Static Checking of Samsung IoT Device Relationships

Yifei Yang

Github: `https://github.com/yifeiy3/SmartappAnalysis`

For my final project, I implemented a simple static checker for Internet of Things (IoT) devices relationships in a Samsung Smartthings environment. Aiming to detect potential security risks and abnormal behaviors in the environment, the checker parses the source code for the environment's Smartapps and generates an abstract syntax tree (AST) for each app. Through the ASTs, the checker builds a relationship graph for the devices in the environment to analyze for hidden and anomalous interactions. Section 1 of the report will give a brief background introduction and my motivation for building the checker. Then, section 2 of the report will describe the details of the checker's implementation. Finally, section 3 of the report will elaborate on our checker's performance on simple test cases and section 4 on potential future work.

1 Background

1.1 IoT and Samsung Smatthings Environment

IoT is a blanket term for various gadgets that most people don't think as computers but still have the processing power and internet connection, such as home routers and household smart kitchenware. Samsung Smartthings is a platform for establishing an IoT environment, where each IoT device is called a "Smart device" that can be either controlled physically or through apps called "Smartapp". Due to devices' access to internet connection, the Smartapps are able to automatically change a device's behavior when conditions for the app to run are met. Thus, instead of having each device being isolated by itself in a traditional household setting, the Smartapps are able to connect the devices together through their specified interactions that our static checker aims to analyze.

1.2 Motivation and Related Work

As IoT devices become increasingly popular with its predicted 22 billion worldwide devices by 2025, the IoT systems each device lies in is getting significantly more diverse and complicated, which could lead to potential security risks. In complex IoT environments, it is often difficult to account for all added interactions when a new Smartapp is introduced, as many interactions are hidden. For example, in figure 1 we have an environment with a "Homemode" app that turns the oven to heating. Now, while adding another "FireAlarm" app that unlocks the door when smoke is detected to the environment may sound perfectly normal, suddenly, our "Homemode" app is able to unlock the door since oven heating releases smoke. This hidden interaction could lead to a malicious attacker finding a way to trigger the "Homemode" app to get in the house, while the "FireAlarm" app could be perfectly secure.

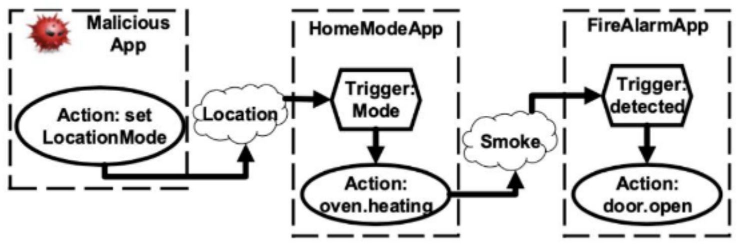


Figure 1: Hidden Interaction Example[1]

Previous work by IoTCOM[2] implemented a static checker for such potential security risks due to hidden, anomalous, and conflicting interactions within an IoT system. Their work has inspired my current fifth-year master research, where I am creating a runtime checker that learns the device interactions through environment's runtime behavior, and aims to prevent abnormal device changes due to these potential security risks as the change command is being issued. As a result, I am very fascinated by my current runtime checker's static counterpart, especially after taking this course, which motivates me in pursuing this final project topic in reproducing IoTCOM's static checker on checking a subset of behaviors with a potential security risk.

2 Implementation Details

As shown in the workflow graph below. For each Smartapp, we construct an AST from lexing and parsing its source file, then analyzes AST to store necessary information to build our relationship graph. After analyzing all the ASTs, we build the relationship graph and analyzes for anomalous and hidden behaviors.

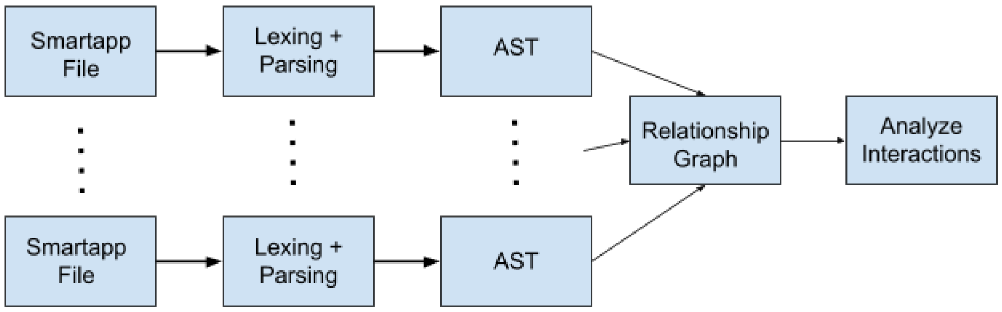


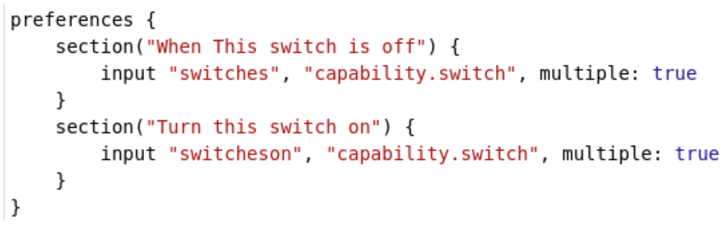
Figure 2: Implementation Workflow

2.1 Lexing and Parsing

The Samsung Smartapps are written in the groovy language, for which the static checker constructed a lexer and parser of using Python's PLY package. For the purpose of building a relationship graph between the devices in the environment, we would only need to parse a small subset of the language, specifically the three key components of a Smartapp listed below. For rest of the statements in a Smartapp file, our parser simply skips them as they are irrelevant for the scope of our static analysis.

2.1.1 Preference Block

Each Smartapp is required to have a preference block, which specifies what devices our Smartapp associates with. Each smartapp can only change the devices associated with it described in the preference block. The block is composed of a list of section statements, which within each statement the program asks device inputs with the associated capability that is given as a string parameter. The multiple field determines whether the section can associate with multiple input devices, and the required field determines whether we need to associate at least 1 device. Finally, all the devices associated with the section is assigned to the variable with name determined by the string after input field. So in figure 3, we would reference the devices associated with the first section with variable **switches**, and for second section with variable **switcheson**. When parsing the preference block, we store the variable name for each section's associated device.

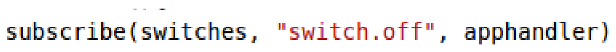


```
preferences {
    section("When This switch is off") {
        input "switches", "capability.switch", multiple: true
    }
    section("Turn this switch on") {
        input "switcheson", "capability.switch", multiple: true
    }
}
```

Figure 3: PreferenceBlock

2.1.2 Subscribe Function

The subscribe function specifies the device changes events each Smartapp handles. The subscribe function takes in 3 parameters, the first parameter specifies the device, the second specifies the specific change in state for the device, and the last specifies the handler function for such event. For example, in figure 4, the subscribed function is translates to "When devices corresponding to input variable switches gets turned off, handle the event with apphandler function".



```
subscribe(switches, "switch.off", apphandler)
```

Figure 4: Subscribe Function

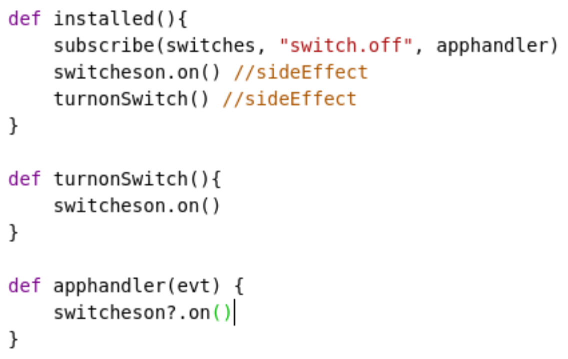
2.1.3 Smartapp Functions

Each Smartapp is required to have an Installed() and Updated() function, which initializes/updates the Smartapp to subscribe to specific device changes through the subscribe function. The updated function should always subscribe to the same input variable as the installed function, since each update to the Smartapp should only change the devices associated with input, not changing the smartapp behavior directly.

Within each function, it is possible to change a device state by simply calling a function on the device corresponding to the state change. For example, a device with switch capability can be turned on or off, and calling "switcheson.on()" in figure 5 will turn all the input devices associated with variable name switcheson on. Thus, for our static checker, we would only need to parse all the function call statements to get the control flow of the Smartapp and all the possible device changes it may have.

Finally, each Smartapp is also able to cause side effects, which it may change some device state without any subscribed device changing state. As shown in figure 5. In the installed function we can directly turn switcheson on or indirectly call a function to change switcheson to on without calling the handling function.

For the scope of our static analysis, we assume that Installed(), Updated(), and subscribed device changes are the only entry points of the remaining functions. That is, all other functions within the Smartapp are called either directly by the two function, or by handling the event specified by the subscribe function within the function reached by the two functions. We also assumed that the smartapp functions are simple, that is, there are no cycling in function calls where function A calls B and B, after a number of function calls, calls back A again.



```
def installed(){
    subscribe(switches, "switch.off", apphandler)
    switcheson.on() //sideEffect
    turnonSwitch() //sideEffect
}

def turnonSwitch(){
    switcheson.on()
}

def apphandler(evt) {
    switcheson?.on()
}
```

Figure 5: Smartapp Functions and SideEffects

2.2 AST

After parsing the program by described above, for each Smartapp, we will have an AST with a parsed preference block with its corresponding input sections, and a list of function definitions, within each only function call statements will be parsed. For each parsed Smartapp, we store the following information by analyzing its AST:

1. All the device input variables specified in the preference block.

2. For each subscribe function call within each function, say **subscribe(obj, state, handler)**, we store the function **handler** to a list containing the handler functions for the device state change tuple **(obj, state)**
3. For each other function call within each function **F**, if it is called by an object, (i.e.: object.functionName()), check if the object is corresponding to a device input variable. If so, add (object, functionName) a list storing the possible device changes **F** can achieve.
4. For each other function call within each function **F** that is not called by an object, say **G()**, check if **G** is in the list of defined function within the Smartapp. If so, add **G** a list storing the functions **F** can call.

Since we assumed there is no cycles in function calls within each Smartapp, we can construct function call trees for each entry point of the smartapp, which we have function F is G's parent if G is in the list storing the function F can call. Then, we can recurse down the tree structure to accumulate the information stored by 3. above, which we have for each function F, the list storing the possible device changes F can achieve would contain the device changes F does and all the device changes F's children function can do. By our assumption, only Installed(), Updated(), and subscribed device handlers can be the entry point of the Smartapp. Thus, after tree recursion, we are able to construct a dictionary that maps each entry point of the Smartapp to the possible device changes it can achieve and use it to construct our relationship graph.

As a final note, all the possible device changes starting from entry point Installed() or Updated() function are side effects, as no device change is needed for them to happen.

2.3 Relationship Graph

After constructing the dictionary for all the Smartapps in the environment, we build a relationship graph using tuple **(DeviceA, StateA)** as graph nodes. Shown in figure 6 below, we form an edge **S** between **(DeviceA, StateA)** and **(DeviceB, StateB)** if changing DeviceA to StateA can cause a change for DeviceB to State B through Smartapp S.

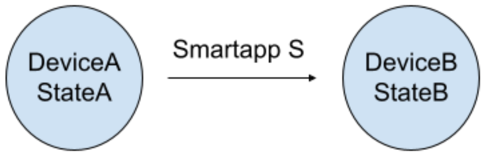


Figure 6: Relationship Graph Construction

To obtain all the device and state tuples, we look through dictionary for all the Smartapps from analyzing our AST, and check for all the (device, state) tuple the Smartapps subscribed to and can change from their entry point. Since Smartapps are the only medium to construct device interactions, for all the other possible (device, state) combinations, we can simply ignore them as there are no interactions for them in our environment. Finally, we would also need to add a dummy node for all the possible side effects in the Smartapps, since they do not need any device change to happen to change device states.

To build the relationship graph, we use a depth-first search algorithm for each node in the graph. Each time we reach a node, we append the paths to a list that stores all possible paths from our starting node to the node, and begin exploring another path if the node we reach is already visited. By our construction, if there is a path between (deviceA, stateA) to (deviceB, stateB), this means it is possible for deviceA changing to stateA to cause a chain of interaction by Smartapps for deviceB to change to stateB. Finally, if there is a path between the side effect dummy node to (deviceA, stateA), that means we can change deviceA to stateA without any preconditions by some Smartapp side effects.

2.4 Analyzing Interactions

After constructing our relationship graph between devices in the environment, we are able to analyze their interactions similar to what is done by the work of IoTCOM[2]. Due to time constraint for the project, the static checker currently is only able to check a subset of anomalous interactions described by IoTCOM, listed below.

2.4.1 Circularity

One common anomalous interaction comes with increasingly complex environments is circularity. As shown by figure 7, when a device changes to state A, due to the complex environment, there may be an overlooked path that lead to device changes to state B. This may result in unintended behavior that nullifies our change to state A.

To check for this interaction, we checks for every pair of (device, state) node that has the same device to see if there are paths between them. Such paths are the circularity paths we want to avoid.

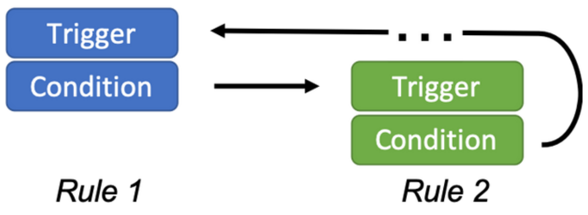


Figure 7: Circularity Interaction

2.4.2 Direct Conflict

Another common anomalous interaction is direct conflict. As shown by figure 8, there could be two different paths that can change a device to different states at the same time, which leads to undefined behavior in the environment. To check for this interaction, we simply need to check all the paths in the list that connects to (device, state A), and there will be a potential direct conflict for each path in the list to any paths in the list connects to the same device with a different state.

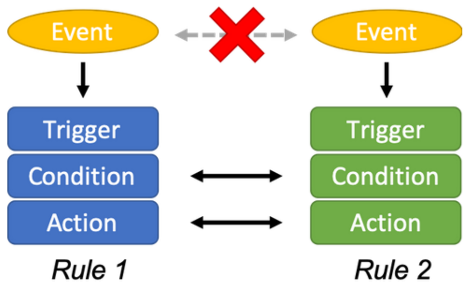


Figure 8: Direct Conlict Interaction

2.4.3 Hidden Interaction

Finally, our checker is able to check for hidden abnormal interaction within the environment similar to the one shown previously in figure 1 by directly presenting to the user our relationship graph. For the context in figure 1, there would be a path between (Oven, heating) to (Door, open), which the user would identify as a security risk.

3 Performance on Simple Test Cases

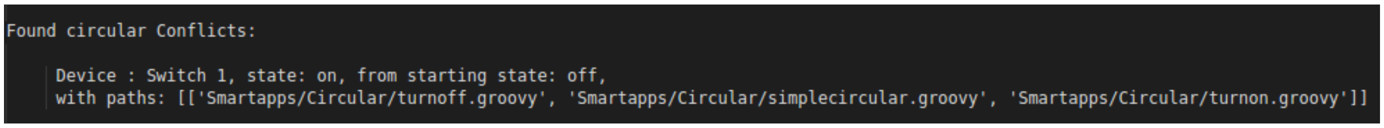
To evaluate the performance of the checker, I have created three simple test Samsung Smartthings environment that each contains a circularity, direct conflict, and hidden interaction correspondingly. To verify the correctness of the Smartapp and generated behaviors, each test environment is simulated on the Samsung Smartthings Developer Platform with virtual devices [3]. The checker is then able to generate a log describing the relationship graph and potential anomalous interactions as described in the previous section. The details for each case is shown below.

3.1 Circularity

To check our detection for circularity, we created an environment where we have three switches and Smartapps, with specified interaction below:

1. App ”Turnoff.groovy”: Switch 1 off $\implies$ Switch 3 off
2. App ”Simplecircular.groovy”: Switch 3 off $\implies$ Switch 2 off
3. App ”Turnon.groovy”: Switch 2 off $\implies$ Switch 1 on

From the interactions, we can see that turning Switch 1 off can directly cause Switch 1 to turn back on again due to circularity. Our checker detects such behavior in the generated log below:



```
Found circular Conflicts:

    Device : Switch 1, state: on, from starting state: off,
    with paths: [['Smartapps/Circular/turnoff.groovy', 'Smartapps/Circular/simplecircular.groovy', 'Smartapps/Circular/turnon.groovy']]
```

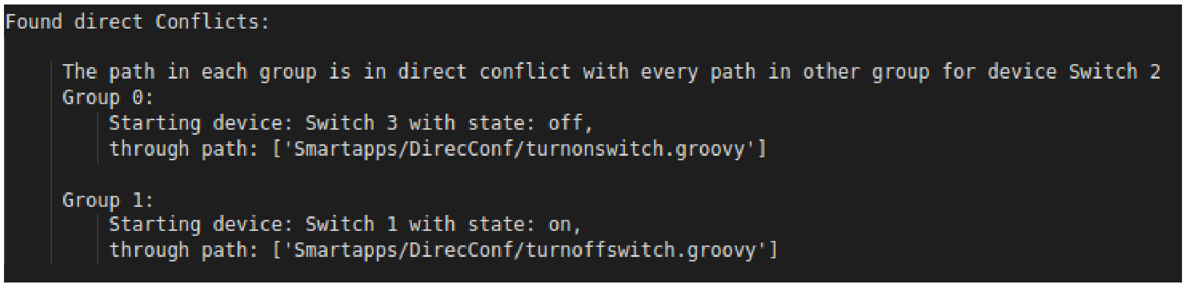
Figure 9: Checker Output for Circularity Test Case

3.2 Direct Conflict

To check our detection for circularity, we created an environment where we have three switches and two Smartapps with conflicting specifications. The interactions is shown below:

1. App ”Turnoffswitch.groovy”: Switch 1 on $\implies$ Switch 2 off
2. App ”Turnonswitch.groovy”: Switch 3 off $\implies$ Switch 2 on

From the interactions, we can see that in the event where Switch 1 turns on and Switch 3 turns off in close time proximity will lead to the Smartapps trying to turn Switch 2 on and off at the same time due to direct conflict. Our checker detects such behavior in the generated log below:



```
Found direct Conflicts:

    The path in each group is in direct conflict with every path in other group for device Switch 2
    Group 0:
        Starting device: Switch 3 with state: off,
        through path: ['Smartapps/DirecConf/turnonswitch.groovy']

    Group 1:
        Starting device: Switch 1 with state: on,
        through path: ['Smartapps/DirecConf/turnoffswitch.groovy']
```

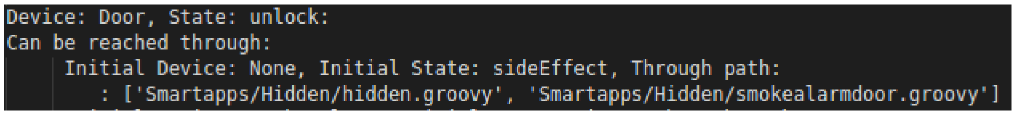
Figure 10: Checker Output for Direct Conflict Test Case

3.3 Hidden Interaction

To check our detection for hidden behavior, we created an environment similar to our example in Figure 1 with replacing the oven app to an app that turns switches on. The interactions is shown below:

1. App ”Hidden.groovy”: Switch 1 on $\implies$ Switch 2 off, with side effect of turning on Smoke Alarm
2. App ”Smokealarmdoor.groovy”: Smoke Alarm siren $\implies$ Door unlock

Due to its side effect, ”Hidden.groovy” Smartapp has a hidden interaction of unlocking the door, even though the intention of the app is only to turnoff a switch. Our checker detects such hidden behavior through the paths in the relationship graph between devices in the generated log below:



```
Device: Door, State: unlock:
Can be reached through:
    Initial Device: None, Initial State: sideEffect, Through path:
        : ['Smartapps/Hidden/hidden.groovy', 'Smartapps/Hidden/smokealarmdoor.groovy']
```

Figure 11: Checker Output for Hidden Interaction Test Case

4 Future Work

Due to time limitations, the static checker is fairly minimal, and currently only operates under simple environments under the assumptions listed below:

1. Installed(), Updated() and subscribed device changes are the only entry points of the Smartapp.
2. There are no cycling in function calls within a Smartapp where function A calls B and B calls back A after a number of function calls.
3. The device state can only be changed in the Smartapp through calling "Device.statevalue()" function as described in section 2.1.3.

Environments under these assumptions only cover a fairly small subset of the functionalities offered by Samsung Smartthings environment. For example, it is possible to have a Smartapp respond to a specific time of the day or respond to requests from other public servers, both are entry points vulnerable to security risks that our checker currently does not support. To accomodate for these features, we would simply need to expand our language specified in lexing and parsing when building our AST, which could be done in the future.

Finally, the static checker is not sound, since we assumed all the device change calls within a function is reachable. However, it is possible to have multiple control flows within a Smartapp function that the checker currently does not account for, such as when device A changes to state A, the app changes device B to state B only if device C is at state C. While a perfectly sound static checker is difficult to achieve, the checker's soundness can be improved by expanding our language to account for these control flows when performing the analysis.

Acknowledgements: Thank you 17355 staff for giving me this wonderful opportunity to explore static checking on Samsung IoT environment, and thank you for a wonderful and rewarding class this semester.

Citations: [1] IOTCOM: Compositional Safety Analysis of IoT systems, still under review.
[2] "IoTCOM." Google Sites, sites.google.com/view/iotcom/home.
[3] SmartThings. Add a Little Smartness to Your Things., graph.api.smartthings.com/ide/apps.