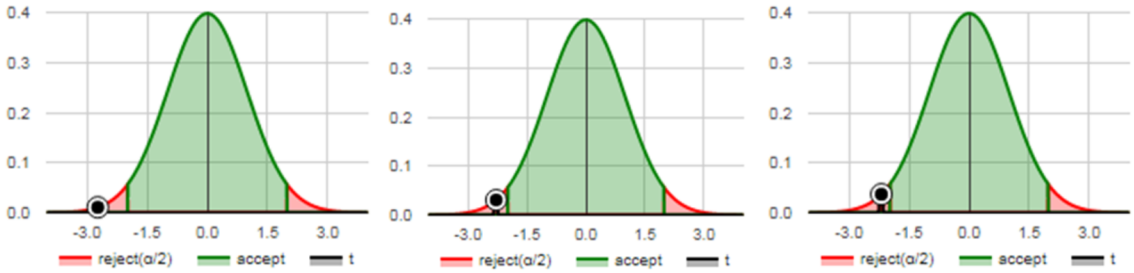


Figure 6: Two-sample Welch's t-test using T distribution (from left to right: reliability, maintainability, security)

to event handlers (refer to Table 7, *wattvision-manager.groovy* example) are also possible. These are *Web Services SmartApps* and they operate by exposing an endpoint that allows external applications to make API calls to a SmartApp to get information and control the devices. Both tables show SmartApps with violations and we observed that in most cases, it shows a relationship where the number of input devices is higher than subscriptions because some of the devices are subscribed while others are used only in action commands to control the devices. The automatic code review tool has revealed that some cases are due to the violation of *Subscription should be specific* rule as stated in the results of RQ1. However, it is also possible that the number of input devices is lower than subscriptions because an input device can generate multiple events.

Official SmartApps are projects that can be considered as mature since they are deployed and already being used in operation. They have been found to have fewer defects in maintainability and reliability. We can see that the reliability and maintainability defect density scores of the top 20 official apps in Table 7 are significantly lower compared to those of the top 20 community-created apps in Table 8 (See Appendix B for defect density scores of all the apps). This implies that official apps most likely follow Groovy conventions related to the readability of code. Meanwhile, we found out that most of the community-created apps do not follow conventions, therefore, the code can be hard to read and maintain.

There are noticeable gaps between security, reliability, and maintainability in community-created SmartApps. Even though the ranks of the qualities are the same for both apps, the defect density score in community-created apps is greater in all aspects. This implies that the quality of community-created apps needs more improvement, especially in terms of security and reliability of source code.

There are several factors contributing to the high defect density. It is computed using the ratio of the defects against lines of code. Using the formula, we expect that SmartApps with less LOC typically have fewer defects than longer codes. Based on the set of SmartApps analyzed, the typical size of a SmartApp code is less than 200 lines. The ideal value for the defect density metrics must be zero, which means there are no issues found. We observed that some official SmartApps only contain violations related to security, which greatly influenced its ranking as the highest among the three qualities that need improvement.

One of the apps with a high total defect density (see Table 7), in fact, only contained one kind of violation under security but because of its low LOC count, the defect density score turned out to be very high. Another factor can be attributed to the frequency of the same type of error in an app. The same type of error can appear multiple times (see Figure 4) in the app and it could greatly affect the defect density.

Defects are defined as non-compliance of the guidelines and best practices in this paper. They can be warnings to potential vulnerabilities that may arise if guidelines for SmartThings and the universal standard in programming are not followed. Based on the reviews and tests conducted by SmartThings, it implies that official published apps are functioning normally as expected. However, according to our results in RQ1 which showed that 23% of official apps incurred security defects related to unspecific subscription, unintended functions may arise and could cause problems that may not manifest in the current state of the system. As systems become more complicated and critical as more IoT devices are interconnected with each other, the probability of failure or malfunction will continue to increase if defects occur in the code.

The results also indicate that developers in the SmartApp community could improve their code if they follow the guidelines provided by SmartThings and it would help if they follow coding conventions when

Table 7: Official SmartApp code defect density metrics (excerpt from Appendix B)

| SmartApp | LOC | Input | Subscr. | Reliability | Maintainability | Security | Total |
|---|---|---|---|---|---|---|---|
| presence-change-push.groovy | 28 | 1 | 1 | 0 | 0 | 71.43 | 71.43 |
| let-there-be-light.groovy | 32 | 2 | 1 | 0 | 0 | 62.50 | 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 | 0 | 0 | 25.64 | 9.62 | 9.62 | 44.87 |
| energy-alerts.groovy | 73 | 5 | 1 | 27.4 | 13.7 | 0 | 41.1 |
| turn-on-only-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 |
| yoics-connect.groovy | 411 | 2 | 0 | 12.17 | 9.73 | 14.6 | 36.5 |
| life360-connect.groovy | 419 | 7 | 0 | 11.93 | 4.77 | 16.71 | 33.41 |
| tesla-connect.groovy | 338 | 3 | 0 | 2.96 | 8.88 | 20.71 | 32.55 |
| logitech-harmony-connect.groovy | 843 | 14 | 1 | 1.19 | 21.35 | 9.49 | 32.03 |
| garage-door-monitor.groovy | 94 | 3 | 1 | 0 | 10.64 | 21.28 | 31.92 |
| lock-it-when-i-leave.groovy | 63 | 4 | 1 | 0 | 0 | 31.75 | 31.75 |
| close-the-valve.groovy | 64 | 6 | 1 | 0 | 0 | 31.25 | 31.25 |
| cameras-on-when-im-away.groovy | 69 | 2 | 1 | 0 | 0 | 28.99 | 28.99 |
| lifx-connect.groovy | 424 | 1 | 0 | 7.08 | 11.79 | 9.43 | 28.3 |
| ecobee-connect.groovy | 673 | 1 | 0 | 1.49 | 17.83 | 8.92 | 28.24 |
| ifttt.groovy | 214 | 11 | 1 | 4.67 | 4.67 | 18.69 | 28.03 |

Table 8: Community-created SmartApp code defect density metrics (excerpt from Appendix B)

| SmartApp | LOC | Input | Subscr. | Reliability | Maintainability | Security | 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 | 18.18 | 0 | 27.27 | 45.45 |
| 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 |

they submit their code for public use. Open development is indeed beneficial to the community and could encourage more developers but certain measures such as automatic code review and testing must be considered in order to publish high-quality apps. Also, this study suggests, as what had been stated in previous studies that static code analysis is not a replacement for human-led code reviews, but rather a supplement to them [24, 15]. In addition, the static analysis tool proved to be a good pre-stage to a review since not all defects should be manually checked judging from the high numbers of the defect density scores for both kinds of SmartApps in the evaluation. Problematic code that poses a possibility of threat can be marked so it will not be overlooked in the review [16].

## 6. Limitations

The automatic code review tool used in the study treats all external HTTP calls as violation since 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 WebService SmartApps in the population of the analyzed apps has a significant effect to the outcome of the code review evaluation. 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 the future work by incorporating other analysis techniques [28, 29] more dedicated to security.

Finally, the implementation of the custom rules is based on the code review guidelines by SmartThings and some of them may allow exceptions based on the context of the code. In the same way, it could lead to false positives though it may be rare.

## 7. Related Work

This section discusses related literature regarding the use of static analysis tools in software development using popular programming languages. In addition, we discuss the role of static analysis in code review.

### *7.1. Static Analysis Tools*

In order to aid programmers in software development, open source and commercial static analysis tools have been widely used. Among the existing popular Java static analysis tools are Checkstyle [30], PMD [31], and FindBugs[32]. Checkstyle focuses on readability problems, while PMD highlights suspicious situations in the source code [17]. FindBugs concentrates on catching code vulnerabilities, bad practices, and design flaws. Fernandes et al [2] reveals by static analysis that over 55% of SmartApps in the store are overprivileged due to the capabilities being too coarse-grained. Moreover, a SmartApp is granted full access to a device even if it specifies needing only limited access to the device. In our study, we used CodeNarc, which also has customizable properties similar to its more famous Java counterparts. CodeNarc is open source and specific for the Groovy programming language. Panichella et al [17] used the existing tools with their default configurations in their study. Meanwhile, our research used an existing static analysis tool but we explored its custom rule capabilities in order to fit the goal of the research, which is to analyze SmartApps.

### *7.2. Code Review*

Code review is the systematic analysis of program code that aims to correct mistakes in order to improve the overall quality of software [17]. It can be done through manual review involving human code auditors or through an automated tool. In both cases, code review requires a set of rules or the type of errors to look out for before the code examination can be performed [15]. The rules can be based on the requirements and design found in the documentation, or secure coding guidelines, if available.

For more popular programming languages such as Java and PHP, there are existing baseline corpora of vulnerable samples and applications. They serve as a guide in understanding the insecure coding practices and anti-patterns in those languages [23]. Since there are no formal secure coding guidelines specific to SmartThings, we combined the rules from the code review guidelines, documentation, and applicable rules from CodeNarc to implement a custom tool.

### *7.3. Using Static Analysis in Code Review*

Static analysis tools have been widely used in software development to detect bugs and defects in the program. Researchers have studied several topics on how static analysis tools facilitate in the code review process.

Panichella et al. [17] investigated how warnings detected by static analysis tools are removed in the context of code reviews conducted on Java open source projects. Results indicated that by analyzing specific categories of warnings, the percentage of removed warnings is high. They also found out that projects using static analysis tools fixed a higher percentage of warnings than other projects. It is relevant to our study since it suggests that static analysis tools can be used to support developers during code reviews. The study has proved that the removal of warnings in the development phase will produce apps with higher quality.

Some other studies focused on how static analysis tools can reduce code review effort. Singh et al. [7] concluded that static analysis tools such as PMD may prove to reduce the workload of code reviewers through a study where the warnings made by the tool matched those of the reviewer's comments. Gomes et al. [15] explored the use of automated tools for static analysis. They discussed how errors such as security vulnerabilities can be complicated and exist in hard-to-reach states. Static analysis tools are capable of looking into those errors with less fuss since it does 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.

### *7.4. Security Analysis*

There have been some researches to study security analysis on smart home applications based on SmartThings [2, 28, 29]. Fernandes et al [2] presented the in-depth empirical security analysis of the SmartThings platform to report two security flaws. First, in SmartThings's privilege separation model, the flaws lead to significant overprivilege in SmartApps. Second, the event system does not sufficiently protect events that carry sensitive information. Cilik et al [28] developed a static taint analysis tool called SAINT for IoT applications such as SmartThings. Using it, they reported to uncover many sensitive data flows on SmartThings apps. Celik et al [29] designed and implemented a system called SOTERIA for model checking IoT applications. Given SmartApps and a property of interest, the system can automatically extracts a state model and can apply model checking to find property violations.

## 8. Conclusion and Future Work

This paper investigated SmartApp quality through an automatic code review tool using static analysis. As far as we know, this is the first automatic code review tool specific for SmartThings apps. Throughout this study, we found out the high ratio of violations in both of the analyzed official and community-created SmartApps. At first, violations related to security such as declaring unspecific subscriptions turned out to be the most common. Next, web services-related flags are also common and they can pose a security threat if not reviewed properly. In addition, convention and size related violations that indicate poor maintainability is also an issue with the community-created SmartApps. This study also indicates that security defects contributed the highest for both types of SmartApps. However, the maintainability defects of community-created SmartApps are noticeably higher compared to official apps, therefore they need more improvement in terms of readability of the source code.

Automatic code review can ensure that the source code is reliable, maintainable and secure based on the rules defined in the static analysis tool. We were able to come up with meaningful results for basic SmartApps. Certain apps make use of external web services that involve sharing of data to third-party domains. Therefore, it needs manual review to ensure the security of the app. The automatic code review tool can only detect and flag warnings to external web services that may pose security threats without confirming. For the future work, we plan to perform an in-depth analysis on how to evaluate the qualities of SmartApps with external services.

## 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 4 (5) (2017) 1250–1258. doi:10.1109/JIOT.2017.2694844.

[2] E. Fernandes, J. Jung, A. Prakash, Security analysis of emerging smart home applications, in: 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 636–654.

[3] SmartThings, Smartthings developer documentation (2017).
URL http://docs.smartthings.com

[4] B. Aloraini, M. Nagappan, Evaluating state-of-the-art free and open source static analysis tools against buffer errors in Android apps, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 295–306.

[5] J.-S. Oh, H.-J. Choi, A reflective practice of automated and manual code reviews for a studio project, in: Fourth Annual ACIS International Conference on Computer and Information Science (ICIS'05), 2005, pp. 37–42.

[6] R. M. Hartog, Octopull: Integrating static analysis with code reviews, Master's thesis, Delft University of Technology, the Netherlands (02 2015).

[7] D. Singh, V. R. Sekar, K. T. Stolee, B. Johnson, Evaluating how static analysis tools can reduce code review effort, in: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2017, pp. 101–105. doi:10.1109/VLHCC.2017.8103456.

[8] V. R. Basili, Software modeling and measurement: The goal/question/metric paradigm, Tech. rep., College Park, MD, USA (1992).

[9] 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: Proceedings Seventh International Software Metrics Symposium, 2001, pp. 246–258. doi:10.1109/METRIC.2001.915533.

[10] CodeNarc, Codenarc homepage (2018).
URL http://codenarc.sourceforge.net

[11] Groovy, Groovy homepage (2003).
URL http://groovy-lang.org

[12] Oracle, Code conventions for the java programming language.
URL http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html#16712

[13] M. Kim, J. H. Park, N. Y. Lee, A quality model for IoT service, in: J. J. J. H. Park, Y. Pan, G. Yi, V. Loia (Eds.), Advances in Computer Science and Ubiquitous Computing, Springer Singapore, Singapore, 2017, pp. 497–504.

[14] D. Evans, D. Larochelle, Improving security using extensible lightweight static analysis, IEEE Software 19 (1) (2002) 42–51. doi:10.1109/52.976940.

[15] I. Gomes, P. Morgado, T. Gomes, R. Moreira, An overview on the static code analysis approach in software development.

[16] S. Wagner, J. Jürjens, C. Koller, P. Trischberger, Comparing bug finding tools with reviews and tests, in: F. Khendek, R. Dssouli (Eds.), Testing of Communicating Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 40–55.

[17] S. Panichella, V. Arnaoudova, M. D. Penta, G. Antoniol, Would static analysis tools help developers with code reviews?, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 161–170. doi:10.1109/SANER.2015.7081826.

[18] B. Chess, G. McGraw, Static analysis for security, IEEE Security Privacy 2 (6) (2004) 76–79. doi:10.1109/MSP.2004.111.

[19] D. Insa, J. Silva, Automatic assessment of Java code, Computer Languages, Systems & Structures 53 (2018) 59 – 72. doi:https://doi.org/10.1016/j.cl.2018.01.004.
URL http://www.sciencedirect.com/science/article/pii/S1477842417301045

[20] JetBrains, IntelliJ IDEA homepage (2000).
URL https://www.jetbrains.com/idea

[21] ISO/IEC, ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, Tech. rep. (2010).

[22] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, H. Watanabe, A framework for measuring and evaluating program source code quality, in: J. Münch, P. Abrahamsson (Eds.), Product-Focused Software Process Improvement, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 284–299.

[23] A. Costin, Lua code: Security overview and practical approaches to static analysis, in: Workshop on Language-Theoretic Security, 2017.

[24] B. Chess, J. West, Secure Programming with Static Analysis, 1st Edition, Addison-Wesley Professional, 2007.

[25] S. McConnell, Code Complete: A Practical Handbook of Software Construction, Code Series, Microsoft Press, 1993.
URL https://books.google.co.kr/books?id=lohA2aY9gu0C

[26] SmartThings community, Smartthings open-source DeviceTypeHandlers and SmartApps code (2015).
URL https://github.com/SmartThingsCommunity/SmartThingsPublic

[27] 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: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 361–378.
URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian

[28] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. D. McDaniel, A. S. Uluagac, Sensitive information tracking in commodity iot, CoRR abs/1802.08307. arXiv:1802.08307.
URL http://arxiv.org/abs/1802.08307

[29] Z. B. Celik, P. D. McDaniel, G. Tan, Soteria: Automated iot safety and security analysis, CoRR abs/1805.08876. arXiv:

1805.08876.
URL http://arxiv.org/abs/1805.08876
[30] Checkstyle, Checkstyle homepage (2001).
URL http://checkstyle.sourceforge.net/
[31] PMD, PMD homepage (2017).
URL https://pmd.github.io/
[32] FindBugs, FindBugs homepage (2015).
URL http://findbugs.sourceforge.net/

## Appendix A. SmartThings custom rules

Table A.9: Rules and their quality attributes

| Goal | Question | Metric | Rule | Source |
|---|---|---|---|---|
| Reliability | R1 | *Handle null values* | Custom rule | Best practices |
| Reliability | R1 | *Broken null check* | Default rule | Basic |
| Reliability | R2 | *Assignment in conditional* | Default rule | Basic |
| Reliability | R2 | *Bitwise operator in conditional* | Default rule | Basic |
| Reliability | R2 | *Broken oddness check* | Default rule | Basic |
| Reliability | R2 | *Comparison of two constants* | Default rule | Basic |
| Reliability | R2 | *Comparison with self* | Default rule | Basic |
| Reliability | R2 | *Constant if expression* | Default rule | Basic |
| Reliability | R2 | *Constant ternary expression* | Default rule | Basic |
| Reliability | R2 | *Empty for statement* | Default rule | Basic |
| Reliability | R2 | *Empty method* | Default rule | Basic |
| Reliability | R2 | *Empty while statement* | Default rule | Basic |
| Reliability | R3 | *Avoid recurring short schedules* | Custom rule | Best practices |
| Reliability | R3 | *Do not use busy loop* | Custom rule | Best practices |
| Reliability | R3 | *Avoid chained runIn() calls* | Custom rule | Best practices |
| Reliability | R4 | *Missing switch default* | Custom rule | Basic |
| Reliability | R4 | *Missing event handler* | Custom rule | Basic |
| Reliability | R4 | *Empty catch block* | Default rule | Basic |
| Reliability | R4 | *Empty else block* | Default rule | Basic |
| Reliability | R4 | *Empty finally block* | Default rule | Basic |
| Reliability | R4 | *Empty if statement* | Default rule | Basic |
| Reliability | R4 | *Empty switch statement* | Default rule | Basic |
| Reliability | R4 | *Empty try block* | Default rule | Basic |
| Reliability | R5 | *Do not use synchronized* | Custom rule | Best practices |
| Reliability | R5 | *Use consistent return values* | Custom rule | Best practices |
| Reliability | R5 | *Verify array index* | Custom rule | Best practices |
| Reliability | R5 | *Correct use of atomic state* | Custom rule | Best practices |
| Reliability | R5 | *Correct use of atomic state update* | Custom rule | Best practices |
| Reliability | R5 | *Random double coerced to zero* | Default rule | Basic |
| Reliability | R6 | *Event handler must have a single argument* | Custom rule | Basic |
| Reliability | R6 | *Unused array* | Default rule | Unused |
| Reliability | R6 | *Unused object* | Default rule | Unused |
| Reliability | R6 | *Duplicate map key* | Default rule | Basic |
| Reliability | R6 | *Duplicate set value* | Default rule | Basic |
| Reliability | R6 | *Return from finally block* | Default rule | Basic |
| Reliability | R6 | *Throw exception from finally block* | Default rule | Basic |
| Reliability | R6 | *Dead code* | Default rule | Basic |
| Reliability | R6 | *Parameter reassignment* | Default rule | Convention |
| 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 | *Separate parent and child apps* | Custom rule | Best practices |
| Maintainability | M2 | *For loop should be while loop* | Default rule | Basic |
| Maintainability | M2 | *Double negative* | Default rule | Basic |
| Maintainability | M2 | *Could be Elvis* | Default rule | Convention |
| Maintainability | M2 | *Inverted if else* | Default rule | Convention |
| Maintainability | M2 | *Ternary could be elvis* | Default rule | Convention |
| Maintainability | M2 | *Nested block depth* | Default rule | Size |
| Reliability | M2 | *Method size* | Default rule | Size |
| Reliability | M2 | *Method count* | Default rule | Size |

Continue on the next page

Table A.9: Rules and their quality attributes

| Goal | Question | Metric | Rule | Source |
|---|---|---|---|---|
| Maintainability | M2 | *Abc metric* | Default rule | Size |
| Maintainability | M2 | *Total lines of code* | - | Size |
| Security | S1 | *Do not hard-code SMS* | Custom rule | Security considerations |
| Security | S2 | *Document external HTTP requests* | Custom rule | Documentation |
| Security | S2 | *Document exposed endpoints* | Custom rule | Documentation |
| Security | S3 | *Subscriptions should be clear* | Custom rule | Security considerations |
| Security | S3 | *Subscriptions should be specific* | Custom rule | Security considerations |
| Security | S3 | *Do not use dynamic method execution* | Custom rule | Security considerations |
| Security | S4 | *Do not use restricted method calls* | Custom rule | Security considerations |

## Appendix B. Code defect density metrics

Table B.10: Official SmartApps with violations

| SmartApp | LOC | Input | Subscr. | Reliability | Maintainability | Security | Total |
|---|---|---|---|---|---|---|---|
| presence-change-push.groovy | 28 | 1 | 1 | 0 | 0 | 71.43 | 71.43 |
| let-there-be-light.groovy | 32 | 2 | 1 | 0 | 0 | 62.50 | 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 | 0 | 0 | 25.64 | 9.62 | 9.62 | 44.87 |
| energy-alerts.groovy | 73 | 5 | 1 | 27.4 | 13.7 | 0 | 41.1 |
| turn-on-only-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 |
| yoics-connect.groovy | 411 | 2 | 0 | 12.17 | 9.73 | 14.6 | 36.5 |
| life360-connect.groovy | 419 | 7 | 0 | 11.93 | 4.77 | 16.71 | 33.41 |
| tesla-connect.groovy | 338 | 3 | 0 | 2.96 | 8.88 | 20.71 | 32.55 |
| logitech-harmony-connect.groovy | 843 | 14 | 1 | 1.19 | 21.35 | 9.49 | 32.03 |
| garage-door-monitor.groovy | 94 | 3 | 1 | 0 | 10.64 | 21.28 | 31.92 |
| lock-it-when-i-leave.groovy | 63 | 4 | 1 | 0 | 0 | 31.75 | 31.75 |
| close-the-valve.groovy | 64 | 6 | 1 | 0 | 0 | 31.25 | 31.25 |
| cameras-on-when-im-away.groovy | 69 | 2 | 1 | 0 | 0 | 28.99 | 28.99 |
| lifx-connect.groovy | 424 | 1 | 0 | 7.08 | 11.79 | 9.43 | 28.3 |
| ecobee-connect.groovy | 673 | 1 | 0 | 1.49 | 17.83 | 8.92 | 28.24 |
| ifttt.groovy | 214 | 11 | 1 | 4.67 | 4.67 | 18.69 | 28.03 |
| greetings-earthling.groovy | 77 | 4 | 1 | 0 | 0 | 25.97 | 25.97 |
| withings.groovy | 398 | 1 | 0 | 15.08 | 0 | 7.54 | 22.62 |
| ubi.groovy | 437 | 5 | 1 | 6.86 | 4.58 | 9.15 | 20.59 |
| gentle-wake-up.groovy | 862 | 19 | 3 | 11.6 | 8.12 | 0 | 19.72 |
| ridiculously-automated-garage-door.groovy | 156 | 8 | 4 | 0 | 0 | 19.23 | 19.23 |
| foscam-connect.groovy | 163 | 3 | 1 | 6.13 | 12.27 | 0 | 18.4 |
| bon-voyage.groovy | 111 | 4 | 1 | 0 | 0 | 18.02 | 18.02 |
| elder-care-daily-routine.groovy | 113 | 6 | 0 | 8.85 | 8.85 | 0 | 17.7 |
| virtual-thermostat.groovy | 117 | 7 | 2 | 0 | 0 | 17.09 | 17.09 |
| carpool-notifier.groovy | 64 | 4 | 1 | 0 | 15.63 | 0 | 15.63 |
| withings-manager.groovy | 482 | 1 | 0 | 8.3 | 0 | 6.22 | 14.52 |
| smart-care-daily-routine.groovy | 156 | 7 | 0 | 6.41 | 6.41 | 0 | 12.82 |
| beacon-control.groovy | 253 | 15 | 1 | 7.91 | 0 | 3.95 | 11.86 |
| smart-security.groovy | 265 | 10 | 4 | 0 | 0 | 11.32 | 11.32 |
| keep-me-cozy-ii.groovy | 95 | 4 | 4 | 0 | 0 | 10.53 | 10.53 |
| medicine-reminder.groovy | 97 | 8 | 0 | 10.31 | 0 | 0 | 10.31 |
| sonos-music-modes.groovy | 211 | 8 | 1 | 4.74 | 4.74 | 0 | 9.48 |
| sonos-remote-control.groovy | 113 | 3 | 1 | 8.85 | 0 | 0 | 8.85 |
| nfc-tag-toggle.groovy | 113 | 7 | 2 | 0 | 0 | 8.85 | 8.85 |
| the-flasher.groovy | 118 | 9 | 5 | 0 | 0 | 8.47 | 8.47 |
| notify-me-when.groovy | 132 | 15 | 13 | 7.58 | 0 | 0 | 7.58 |
| button-controller.groovy | 273 | 19 | 1 | 7.33 | 0 | 0 | 7.33 |
| smart-nightlight.groovy | 141 | 9 | 5 | 0 | 0 | 7.09 | 7.09 |
| laundry-monitor.groovy | 144 | 6 | 2 | 0 | 6.94 | 0 | 6.94 |
| every-element.groovy | 472 | 4 | 0 | 6.36 | 0 | 0 | 6.36 |
| hue-mood-lighting.groovy | 286 | 32 | 15 | 3.5 | 0 | 0 | 3.5 |
| bose-soundtouch-control.groovy | 294 | 32 | 15 | 0 | 3.4 | 0 | 3.4 |

Table B.11: Community-created SmartApps with violations

| SmartApp | LOC | Input | Subscr. | Reliability | Maintainability | Security | 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 | 18.18 | 0 | 27.27 | 45.45 |
| 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 |
| weatherbug-home.groovy | 196 | 1 | 7 | 0 | 5.1 | 25.51 | 30.61 |
| simple-control.groovy | 633 | 16 | 2 | 15.8 | 9.48 | 4.74 | 30.02 |
| alfred-workflow.groovy | 141 | 2 | 0 | 14.18 | 7.09 | 7.09 | 28.36 |
| weather-windows.groovy | 108 | 8 | 1 | 0 | 27.78 | 0 | 27.78 |
| door-state-to-color-light-hue-bulb.groovy | 108 | 2 | 1 | 18.52 | 0 | 9.26 | 27.78 |
| simple-sync-connect.groovy | 324 | 2 | 1 | 21.6 | 6.17 | 0 | 27.77 |
| jawbone-up-connect.groovy | 401 | 0 | 0 | 2.49 | 2.49 | 22.44 | 27.42 |
| whole-house-fan.groovy | 74 | 7 | 4 | 0 | 0 | 27.03 | 27.03 |
| goodnight-ubi.groovy | 75 | 6 | 1 | 13.33 | 0 | 13.33 | 26.66 |
| my-light-toggle.groovy | 42 | 3 | 1 | 0 | 0 | 23.81 | 23.81 |
| opent2t-smartapp-test.groovy | 170 | 2 | 5 | 0 | 5.88 | 17.65 | 23.53 |
| door-jammed-notification.groovy | 44 | 2 | 1 | 0 | 0 | 22.73 | 22.73 |
| jenkins-notifier.groovy | 47 | 10 | 0 | 0 | 0 | 21.28 | 21.28 |
| thermostat-window-check.groovy | 94 | 6 | 2 | 0 | 0 | 21.28 | 21.28 |
| curb-energy-monitor.groovy | 401 | 1 | 0 | 4.99 | 4.99 | 9.98 | 19.96 |
| beaconthings-manager.groovy | 102 | 0 | 0 | 9.8 | 0 | 9.8 | 19.6 |
| quirky-connect.groovy | 727 | 2 | 0 | 6.88 | 4.13 | 8.25 | 19.26 |
| switch-activates-home-phrase-or-mode.groovy | 109 | 3 | 1 | 0 | 0 | 18.35 | 18.35 |
| smartblock-chat-sender.groovy | 55 | 1 | 1 | 0 | 0 | 18.18 | 18.18 |
| bright-when-dark-and-or-bright-after-sunset.groovy | 732 | 39 | 7 | 4.1 | 6.83 | 6.83 | 17.76 |
| jawbone-button-notifier.groovy | 61 | 5 | 1 | 0 | 0 | 16.39 | 16.39 |
| thermostat-auto-off.groovy | 63 | 3 | 1 | 0 | 0 | 15.87 | 15.87 |
| smartblock-linker.groovy | 127 | 4 | 4 | 0 | 0 | 15.75 | 15.75 |
| weather-underground-pws-connect.groovy | 66 | 4 | 0 | 0 | 0 | 15.15 | 15.15 |

Continue on the next page

Table B.11: Community-created SmartApps with violations

| SmartApp | LOC | Input | Subscr. | Reliability | Maintainability | Security | Total |
|---|---|---|---|---|---|---|---|
| routine-director.groovy | 261 | 11 | 3 | 7.66 | 0 | 3.83 | 11.49 |
| gidjit-hub.groovy | 182 | 3 | 0 | 5.49 | 0 | 5.49 | 10.98 |
| smart-humidifier.groovy | 92 | 6 | 1 | 10.87 | 0 | 0 | 10.87 |
| smartblock-notifier.groovy | 858 | 6 | 3 | 4.66 | 3.5 | 2.33 | 10.49 |
| medicine-management-contact-sensor.groovy | 98 | 3 | 1 | 0 | 0 | 10.2 | 10.2 |
| nobody-home.groovy | 119 | 3 | 3 | 0 | 0 | 8.4 | 8.4 |
| hello-home-phrase-director.groovy | 258 | 10 | 3 | 3.88 | 0 | 3.88 | 7.76 |
| vinli-home-connect.groovy | 139 | 2 | 0 | 0 | 0 | 7.19 | 7.19 |
| color-coordinator.groovy | 139 | 3 | 5 | 0 | 0 | 7.19 | 7.19 |
| thermostat-mode-director.groovy | 521 | 19 | 5 | 1.92 | 3.84 | 0 | 5.76 |
| vacation-lighting-director.groovy | 359 | 12 | 1 | 5.57 | 0 | 0 | 5.57 |
| smartblock-manager.groovy | 218 | 2 | 0 | 0 | 0 | 4.59 | 4.59 |
| simple-sync-trigger.groovy | 254 | 29 | 12 | 0 | 3.94 | 0 | 3.94 |

## Appendix C. A Detailed Descriptions of Metrics

Table C.12: A Detailed Descriptions of Metrics

| No | Source | Metric | Description In Detail |
|---|---|---|---|
| 1 | Best practices | Avoid recurring short schedules | Scheduled and other periodic functions should not execute more often than every five minutes, unless there is a good reason for it, and the reviewers agree.<br>If your code executes more frequently than every five minutes, add a comment to your code explaining why this is necessary. |
| 2 | Best practices | Do not use busy loop | Instead of trying to force a delay in execution, you should schedule a future execution of your app. |
| 3 | Best practices | Do not use synchronized | Concurrent executions of the SmartApp or Device Handler are not guaranteed, or even likely, to be executing on the same server. Because of this, trying to force synchronous behavior by using synchronized would only work in the rare occurrence that a concurrent execution happens on the same server, yet it always incurs overhead. |
| 4 | Best practices | Avoid chained runIn() calls | If for some reason it is necessary, add a comment describing why it is necessary. |
| 5 | Best practices | Use consistent return values | Despite the fact that Groovy is a dynamically typed language, a method should return a single type of data, regardless of if the method signature is typed or not. |
| 6 | Best practices | Verify array index | When parsing data, pay attention to arrays if you use them. Do not index into arrays directly without making sure that the array actually has enough elements. |
| 7 | Best practices | Handle null values | NullPointerExceptions are one of the most frequently occurring exceptions on the SmartThings platform - take care to avoid them! A NullPointerException will terminate the SmartApp or Device Handler execution, but can be avoided easily with the safe navigation (?) operator. Any code that may encounter a null value should anticipate and handle this. |
| 8 | Basic | Missing switch default | Make sure any if() or switch() blocks handle all expected inputs. Forgetting to handle a certain condition can cause unexpected logic errors. Also, every switch() statement should have a default: case statement to handle any cases where there is no match. |
| 9 | Documentation | Document external HTTP requests | HTTP requests to outside services should be documented, explaining the need to make external requests, what data is sent, and how it will be used. Please also include a comment with a link to the third party's privacy policy, if applicable. |
| 10 | Documentation | Document exposed endpoints | If your SmartApp or Device Handler exposes any endpoints, add comments that document what the API will be used for, what data may be accessed by those APIs, and where possible, include a link to the privacy policies of any remote services that may access those APIs. |
| 11 | Security considerations | Subscriptions should be clear | It is possible to subscribe to Events using a string variable, so what the SmartApp is subscribing to might be somewhat opaque. |
| 12 | Security considerations | Subscriptions should be specific | Do not create overly-broad subscriptions. A SmartApp that is subscribed to every location Event will execute excessively, and is rarely necessary. Instead, create subscriptions specific to the Event you are interested in. |

Continue on the next page

Table C.12: A Detailed Descriptions of Metrics

| No | Source | Metric | Description In Detail |
|---|---|---|---|
| 13 | Security considerations | Do not use dynamic method execution | In groovy you can execute functions based on a string, like ${mystring}(), which can be very handy, but when ${mystring} comes from a HTTP request, outside the SmartThings platform, or from another SmartApp or Device Handler, we need to validate the input. The preferred method of validation is to use a switch() statement on the input before doing anything with it. |
| 14 | Security considerations | Do not hard-code SMS | Notifications should never be sent to a hard-coded number. They should always use a number provided by the user using the contact input (even though Contact Book is not enabled, the contact input type is available and contains a fall-back mechanism for non-Contact Book users. Using this future-proofs your SmartApp). |
| 15 | Security considerations | Do not use restricted method calls | The SmartThings platform defines a subset of Groovy library APIs for development of SmartApps. Stick to the use of the APIs in the subset. |
| 16 | Best practices | Separate parent and child apps | The parent and child should exist in separate files. Putting the parent and child code in the same file leads to file size bloat, makes the code harder to understand, is error-prone, and difficult to debug. |
| 17 | Basic | Missing event handler | Avoid to leave no event handler for declared input devices. |
| 18 | Basic | Event handler must have a single argument | Every event handler must take one argument no matter whether or not it is used in the handler. |
| 19 | Best practices | Correct use of atomic state | Understand the difference between atomic state and state, make sure you use the correct one for your needs, and avoid using both in the same SmartApp. |
| 20 | Best practices | Correct use of atomic state update | Take care when storing collections in atomic state. Modifying collections in Atomic State does not work as it does with State. You will need to assign the collection to a local variable, make changes as needed, then assign it back to atomic state. |
| 21 | Unused | Unused array | Checks for array allocations that are not assigned or used, unless it is the last statement within a block (because it may be the intentional return value). |
| 22 | Unused | Unused object | Checks for object allocations that are not assigned or used, unless it is the last statement within a block (because it may be the intentional return value). |
| 23 | Basic | Assignment in conditional | An assignment operator (=) was used in a conditional test. This is usually a typo, and the comparison operator (==) was intended. |
| 24 | Basic | Bitwise operator in conditional | Checks for bitwise operations in conditionals. For instance, the condition if (a \| b) is almost always a mistake and should be if (a \|\| b). If you need to do a bitwise operation then it is best practice to extract a temp variable. |
| 25 | Basic | Broken null check | Looks for faulty checks for null that can cause a NullPointerException. |
| 26 | Basic | Broken oddness check | The code uses x % 2 == 1 to check to see if a value is odd, but this won't work for negative numbers (e.g., (-5) % 2 == -1). If this code is intending to check for oddness, consider using?x & 1 == 1, or x % 2 != 0. |

Continue on the next page

Table C.12: A Detailed Descriptions of Metrics

| No | Source | Metric | Description In Detail |
|---|---|---|---|
| 27 | Basic | Comparison of two constants | Checks for expressions where a comparison operator or equals() or compareTo() is used to compare two constants to each other or two literals that contain only constant values. |
| 28 | Basic | Comparison with self | Checks for expressions where a comparison operator or equals() or compareTo() is used to compare a variable to itself, e.g.: x == x, x != x, x <=>x, x <x, x >= x, x.equals(x) or x.compareTo(x), where x is a variable. |
| 29 | Basic | Constant if expression | Checks for if statements with a constant value for the if boolean expression, such as true, false, null, or a literal constant value. These if statements can be simplified or avoided altogether. |
| 30 | Basic | Constant ternary expression | Checks for ternary expressions with a constant value for the boolean expression, such as true, false, null, or a literal constant value. |
| 31 | Basic | Duplicate map key | A Map literal is created with duplicated key. The map entry will be overwritten. |
| 32 | Basic | Duplicate set value | A Set literal is created with duplicate constant value. A set cannot contain two elements with the same value. |
| 33 | Basic | Empty catch block | Checks for empty catch blocks. In most cases, exceptions should not be caught and ignored (swallowed). |
| 34 | Basic | Empty else block | Checks for empty else blocks. Empty else blocks are confusing and serve no purpose. |
| 35 | Basic | Empty finally block | Checks for empty finally blocks. Empty finally blocks are confusing and serve no purpose. |
| 36 | Basic | Empty for statement | Checks for empty for blocks. Empty for statements are confusing and serve no purpose. |
| 37 | Basic | Empty if statement | Checks for empty if statements. Empty if statements are confusing and serve no purpose. |
| 38 | Basic | Empty method | A method was found without an implementation. If the method is overriding or implementing a parent method, then mark it with the @Override annotation. |
| 39 | Basic | Empty switch statement | Checks for empty switch statements. Empty switch statements are confusing and serve no purpose. |
| 40 | Basic | Empty try block | Checks for empty try blocks. Empty try blocks are confusing and serve no purpose. |
| 41 | Basic | Empty while statement | Checks for empty while statements. Empty while statements are confusing and serve no purpose. |
| 42 | Basic | Random double coerced to zero | The Math.random() method returns a double result greater than or equal to 0.0 and less than 1.0. If you coerce this result into an Integer, Long, int, or long then it is coerced to zero. Casting the result to int, or assigning it to an int field is probably a bug. |
| 43 | Basic | Return from finally block | Checks for a return from within a finally block. Returning from a finally block is confusing and can hide the original exception. |
| 44 | Basic | Throw exception from finally block | Checks for throwing an exception from within a finally block. Throwing an exception from a finally block is confusing and can hide the original exception. |
| 45 | Basic | For loop should be while loop | A for loop without an init and update statement can be simplified to a while loop. |
| 46 | Basic | Double negative | There is no point in using a double negative, it is always positive. For instance !!x or !(!x) can always be simplified to x. |

Continue on the next page

Table C.12: A Detailed Descriptions of Metrics

| No | Source | Metric | Description In Detail |
|---|---|---|---|
| 47 | Basic | Dead code | Dead code appears after a return statement or an exception is thrown. If code appears after one of these statements then it will never be executed and can be safely deleted. |
| 48 | Convention | Parameter reassignment | Checks for a method or closure parameter being reassigned to a new value within the body of the method/closure, which is a confusing and questionable practice. Use a temporary variable instead. |
| 49 | Convention | Confusing ternary | In a ternary expression avoid negation in the test. For example, rephrase: (x != y) ? diff : same as: (x == y) ? same : diff. Consistent use of this rule makes the code easier to read. Also, this resolves trivial ordering problems, such as "does the error case go first?" or "does the common case go first?". |
| 50 | Convention | Could be Elvis | Catch an if block that could be written as an elvis expression. |
| 51 | Convention | If statement could be ternary | An if statement where both the if and else blocks contain only a single return statement returning a constant or literal value. A block where the second-to-last statement in a block is an if statement with no else, where the block contains a single return statement, and the last statement in the block is a return statement, and both return statements return a constant or literal value. |
| 52 | Convention | Inverted if else | An inverted if-else statement is one in which there is a single if statement with a single else branch and the boolean test of the if is negated. For instance if (!x) false else true. It is usually clearer to write this as if (x) true else false. |
| 53 | Convention | Ternary could be elvis | Checks for ternary expressions where the boolean and true expressions are the same. These can be simplified to an Elvis expression. |
| 54 | Size | Cyclomatic complexity | Calculates the Cyclomatic Complexity for methods/classes and checks against configured threshold values. The maximum cyclomatic complexity value allowed for a single method (or "closure field") is 20. The maximum average cyclomatic complexity value allowed for a class, calculated as the average complexity of its methods or "closure fields" is 20. The maximum total cyclomatic complexity value allowed for a class, calculated as the total complexity of its methods or "closure fields" is 0. |
| 55 | Size | Nested block depth | Calculate the nested block depths. |
| 56 | Size | Method size | Calculate the method sizes. |
| 57 | Size | Method count | Checks if the number of methods within a class exceeds the number of lines specified by the maxMethod property. A class with too many methods is probably a good suspect for refactoring, in order to reduce its complexity and find a way to have more fine grained objects. The maximum number of methods allowed in a class definition is 30 by default. |
| 58 | Size | Abc metric | Calculates the ABC size metric for methods/classes and checks against configured threshold values. The maximum ABC score allowed for a single method (or "closure field") is 60. The maximum average ABC score allowed for a class, calculated as the average score of its methods or "closure fields" is 60. The maximum ABC score allowed for a class, calculated as the total ABC score of its methods or "closure fields" is 0. |
| 59 | Size | Total lines of code | Calculate the total lines of code. |