NetSim

Accelerate Network R & D

DIO Suppression Attack in RPL

MATLAB and Python Interfacing Workflows

Version 15.0

A Network Simulation & Emulation Software

By

TETCOS LLP

Contents

Software: NetSim Standard v15.0 (64 bit), Visual Studio 2022, MATLAB R2019 or later, Python 3.13, matplotlib.

MATLAB project download link:
https://github.com/NetSim-TETCOS/DIO-Suppression-Attack-MATLAB-v15.0/archive/refs/heads/main.zip

Python project download link:
https://github.com/NetSim-TETCOS/DIO-Suppression-Attack-Python-v15.0/archive/refs/heads/main.zip

NetSim file exchange setup notes:
https://support.tetcos.com/en/support/solutions/articles/14000128666-downloading-and-setting-up-netsim-file-exchange-projects

1 Introduction

RPL uses DODAG Information Object (DIO) messages during DODAG formation. Nodes listen to DIO messages, refresh their parent and sibling information, compute rank, and advertise route information through DAO messages.

In a DIO suppression attack, a malicious node repeats DIO messages toward its neighbors. When legitimate nodes receive enough DIO messages that they treat as consistent, they suppress their own DIO transmissions in the Trickle interval. That suppression can keep some nodes undiscovered and can prevent routes from forming.

The attack degrades an IoT network because DIO messages carry the topology information needed by RPL. A hidden node may fail to join the DODAG or may lack a route to the destination, so application packets generated by that node do not reach the sink.

The v15.0 project includes MATLAB and Python interfacing for plotting the DODAG during simulation. NetSim sends the RPL parent-child graph, node positions, ranks, simulation time, and malicious-node ID to MATLAB or Python. The client refreshes the DODAG plot and sends a status value back to NetSim.

2 DIO Suppression Attack Overview

- A malicious node executes the attack by repeatedly broadcasting DIO messages to neighboring RPL nodes.
- Legitimate nodes that receive enough consistent DIO messages suppress their own DIO transmissions in the Trickle interval.
- Suppressed DIO messages disrupt neighbor discovery and DODAG topology discovery.
- Hidden nodes and undiscovered routes reduce packet delivery in the RPL network.

2.1 Role of NetSim

NetSim is used to model the smart agriculture IoT network with wireless sensors, a 6LoWPAN gateway, a router, and a wired node. During simulation, the RPL implementation forms the DODAG, computes ranks and route entries, and records the application metrics. The MATLAB

workspace sends DODAG state to `PlotDAG.m`. The Python workspace sends the same DODAG state to `dodag_graph_client.py` through the NetSim Python socket interface.

3 Scenario Overview

The v15.0 workspace contains three saved experiments:

Table 3-1: *Saved v15.0 experiments used in this report.*

Experiment	Attack state	DIO k
DIO_Redundancy-constant-6	Enabled	6
DIO_Redundancy-constant-6 without attack	Disabled	6
DIO-Redundancy-constant-7	Enabled	7

The network has eight wireless sensors, one 6LoWPAN gateway, one router, and one wired node. `WIRELESS_SENSOR_11` is configured as the malicious node. Two sensor applications send traffic to `WIRED_NODE_10`.

In the network screenshot, the malicious wireless sensor is shown in red. The applications and wireless link settings match the earlier v14.3 report: Application 1 runs from Device 1 to Device 10, and Application 2 runs from Device 4 to Device 10. The packet size is 50 bytes and the inter-arrival time is 1,000,000 microseconds for both applications.

Table 3-2: *Application and wireless link settings.*

Item	Parameter	Value
Application 1	Source / destination	Device 1 / Device 10
Application 1	Packet size	50 bytes
Application 1	Inter-arrival time	1,000,000 microseconds
Application 2	Source / destination	Device 4 / Device 10
Application 2	Packet size	50 bytes
Application 2	Inter-arrival time	1,000,000 microseconds
Wireless link	Channel characteristics	Pathloss only
Wireless link	Pathloss model	Log distance
Wireless link	Pathloss exponent	2.5

3.1 Scenario GUI Properties

The saved scenario uses the following GUI settings.

1. Open the application properties and set:
   - Application 1: Source `DEVICE ID 1`, destination `DEVICE ID 10`, packet size 50 bytes, inter-arrival time 1,000,000 microseconds.
   - Application 2: Source `DEVICE ID 4`, destination `DEVICE ID 10`, packet size 50 bytes, inter-arrival time 1,000,000 microseconds.

2. Open the wireless link properties and set:
   - Channel characteristics: Pathloss only.
   - Pathloss model: Log distance.
   - Pathloss exponent: 2.5.
3. Open **LOWPAN Gateway Properties**, select **Network Layer**, select **RPL**, and set **DIO Redundancy Constant**.
4. Set the DIO redundancy constant to `6` for the k=6 attack and no-attack cases.
5. Set the DIO redundancy constant to `7` for the k=7 attack case.

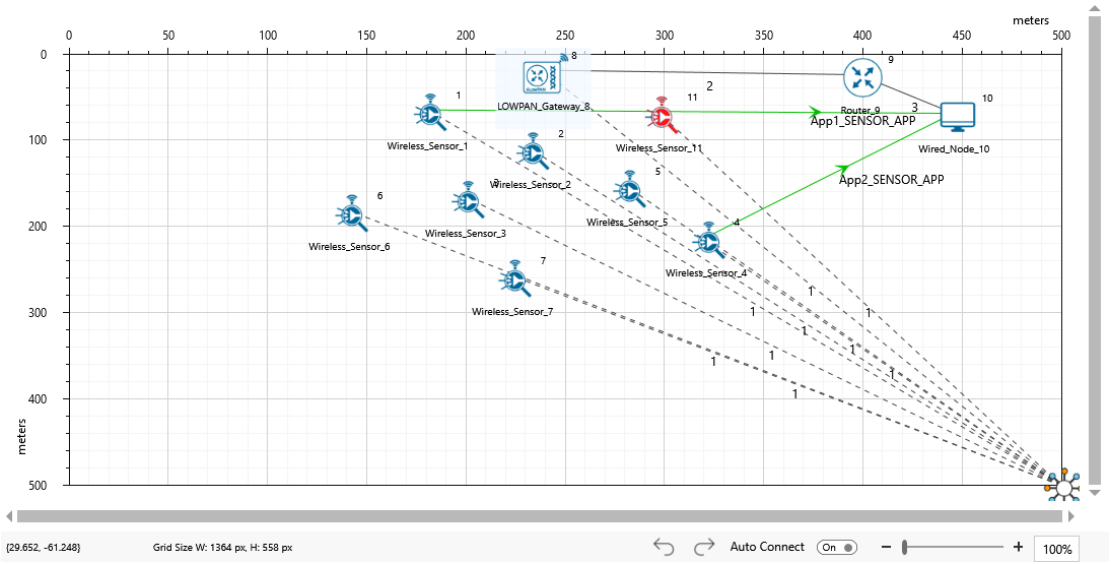

Figure 3-1: *DIO suppression attack scenario in the v15.0 workspace.*

4 RPL Attack Implementation

The attack implementation is in the RPL project under:

```
<workspace>\src\Simulation\RPL
```

The key RPL source files are `RPL.h`, `DIO.c`, `RPL_Message.c`, and `DoDAG_Plot.c`.

4.1 Attack Configuration

`RPL.h` sets the malicious node and enables or disables the attack:

```
#define MALICIOUS_NODE 11
#define DIO_Attack_Enable 1 // set 1 to enable, 0 to disable
```

The DIO redundancy constant is configured from the 6LoWPAN gateway RPL properties. For the first attack run, the value is set to 6. For the third run, it is set to 7. The DIO suppression attack requires the adversary to transmit only `k` DIO messages during each Trickle period. When `k=6`, the malicious node replays the DIO message six times to neighboring nodes; when `k=7`, it replays the message seven times. After receiving enough consistent DIO messages, neighboring nodes suppress their own DIO transmissions.

4.2 Implementation Flow

The RPL behavior follows this sequence:

- The transmitter broadcasts DIO messages during DODAG formation.
- A receiver refreshes its parent list, sibling list, and rank after receiving a DIO.
- The receiver sends a DAO message with route information.
- The malicious node, after receiving a DIO message, retransmits DIO messages repeatedly to legitimate nodes.
- Legitimate nodes that hear repeated consistent DIO messages suppress their own DIO transmissions.
- Continued suppression can leave some nodes hidden and some routes undiscovered.

5 RPL Attack Logic

`DIO.c` contains the functions added for malicious behavior:

- `fn_NetSim_RPL_MaliciousNode()` checks whether the current device is the malicious node.
- `Fn_NetSim_RPL_MaliciousNodeReplay()` retransmits the DIO message up to the configured DIO redundancy constant.

The DIO control message handling in `RPL_Message.c` calls the malicious replay function when the attack is enabled:

```
case DODAG_Information_Object:
#if DIO_Attack_Enable
  if (fn_NetSim_RPL_MaliciousNode(pstruEventDetails)) {
    rpl_process_dio_msg();
    Fn_NetSim_RPL_MaliciousNodeReplay(pstruEventDetails);
  }
  else {
    rpl_process_dio_msg();
  }
#else
  rpl_process_dio_msg();
#endif
  break;
```

6 RPL Log

The RPL log is written under each experiment folder:

```
<experiment-folder>\log\rpl.log
```

The `rpl.log` file records DODAG updates, node isolation events, DIO reception, and DIO replay activity from the malicious node. The same log output format is used for the MATLAB and Python workflows.

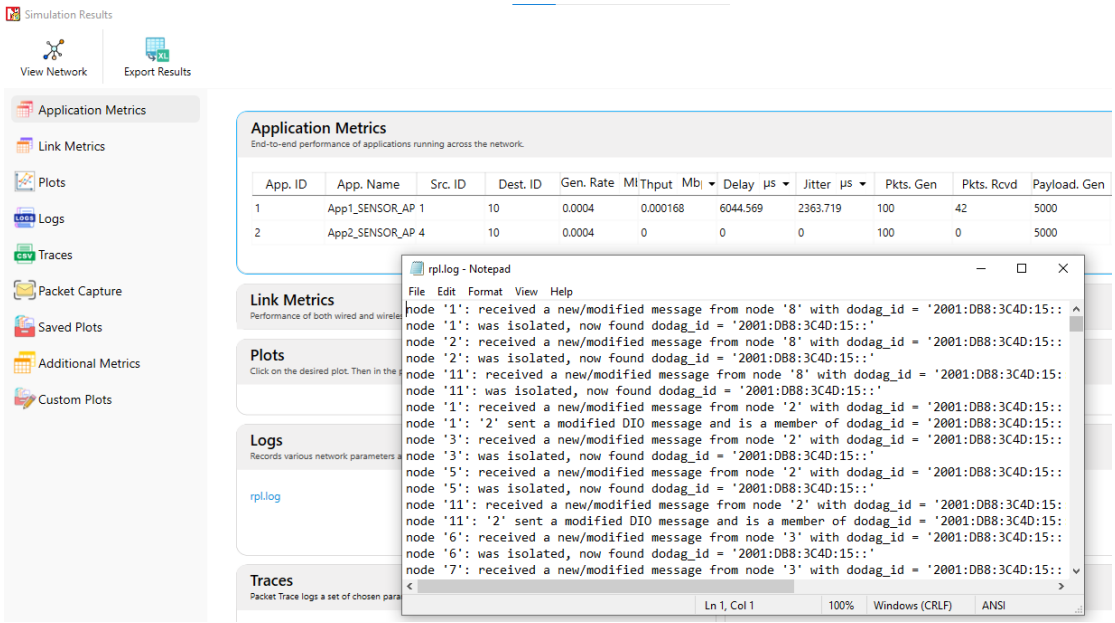

Figure 6-1: *RPL log generated during the DIO suppression attack run.*

7 MATLAB Workflow

The MATLAB workflow uses the same RPL attack logic and saved experiments as the Python workflow. The visualization path changes. NetSim starts the MATLAB interface and sends DODAG data to MATLAB. MATLAB then calls the plotting script.

7.1 MATLAB Script Folder

The MATLAB script is stored under the workspace binary folder:

```
<workspace>\bin_x64\MATLAB\PlotDAG.m
```

`PlotDAG.m` creates a graph from the adjacency matrix sent by NetSim. It labels each node with the NetSim device ID and rank, highlights the 6LoWPAN gateway in green, and highlights malicious node 11 in red.

7.2 Starting the MATLAB Interface

When the simulation starts, the RPL initialization function starts the MATLAB interface process and configures it with the NetSim application path:

```
void netsim_matlab_interface_start()
{
  char exeMATLAB[BUFSIZ];
  sprintf(exeMATLAB, "start \"\" \"%s%s%s\"",
          pszAppPath, pathSeperator, "MatlabInterface.exe");
  (void)system(exeMATLAB);
```

```
}

netsim_matlab_interface_start();
netsim_matlab_interface_configure(pszAppPath);
fn_netsim_matlab_DODAG_run();
```

This follows the same structure as the Dynamic Clustering MATLAB workflow: run the simulation, press any key in the simulation console, and let NetSim open the MATLAB interface console. MATLAB then plots the DODAG during runtime.

7.3 Data Sent to MATLAB

`DoDAG_Plot.c` builds MATLAB commands and sends them through the NetSim MATLAB API. The following values are passed to `PlotDAG.m`:

Table 7-1: *DODAG data sent from NetSim to MATLAB.*

MATLAB variable	Purpose
`theArray`	DODAG adjacency matrix built from preferred-parent links
`Xc`	Node X positions from the NetSim scenario
`Yc`	Node Y positions from the NetSim scenario
`H`	Index of the 6LoWPAN gateway for green highlighting
`nID`	NetSim configuration device IDs used in labels
`nRank`	RPL rank values used in node labels

The MATLAB call made by NetSim is:

```
output=PlotDAG(theArray,Xc,Yc,H,nID,nRank)
```

7.4 Running the MATLAB Workspace

1. Install 64-bit MATLAB and add the following folder to the Windows `PATH`:

   ```
   <MATLAB_INSTALL_DIRECTORY>\bin\win64
   ```

2. If MATLAB automation is not registered, open Command Prompt as administrator and run `matlab -regserver`.
3. Open the MATLAB workspace in NetSim.
4. Open the required saved experiment from **Your Work**.
5. Run the simulation and press any key in the NetSim console.
6. NetSim starts the MATLAB interface. MATLAB opens and refreshes the DODAG plot during runtime.

8 MATLAB Results

The results below were read from the MATLAB workspace `Metrics.xml` files and the RPL route tables stored in those files. The MATLAB DODAG plot receives the same parent-child state from NetSim during runtime; the route-table result explains why some nodes are absent from the DODAG in the attack cases.

8.1 Case 1: Attack Enabled, DIO Redundancy Constant 6

In this case, `WIRELESS_SENSOR_11` repeatedly transmits DIO messages. The RPL forwarding table shows that `WIRELESS_SENSOR_4` has no RPL route entries. Application 2 uses Device 4 as the source, so none of its generated packets reach Device 10.

When the 6LoWPAN gateway broadcasts DIO messages, nodes within communication range also broadcast their DIO messages. When the malicious node broadcasts a DIO, it repeats the same DIO toward neighboring nodes and prevents some neighboring DIO transmissions from being heard. As a result, routing information degrades and Wireless Sensor 4 is not part of the formed DODAG.

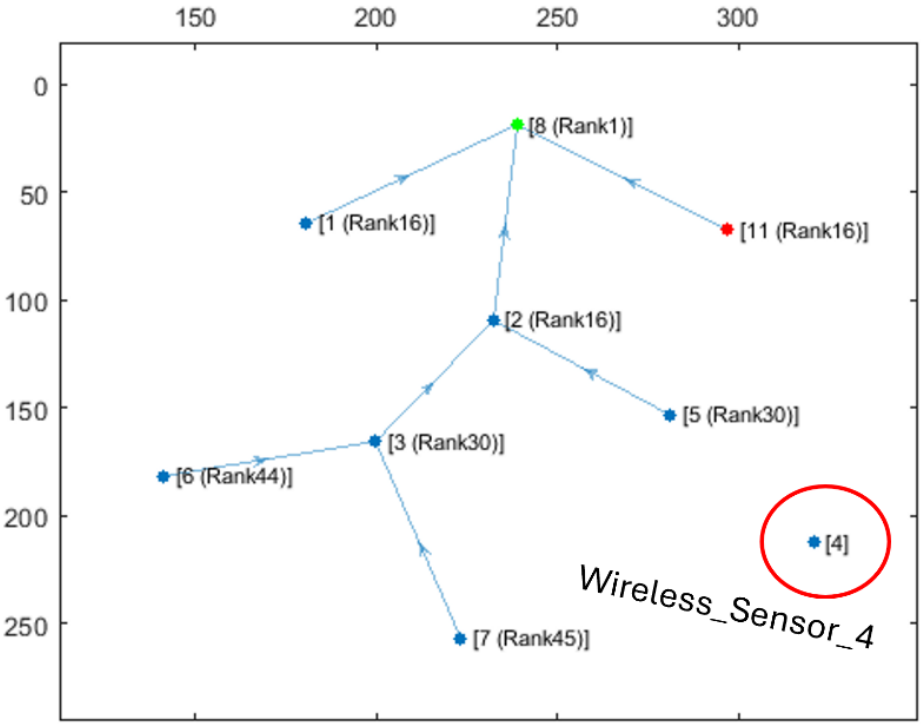

Figure 8-1: *MATLAB DODAG plot for attack enabled with DIO redundancy constant set to 6.*

Table 8-1: *MATLAB application metrics for attack enabled, DIO redundancy constant 6.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	42	0.000168	6044.569
App2_SENSOR_APP	100	0	0.000000	0.000

8.2 Case 2: Attack Disabled, DIO Redundancy Constant 6

With the attack disabled, every wireless sensor has RPL entries in the forwarding table. Application 1 and Application 2 both deliver packets to the wired node.

With the DIO suppression attack disabled, the DODAG forms with all wireless sensors participating. This improves network performance compared with the attack-enabled run because Device 4 can form an RPL route to the destination.

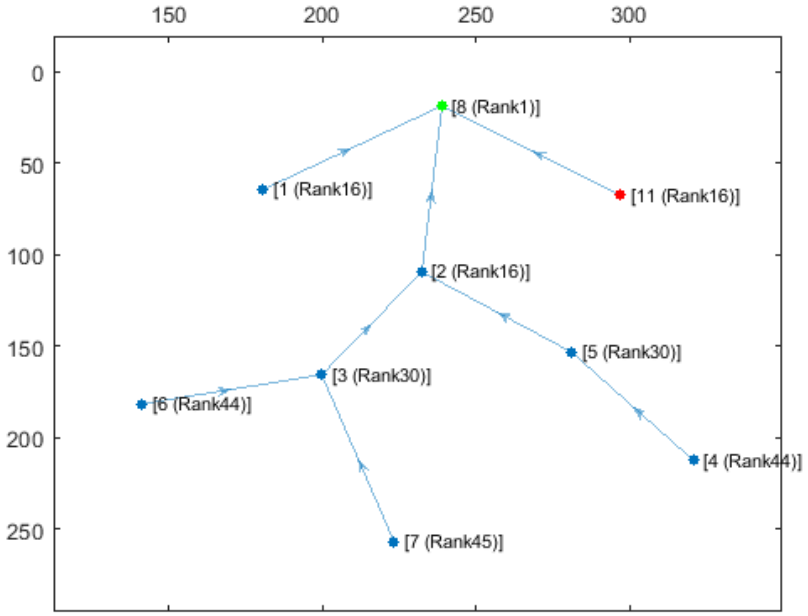

Figure 8-2: *MATLAB DODAG plot for attack disabled with DIO redundancy constant set to 6.*

Table 8-2: *MATLAB application metrics for attack disabled, DIO redundancy constant 6.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	84	0.000336	5254.146
App2_SENSOR_APP	100	53	0.000212	16862.350

8.3 Case 3: Attack Enabled, DIO Redundancy Constant 7

Increasing the DIO redundancy constant increases the number of repeated DIO transmissions by the malicious node. In this run, `WIRELESS_SENSOR_4` and `WIRELESS_SENSOR_5` have no RPL route entries. Application 2 again receives no packets at the destination, and Application 1 also receives fewer packets than in the k=6 attack run.

With the redundancy constant set to 7, the suppression effect is higher than in the k=6 case. Wireless Sensor 4 and Wireless Sensor 5 are absent from the DODAG route table. The simulation result shows lower throughput because the hidden nodes cannot forward application traffic through RPL.

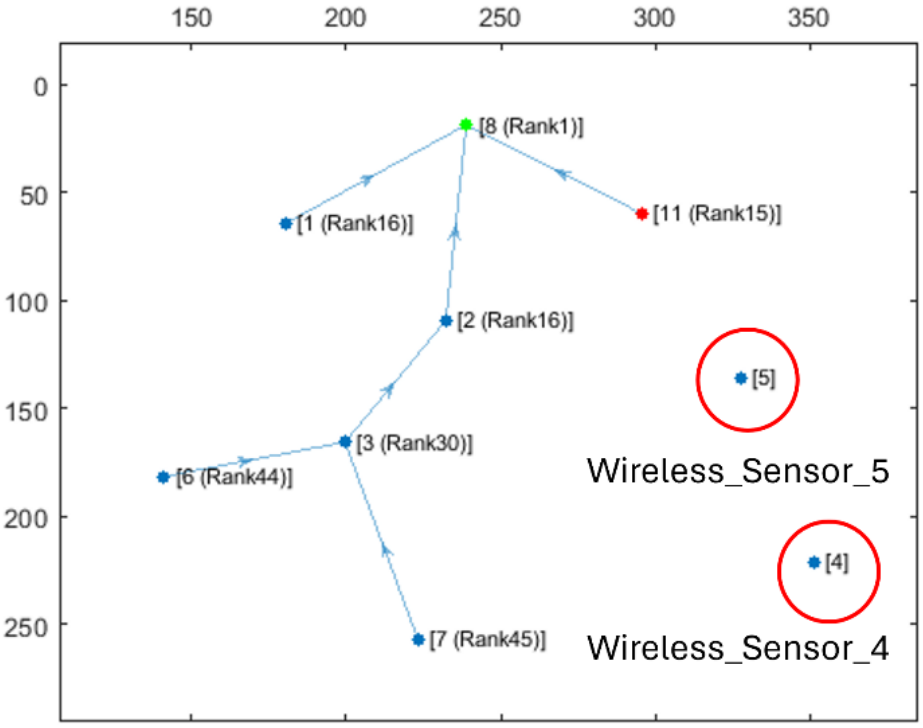

Figure 8-3: *MATLAB DODAG plot for attack enabled with DIO redundancy constant set to 7.*

Table 8-3: *MATLAB application metrics for attack enabled, DIO redundancy constant 7.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	33	0.000132	6388.435
App2_SENSOR_APP	100	0	0.000000	0.000

8.4 Comparison

Table 8-4: *MATLAB packet delivery and hidden-node summary.*

Case	Generated	Received	Nodes without RPL route entries
Attack enabled, k=6	200	42	Wireless Sensor 4
Attack disabled, k=6	200	137	None
Attack enabled, k=7	200	33	Wireless Sensor 4, Wireless Sensor 5

The no-attack case delivers 137 of 200 generated packets. With the attack enabled and k=6, delivery falls to 42 packets. With k=7, delivery falls further to 33 packets. The route table explains the reduction: the source for Application 2, Device 4, has no RPL route entries in both attack runs.

DIO suppression can severely degrade a Low Power and Lossy Network because repeated malicious DIO messages cause victim nodes to suppress their own DIO transmissions. This reduces route quality and can lead to network partitions.

9 Python Workflow

9.1 Python Client Folder in the Workspace

The Python files are located in the workspace source tree:

```
<workspace>\src\Simulation\NetSim_Python_Interface_Client
```

The folder contains:

- `dodag_graph_client.py`: receives DODAG data from NetSim and plots the graph.
- `netsim_socket_client.pyd`: Python extension module that implements the NetSim socket message APIs.
- `sample.py`: API usage sample for the Python interface.

9.2 Python Client File

The DIO suppression attack project uses the following Python script for DODAG visualization:

```
<workspace>\src\Simulation\NetSim_Python_Interface_Client\dodag_graph_client.py
```

This file receives the current DODAG graph data from NetSim, plots the parent-child links, marks the sink node in green, marks the malicious node in red, and sends a status value back to the RPL project.

9.3 Starting the Python Interface

Start the NetSim simulation first. After the simulation console starts the Python socket interface, run the Python client so that it can connect to NetSim. In this workspace, the RPL project starts the client from `DoDAG_Plot.c` by passing the path to `dodag_graph_client.py`. This is the same start sequence used in the Dynamic Clustering Python workflow: NetSim opens the socket during runtime, and the Python file connects to that socket.

```
sprintf(pythonScript, "%s%s%s",
        pszAppPath,
        pathSeperator,
        "..\\src\\Simulation\\NetSim_Python_Interface_Client\\dodag_graph_client.py");
netsim_python_interface_start(pythonScript);
dodag_socket_handle = Init_netsim_python_Interface();
```

If the client is run manually instead of being opened by NetSim, start the simulation and then run:

```
cd <workspace>\src\Simulation\NetSim_Python_Interface_Client
py -3.13 dodag_graph_client.py
```


The interface uses TCP port `5555`, defined in `NetSim_Python_Interface.h`. The Python client connects to:

```
127.0.0.1:5555
```

9.4 Runtime Socket Log

When the simulation starts, the Python console prints connection and refresh messages similar to:

```
DODAG graph Python client started
Connected to NetSim on 127.0.0.1:5555
DODAG graph refreshed: time=<time> ms, nodes=<count>, edges=<count>
```

The log confirms that NetSim has started the Python client and is exchanging DODAG graph data during runtime.

9.5 Data Sent to Python

During simulation, `fn_netsim_python_DODAG_run()` builds and sends a message containing:

Table 9-1: *DODAG data sent from NetSim to Python.*

Message segment	Purpose
Node count	Number of DODAG devices in the plot
Adjacency matrix	Parent-child links formed from preferred parent entries
X coordinates	Node X positions from the NetSim scenario
Y coordinates	Node Y positions from the NetSim scenario
Sink index	One-based index of the 6LoWPAN gateway
Node IDs	NetSim configuration device IDs used as graph labels
Ranks	Current RPL rank for each node
Simulation time	Current NetSim event time
Malicious node ID	Node highlighted in red by Python

`dodag_graph_client.py` reads these segments, builds graph edges from the adjacency matrix, plots the current DODAG using matplotlib, and returns a double status value to NetSim. The sink is plotted in green, the malicious node is plotted in red, and other sensors are plotted in blue.

If a GUI matplotlib backend is present, the plot window refreshes during simulation. If not, the script switches to PNG output and saves:

```
DODAG_Graph_Python.png
```

If plotting fails, the script writes the same graph data to:

```
DODAG_Graph_Python.csv
```

To run the no-attack case, set `DIO_Attack_Enable` to `0`, rebuild the RPL project, and run the no-attack experiment.

10 Running the Experiments

Run NetSim in administrator mode and open the saved experiments from **Your Work**. The GUI run sequence is:

1. Open the DIO suppression attack workspace in NetSim.
2. From **Your Work**, open the saved experiment that must be tested.
3. For the attack cases, keep `DIO_Attack_Enable` set to `1` in `RPL.h`.
4. For the no-attack case, set `DIO_Attack_Enable` to `0`, rebuild the RPL project, and run the no-attack experiment.
5. In the 6LoWPAN gateway properties, select **Network Layer**, select **RPL**, and set **DIO Redundancy Constant** to 6 or 7 as required.
6. Run the simulation and press any key in the simulation console when prompted.
7. Once the simulation starts the Python socket interface, the DODAG plot is refreshed by `dodag_graph_client.py` during runtime.

The same experiments can also be run from the command line using the workspace binary:

```
<workspace>\bin_x64\NetSimCore.exe ^
  -iopath "<workspace>\<experiment-folder>" ^
  -license 5053@192.168.0.4
```

For example, to run the attack case with DIO redundancy constant set to 6:

```
C:\Users\raven\Documents\NetSim\Workspaces\DIO-Suppression-Attack-v15.0\bin_x64\
    NetSimCore.exe ^
  -iopath "C:\Users\raven\Documents\NetSim\Workspaces\DIO-Suppression-Attack-v15.0\
    DIO_Redundancy-constant-6" ^
  -license 5053@192.168.0.4
```

11 Python Results

The results below were read from the Python workspace `Metrics.xml` files and the RPL route tables stored in those files. The Python DODAG plot receives the same parent-child state from NetSim during runtime; the route-table result explains why some nodes are absent from the DODAG in the attack cases.

11.1 Python DODAG Plots

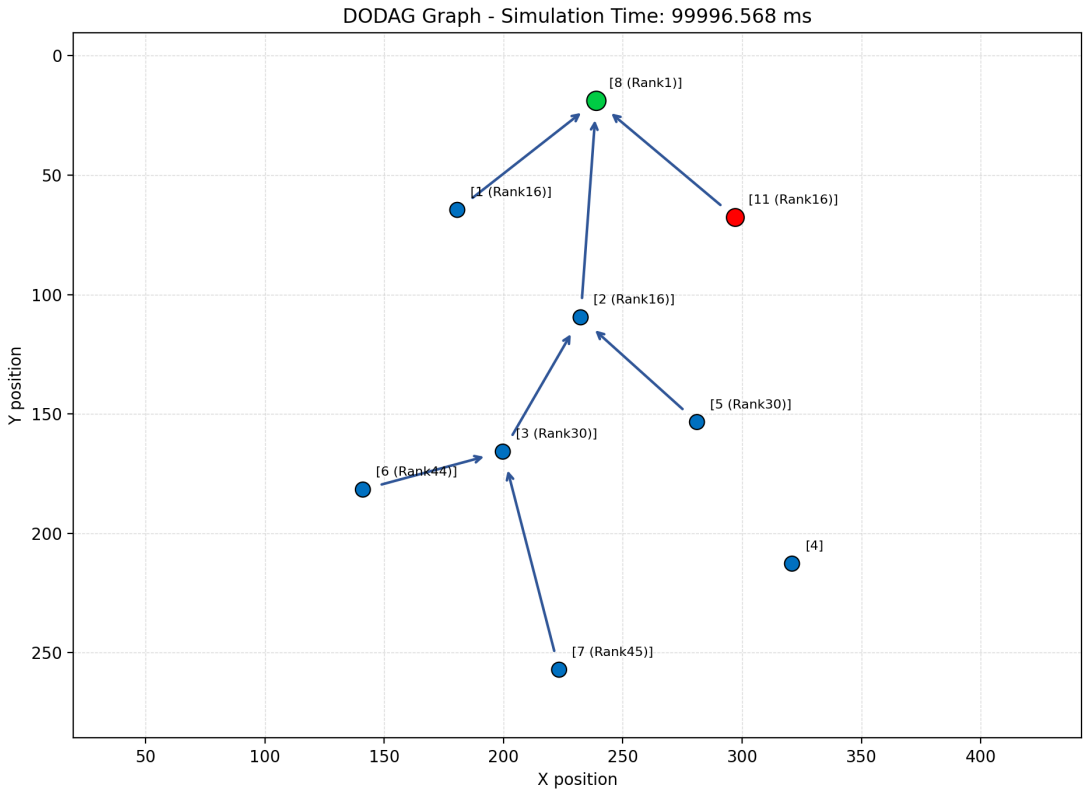

Figure 11-1: *Python DODAG plot for attack enabled with DIO redundancy constant set to 6.*

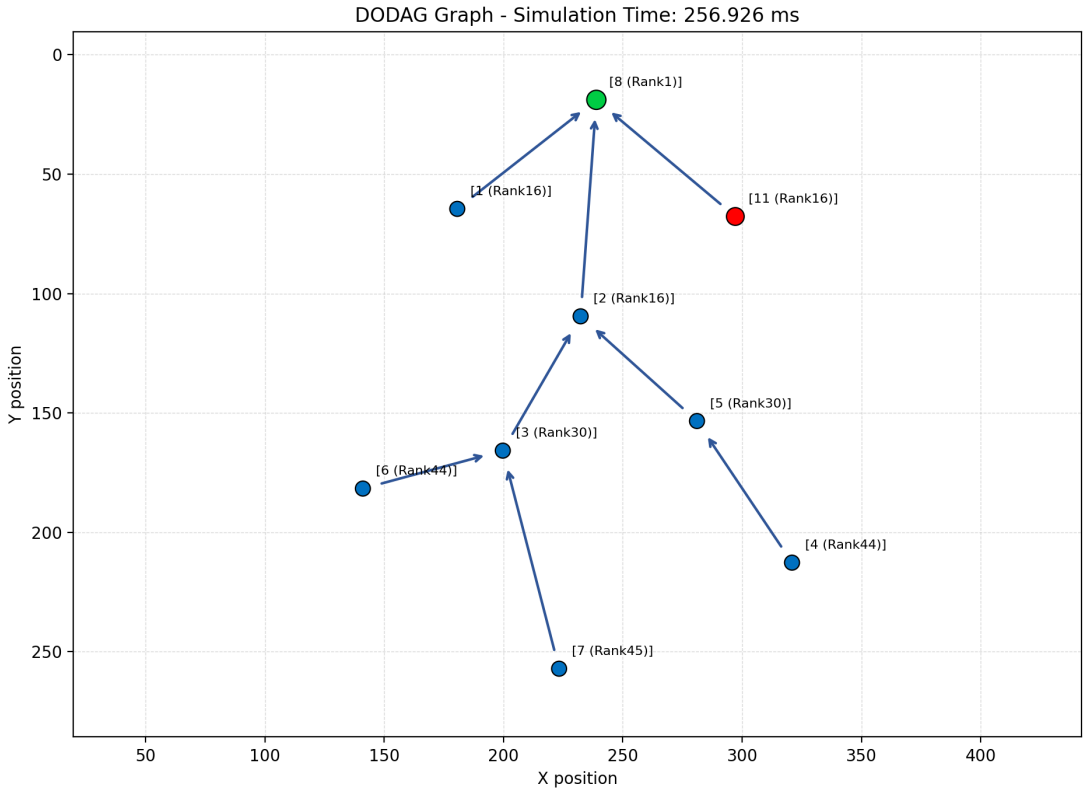

Figure 11-2: *Python DODAG plot for attack disabled with DIO redundancy constant set to 6.*

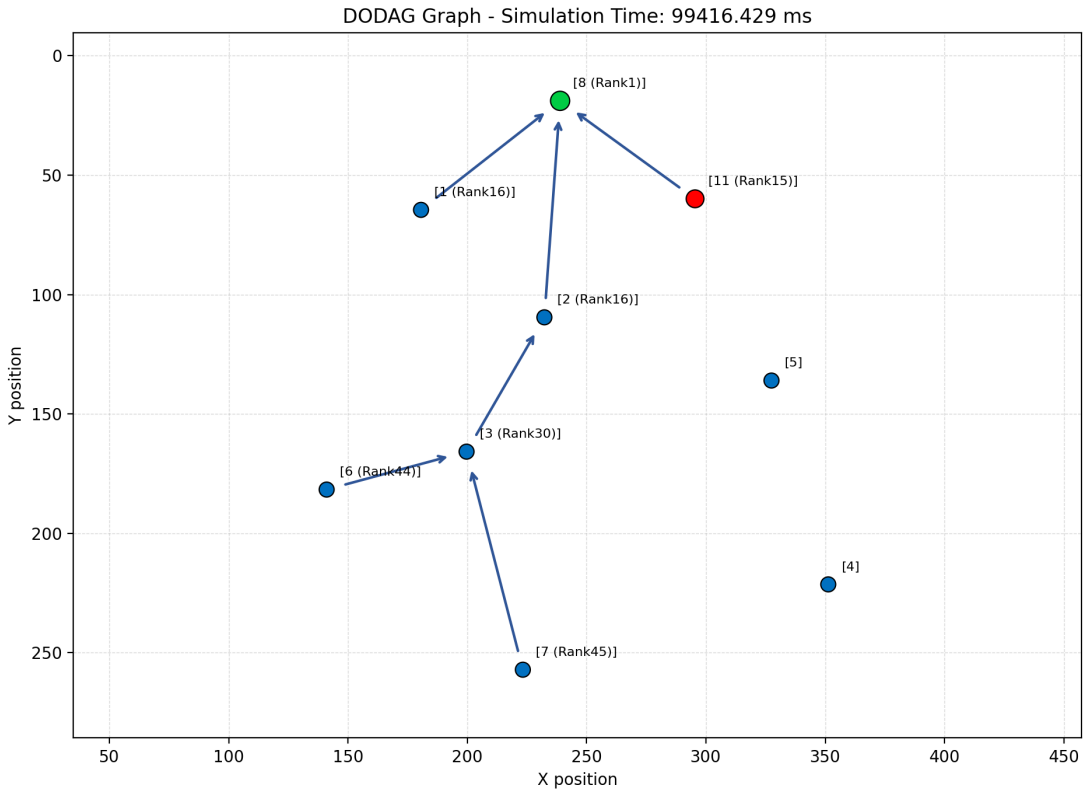

Figure 11-3: *Python DODAG plot for attack enabled with DIO redundancy constant set to 7.*

11.2 Case 1: Attack Enabled, DIO Redundancy Constant 6

In this case, `WIRELESS_SENSOR_11` repeatedly transmits DIO messages. The RPL forwarding table shows that `WIRELESS_SENSOR_4` has no RPL route entries. Application 2 uses Device 4 as the source, so none of its generated packets reach Device 10.

When the 6LoWPAN gateway broadcasts DIO messages, nodes within communication range also broadcast their DIO messages. When the malicious node broadcasts a DIO, it repeats the same DIO toward neighboring nodes and prevents some neighboring DIO transmissions from being heard. As a result, routing information degrades and Wireless Sensor 4 is not part of the formed DODAG.

Table 11-1: *Application metrics for attack enabled, DIO redundancy constant 6.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	42	0.000168	6044.569
App2_SENSOR_APP	100	0	0.000000	0.000

11.3 Case 2: Attack Disabled, DIO Redundancy Constant 6

With the attack disabled, every wireless sensor has RPL entries in the forwarding table. Application 1 and Application 2 both deliver packets to the wired node.

With the DIO suppression attack disabled, the DODAG forms with all wireless sensors participating. This improves network performance compared with the attack-enabled run because Device 4 can form an RPL route to the destination.

Table 11-2: *Application metrics for attack disabled, DIO redundancy constant 6.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	84	0.000336	5254.146
App2_SENSOR_APP	100	53	0.000212	16862.350

11.4 Case 3: Attack Enabled, DIO Redundancy Constant 7

Increasing the DIO redundancy constant increases the number of repeated DIO transmissions by the malicious node. In this run, `WIRELESS_SENSOR_4` and `WIRELESS_SENSOR_5` have no RPL route entries. Application 2 again receives no packets at the destination, and Application 1 also receives fewer packets than in the k=6 attack run.

With the redundancy constant set to 7, the suppression effect is higher than in the k=6 case. Wireless Sensor 4 and Wireless Sensor 5 are absent from the DODAG route table. The simulation result shows lower throughput because the hidden nodes cannot forward application traffic through RPL.

Table 11-3: *Application metrics for attack enabled, DIO redundancy constant 7.*

Application	Generated	Received	Throughput (Mbps)	Delay (us)
App1_SENSOR_APP	100	33	0.000132	6388.435
App2_SENSOR_APP	100	0	0.000000	0.000

11.5 Comparison

Table 11-4: *Packet delivery and hidden-node summary.*

Case	Generated	Received	Nodes without RPL route entries
Attack enabled, k=6	200	42	Wireless Sensor 4
Attack disabled, k=6	200	137	None
Attack enabled, k=7	200	33	Wireless Sensor 4, Wireless Sensor 5

The no-attack case delivers 137 of 200 generated packets. With the attack enabled and k=6, delivery falls to 42 packets. With k=7, delivery falls further to 33 packets. The route table explains the reduction: the source for Application 2, Device 4, has no RPL route entries in both attack runs.

DIO suppression can severely degrade a Low Power and Lossy Network because repeated malicious DIO messages cause victim nodes to suppress their own DIO transmissions. This reduces route quality and can lead to network partitions.

12 Conclusion

The v15.0 workspaces reproduce the DIO suppression behavior using MATLAB and Python interfacing for DODAG visualization. When the attack is enabled, repeated DIO messages from the malicious node suppress DIO transmissions from legitimate nodes. That changes DODAG formation and leaves some sensors without RPL routes.

Disabling the attack allows all wireless sensors to join the DODAG and improves delivery for both applications. Raising the DIO redundancy constant from 6 to 7 makes the attack more severe in this scenario, reducing total received packets from 42 to 33 and causing two sensors to lack RPL route entries.

13 Appendix: Python Setup

Install Python 3.13 and verify that it is available through the Python launcher. Install matplotlib for Python 3.13 if it is not already present:

```
py -3.13 -m pip install matplotlib
```

The Python client imports the NetSim socket extension from the same folder:

```
<workspace>\src\Simulation\NetSim_Python_Interface_Client\netsim_socket_client.pyd
```

Do not move `dodag_graph_client.py` away from this folder unless the path in `DoDAG_Plot.c` is changed. NetSim starts the script using a path relative to the application path:

```
..\src\Simulation\NetSim_Python_Interface_Client\dodag_graph_client.py
```

After changing `RPL.h` or `DoDAG_Plot.c`, rebuild the RPL project in Visual Studio. The build replaces the RPL DLL in the workspace `bin_x64` folder.

14 Appendix: Source Code Modification Steps

The source-code workflow is applicable to both v15.0 workspaces. The MATLAB workspace uses `PlotDAG.m`; the Python workspace uses `dodag_graph_client.py`.

1. Open NetSim and go to **Your Work**.
2. Select the DIO suppression attack workspace and open the source code in Visual Studio.
3. In the RPL project, open `RPL.h`.
4. Set the malicious node ID:

   ```
   #define MALICIOUS_NODE 11
   ```

5. Enable or disable the attack:

   ```
   #define DIO_Attack_Enable 1
   ```

   Set this value to `0` for the no-attack experiment.
6. Check that `RPL_Message.c` calls the malicious replay function for DIO control messages when the attack is enabled.
7. Rebuild the RPL project from Solution Explorer. The rebuilt RPL DLL is copied to the workspace `bin_x64` folder.
8. Run the required saved experiment from NetSim or from the command line.

The DIO control-message block used by the attack is:

```
case DODAG_Information_Object:
#if DIO_Attack_Enable
  if (fn_NetSim_RPL_MaliciousNode(pstruEventDetails)) {
    rpl_process_dio_msg();
    Fn_NetSim_RPL_MaliciousNodeReplay(pstruEventDetails);
  }
  else {
    rpl_process_dio_msg();
  }
#else
  rpl_process_dio_msg();
#endif
  break;
```

The Python DODAG plot script used after the simulation starts is:

```
<workspace>\src\Simulation\NetSim_Python_Interface_Client\dodag_graph_client.py
```