I Introduction

Node-RED is a powerful tool used in building Internet of Things (IoT) applications with a focus on simplifying the ’wiring together’ of code blocks to carry out tasks. It also uses a visual programming approach that allows developers to connect predefined code blocks, known as ’nodes’, together to perform a task.
When wired together, the connected input, processing or output nodes make up ’flow’. Importantly, Node-RED has rapidly developed a significant and growing user base and an active developer community which is helping to create new nodes that allow programmers to reuse Node-RED code for a wide variety of tasks.
In order to secure all parts (nodes) contained in the work flow, end-to-end security is required and implies tracking information flow through all the system services and analyzing data dependencies .
Typically, forwarding confidential data to unauthorized services has to be detected. Consequently, data dependence has to be identified and explicitly exposed, otherwise, security composition can induce security leaks called interference.
For data dependency, we propose SEMIoT (Security Manager for Internet of things) that generates a security data graph. Starting from a Node-RED work-flow and its hidden steps, in fact SEMIoT calculates this data flow and creates a new Node-RED work-flow representing the DDG[1].
This document classifies Node-RED nodes and explains the SEMIoT graph generation .A use-case is illustrated

I.1 SEMIoT Node

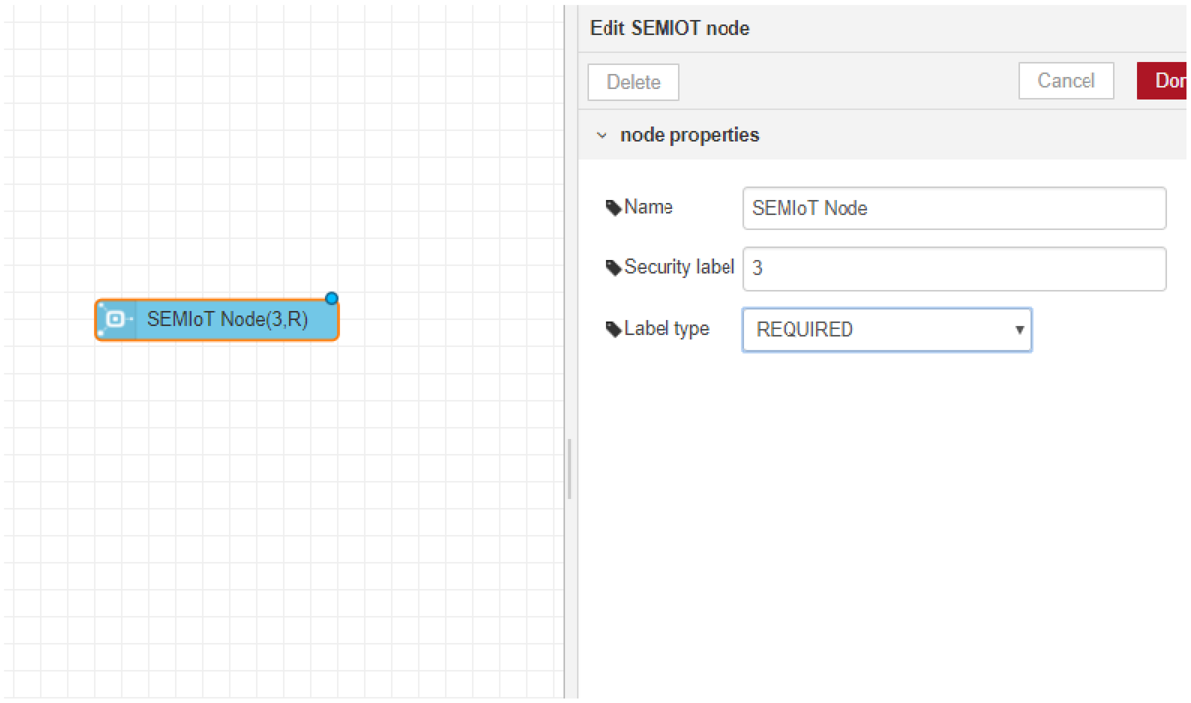
In order to secure the work flow ,the creation of a security graph generator by Node-RED is necessary.
Each node is called SEMIoT node and has two attributes :

Security label : represents the security degree of the data, this degree increases with the confidentiality of the inputs.

Label type : can be either required or provided labels depending on the type of services (managed or external). Required labels are immutable and represent external constraints that simply cannot be changed by the administrator. In contrary, Provided labels are set on managed resources and can be modified by the administrator when needed.

[1]DDG: Data Dependence Graph

Figure1 represents SEMIoT user interface for configuring a SEMIoT node. This interface contains the graphical representation of a SEMIoT node, the security label and type are displayed (in the figure, security label is 3 and label type is required)



Name: SEMIoT Node
Security label: 3
Label type: REQUIRED

Figure 1: A SEMIoT node configuration

I.2 Nodes classification

In order to generate a security graph from the work flow, we need to classify all the IoT system's nodes into three groups:

Input nodes: an input node is easy to identify as it only has one connecting dot on the right-hand side (the output connector) and a missing connecting dot on the left (the input connector). Though it can't be connected to another input node due to a lack of input connector , such nodes has also the ability to :

- Trigger the start of a flow and create the initial information packet
- To get triggered from an external operation such as a HTTP request or a trigger button.

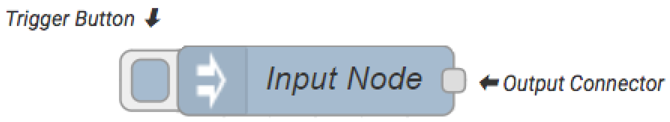
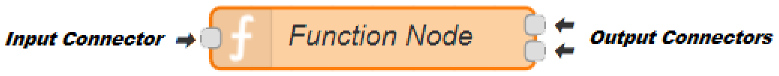


Figure 2: Input node

Input/Output Nodes : For example, we have as illustartes in figure 3, a function node generally contains:

- A single input connector on which it receives data ,executes the function code using the input information.
- One(or more) output connector(s) used to send data to the next node.



graph generator/sub.png

Figure 3: Function Node

Output Nodes : an output node sends data from the work-flow to external services. It can be spotted as it is missing the output connector.

Input Connector
Output Node

Figure 4: Nodes

Note: In a work-flow, the system architecture, node properties and connectors are discriped in JSON[2] code.
This JSON code containes other extra hidden objects like flow ID, MQTT broker and KAFKA server. These objects need to be explicitly exposed when generating the DDG.

I.3 Security graph generator

For SEMIoT user, the security graph generation process is descriped as Node-RED work-flow. As presented In figure 5, it contains 3 nodes:

IoT Work-Flow: is a flow reader node. It retrieves the user IoT work-flow from a Node-Red server and returns the JSON code of the work-flow.

Graph Generator: is a function node that contains a JavaScript function block that takes the work flow's JSON code and uses it to generate the DDG to finally return it in another JSON code.

SEMIoT Graph: is a file node that writes the returned code on a file, exports it and gets the DDG.

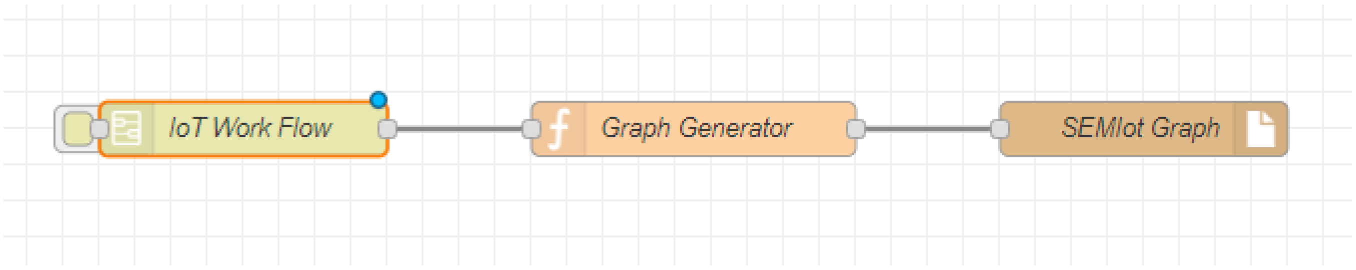


Figure 5: A graph generation process described as a Node-RED work-flow

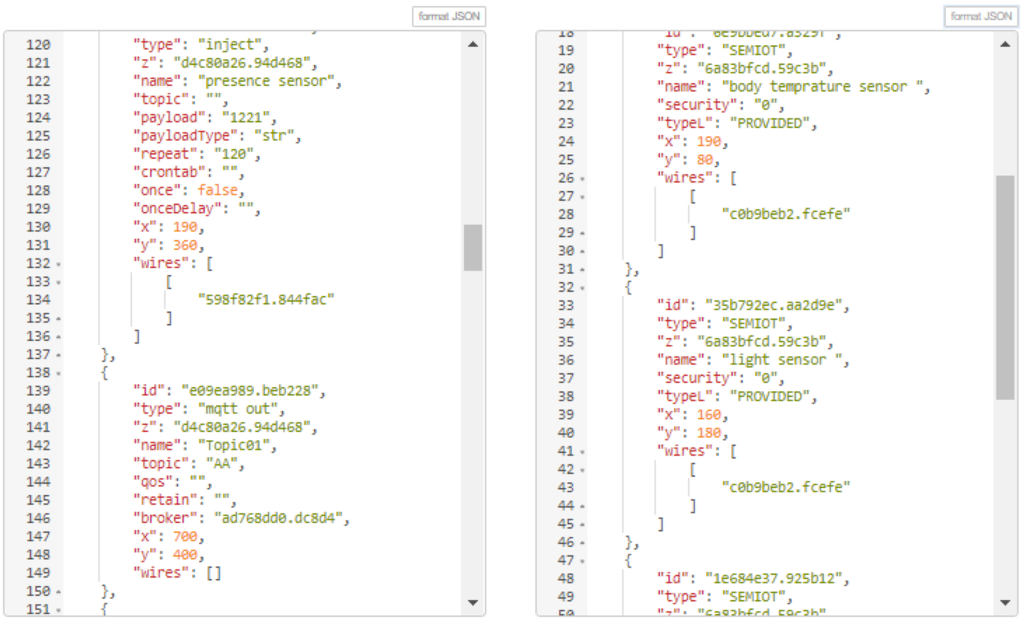
[2]JavaScript Object Notation

I.4 Algorithm

Input : an IoT work-flow's JSON code .

The figure 6-a represents an extract of input code which contains an array of objects .Each object presents a node's definition (name, type, position in the flow (x and y), wires...)

Output : a DDG JSON code. It contains the same objects but after removing the extra nodes, generating the DDG. Just like in the figure 6-b .



```
        "type": "inject",
        "z": "d4c80a26.94d468",
        "name": "presence sensor",
        "topic": "",
        "payload": "1221",
        "payloadType": "str",
        "repeat": "120",
        "crontab": "",
        "once": false,
        "onceDelay": "",
        "x": 190,
        "y": 360,
        "wires": [
            [
                "598f82f1.844fac"
            ]
        ]
    },
    {
        "id": "e09ea989.beb228",
        "type": "mqtt out",
        "z": "d4c80a26.94d468",
        "name": "Topic01",
        "topic": "AA",
        "qos": "",
        "retain": "",
        "broker": "ad768dd0.dc8d4",
        "x": 700,
        "y": 400,
        "wires": []
    },
    {
```

```
        "type": "SEMIOT",
        "z": "6a83bfcd.59c3b",
        "name": "body temprature sensor ",
        "security": "0",
        "typeL": "PROVIDED",
        "x": 190,
        "y": 80,
        "wires": [
            [
                "c0b9beb2.fcefe"
            ]
        ]
    },
    {
        "id": "35b792ec.aa2d9e",
        "type": "SEMIOT",
        "z": "6a83bfcd.59c3b",
        "name": "light sensor ",
        "security": "0",
        "typeL": "PROVIDED",
        "x": 160,
        "y": 180,
        "wires": [
            [
                "c0b9beb2.fcefe"
            ]
        ]
    },
    {
        "id": "1e684e37.925b12",
        "type": "SEMIOT",
```

Figure 6: a-Extract of an input JSON Code
b-Extract of an output JSON Code

Graph generator code : Our generator function contains an implementation in javascript of this algorithm below(Algorithm 1)

Input : IoTG (IoT graph)

Output : DDG (Data Dependence Graph)

Algorithm 1 DDG generator

```
1:  for each $node\ n \in IoTG$ do   ▷ make wires between mqtt(or kafka) nodes with the same topic
2:      if $n\ is\ MQTT(or\ KAFKA)\ input\ Node$ then
3:          for each $node\ M \in IoTG$ do
4:              if $M\ is\ MQTT(or\ KAFKA)\ output\ Node$ then
5:                  if $(M\ and\ n \in Same\ Broker)\ and\ (M\ and\ n \in Same\ Topic)$ then
6:                      $add(M, wires(n))$   ▷ wires(n): list of succesors nodes in the IoTG
7:                  end if
8:              end if
9:          end for
10:     end if
11: end for
12: for each $n \in IoTG$ do   ▷ begin the generation
13:     if $n\ is\ input\ Node$ then
14:         for each $output_i \in n$ do
15:             $create\ node\ n_i$
16:             $add(n_i, DDG)$
17:         end for
18:     else if $n\ is\ output\ Node$ then
19:         $add(n, DDG)$
20:     else if $n\ is\ input/output\ Node$ then
21:         $add(n, DDG)$
22:         for each $output_i \in n$ do
23:             $create\ node\ n_i$
24:             $add(n_i, DDG)$
25:             for each $j \in wires(output_i)$ do
26:                 $add(j, wires(n_i))$
27:             end for
28:             $add(n_i, wires(n))$
29:         end for
30:     end if
31: end for
32: return $DDG$
```

The algorithm 1 handles 2 cases:

output nodes: in this case, this node just replaced by a single SEMIoT node without any modification just like represents in the figure below.

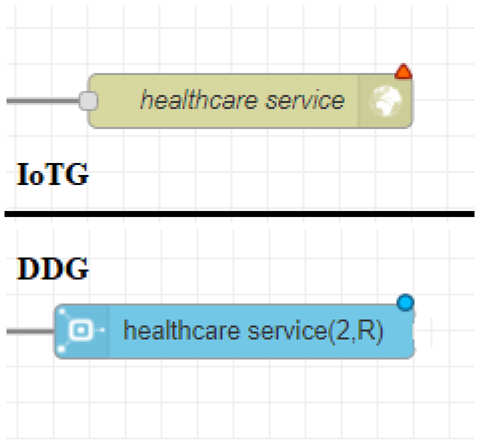


Figure 7: Output node and his correspendant in DDG

Input/ouput nodes: in this case, each output connector will replaced by a SEMIoT node without forget to set to this node the same wires of the connector.

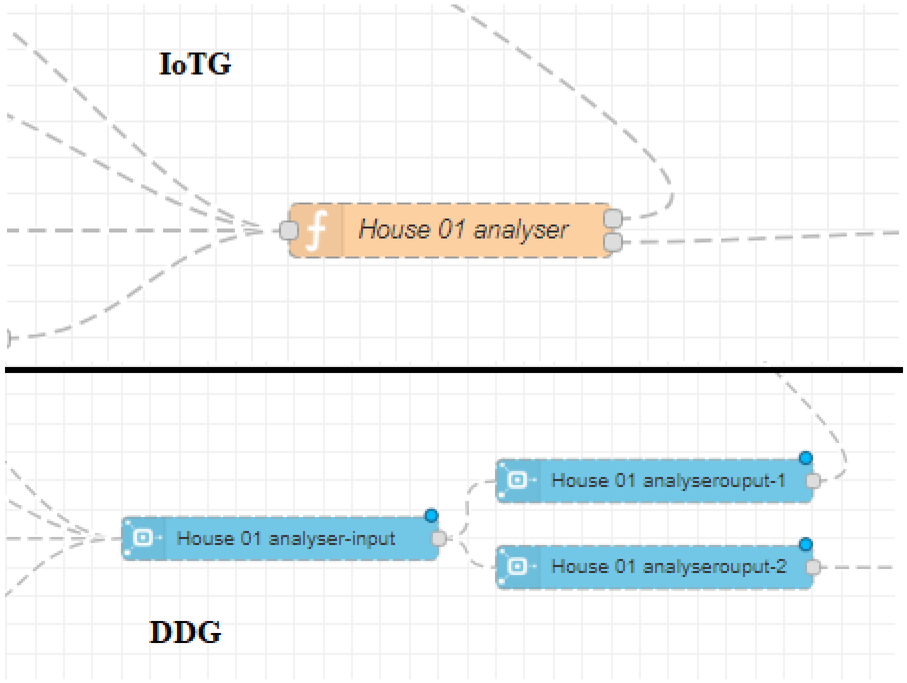


Figure 8: Input/output node and his correspendant in DDG

II Use-case: Home automation

In figure 9, we present a work flow of home automation use case. To make it simple, we created a sub-flow that contains all the nodes necessary to make a smart house. This smart house(presented in figure 10)holds the sensors and the actuators necessary for home observation remote.
”House 01” and ”House 02” are 2 sub-flows deployed in the gateways.
Master work-flow is deployed in the server-side. It communicates with a set of remote services like:

- healthcare service
- emergency service
- storage service
- payment service
- research lab

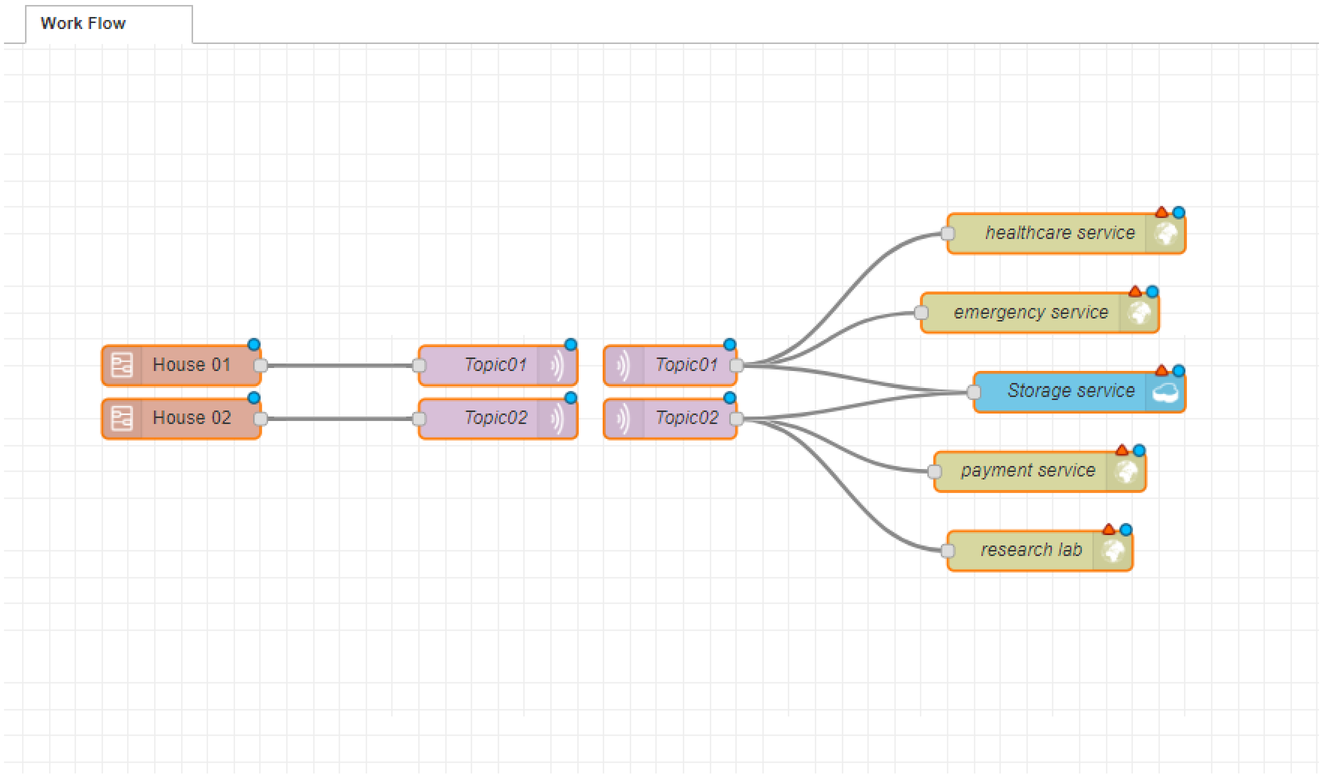


Figure 9: Home automation work-flow

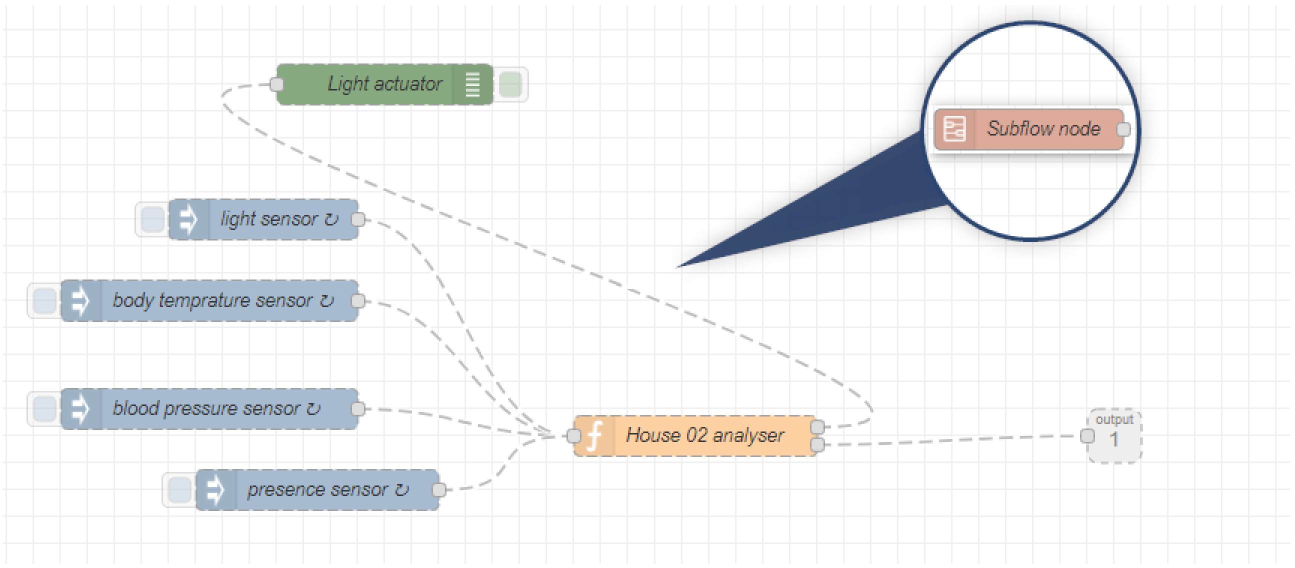


Figure 10: House 01 subflow

In figure 11, we present the DDG which creates wires between the MQTT nodes and replace each one with an SEMIoT input node and SEMIoT output node to set the security parameters without changing the subflow nodes. But as we can see in figure 12, the nodes in the sub flow were also replaced with an SEMIoT node.

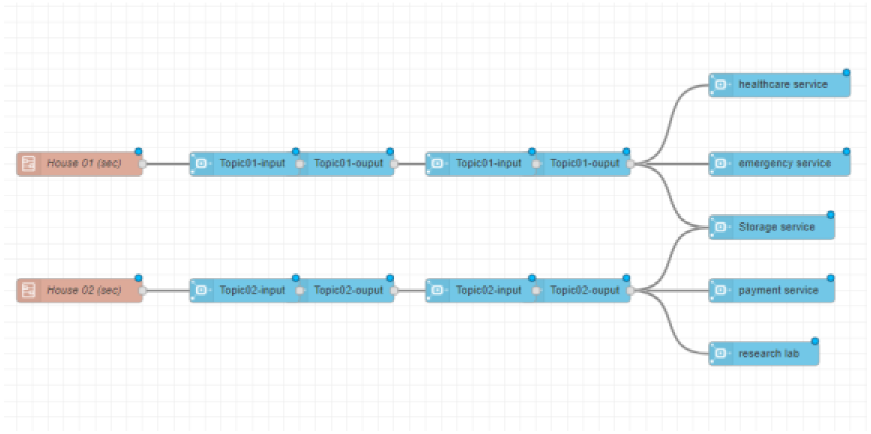


Figure 11: DDG for the master flow

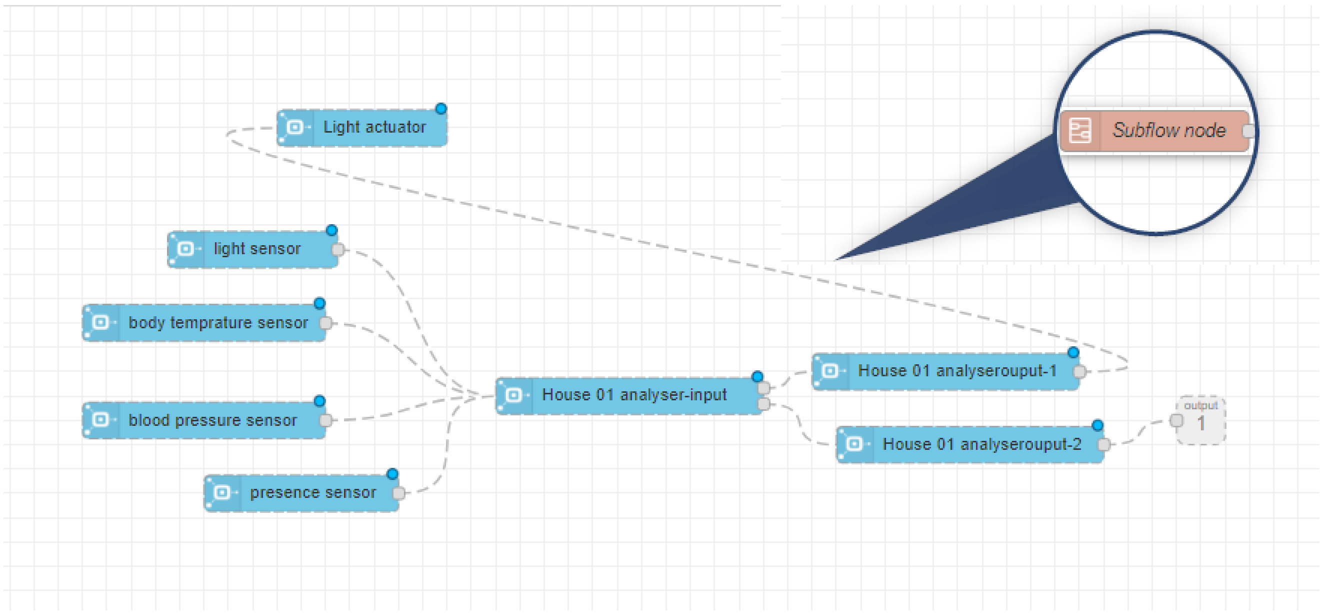


Figure 12: DDG for the sub-flow