National Institute of Technology Karnataka, Surathkal

CS366 - Internet of Things

Design and Evaluation of RPL DIO Replay Attacks in ns-3

Kumaara Ganapathi (221CS153)
Shubhang Walavalkar (221CS248)
Sunil Thunga (221CS252)

November 11, 2025

Contents

0.1 Abstract

The Routing Protocol for Low-Power and Lossy Networks (RPL) underpins many IoT deployments but remains vulnerable to control-plane abuse such as DIO (DODAG Information Object) replay. This work presents a minimal, simulation-oriented study of DIO replay effects and a lightweight duplicate-filter countermeasure, implemented in ns-3 using two scratch programs.

In the baseline program (‘rpl-dio-replay-sim-without-attacker.cc‘), we model normal operation by scheduling only unique, monotonically increasing DIO sequence numbers to two receiver applications. Each receiver (‘DioReceiverApp‘) records seen sequence numbers in a small in-memory map and logs accepted messages; because no duplicates are injected, all scheduled DIOs are accepted.

In the attack program (‘rpl-dio-replay-sim.cc‘), we introduce a simple replay source (‘ReplayAttackerApp‘) and explicitly schedule duplicate deliveries of the *same* sequence number to the receivers to emulate a replay. The mitigation is intentionally simple and stateful: when enabled, ‘DioReceiverApp‘ drops any DIO whose sequence number has already been observed by that receiver and logs the drop event. The model uses a 3-node setup (attacker + two receivers), relies on ns-3’s event scheduler rather than real PHY/MAC traffic, and does not employ cryptography, neighbor identities, timestamps, or probabilistic scoring—only per-receiver sequence-number caching.

Results are illustrative: with mitigation disabled, receivers accept both the original and replayed DIOs; with mitigation enabled, receivers accept the first arrival of a given sequence and deterministically drop subsequent replays of that same sequence. While this toy model does not simulate full RPL control-plane behavior or realistic wireless effects, it clearly demonstrates the core intuition behind duplicate-based replay filtering. We discuss limitations (no per-neighbor state, no blacklist, no timing windows, and no authenticity guarantees) and outline straightforward extensions (real packet I/O, sliding windows, per-neighbor caches, and authenticated freshness) to evolve the prototype into a more faithful and robust evaluation.

0.2 Problem Statement

This project directly addresses "Problem Statement 1" from the provided security analysis document:

> “Develop and implement a mitigation strategy specifically targeting DIO replay attacks within RPL for static (non-mobile) network scenarios. Your solution should be lightweight and compatible with constrained IoT devices. Experimentally assess the effectiveness of your mitigation in maintaining correct DODAG formation and minimizing control message overhead, compared to an unprotected static baseline.”

0.3 Issues Identified

Based on the observed behavior of the replay simulation (rpl-dio-replay-sim.cc), several critical issues emerge when the attacker repeatedly re-injects previously transmitted DIO (DODAG Information Object) messages. Although the simulation models this behavior at an abstract level using scheduled events rather than full packet routing, the resulting patterns illustrate key violations of the CIA Triad—Confidentiality, Integrity, and Availability—within RPL-based IoT systems.

- **Violation of Availability:** The replay attacker continuously injects duplicate DIO messages at high frequency, forcing receivers to process the same control information multiple times. In a real RPL network, such "copycat" or DIO flooding behavior would overload the control plane, leading to excessive control overhead and increased energy consumption. Resource-constrained IoT nodes, which rely on limited battery power and small processing budgets, would waste cycles handling these redundant messages. The resulting congestion and energy drain effectively reduce the availability of legitimate routing services and degrade network responsiveness.

- **Violation of Integrity:**By replaying legitimate—but outdated—DIO messages, the attacker manipulates the perceived topology state of neighboring nodes. In a real deployment, such "neighbor confusion" could cause nodes to maintain stale or misleading routing information, potentially forming loops or selecting sub-optimal parents. This undermines the integrity of routing decisions, leading to degraded network metrics such as increased end-to-end delay, reduced packet delivery ratio (PDR), and unstable DODAG structure. Even in this simplified simulation, the replays clearly demonstrate how repeated control messages could mislead nodes if unmitigated.

- **Critical Impact Level:** While the replayed messages in our simulation are harmless log events, their real-world analog would impose a critical impact on an RPL network. Once nodes begin accepting and acting on duplicated control traffic, the effects cascade—nodes waste resources, valid updates are delayed, and certain links may become effectively "suppressed" by trickle timer misuse. This transforms the replay attack from a moderate nuisance into a severe routing disruption, capable of partitioning the network or isolating nodes entirely. The issue underscores the importance of per-node replay detection and stateful validation, even when cryptographic protection is not feasible on constrained IoT hardware.

0.4 Proposed Solution

To counteract the DIO replay vulnerability demonstrated in our simulation, we designed a lightweight, node-level defense mechanism implemented within the `DioReceiverApp` class in the `rpl-dio-replay-sim.cc` program. This approach embodies the principle of local detection and reaction, where each node independently validates the freshness of control messages before processing them. The technique is deliberately simple and computationally efficient, making it suitable for constrained IoT devices.

The proposed mitigation strategy operates entirely at the receiver node and focuses on identifying duplicate DIO sequence numbers, which are the hallmark of replay behavior. The core logic, implemented in the `ReceiveFakeDio()` method, proceeds as follows:

1. **State Initialization:** Each receiver maintains an in-memory cache, represented by the `m_seenSeq` map. This cache records all sequence numbers of DIO messages that the node has previously processed during the simulation.
2. **Duplicate Sequence Detection:** When a DIO is received, the receiver checks if its sequence number already exists in `m_seenSeq`.
   - If the sequence number is *new*, the DIO is accepted as legitimate and logged as a valid control message.
   - If the sequence number has already been observed, the DIO is flagged as a **potential replay** and immediately dropped.

   This process effectively filters repeated control messages generated by the replay attacker.
3. **Mitigation Control:** The mitigation behavior can be toggled using a command-line flag (`-mitigation=true/false`). When mitigation is disabled, the receiver accepts all DIOs regardless of duplication, simulating a baseline scenario with no protection.
4. **Logging and Observation:** The system employs `NS_LOG_INFO()` statements to record both accepted and dropped DIOs. This allows for clear visual verification of the mitigation's behavior during simulation runs, differentiating between "normal" and "attacked" conditions.
5. **Lightweight Design:** Unlike cryptographic defenses, this mitigation introduces virtually no computational or memory overhead. It relies solely on simple lookups in a small in-memory structure. This ensures feasibility for real-world, low-power IoT nodes.

Although simplified, this model successfully demonstrates the essential principle behind replay defense: each node locally remembers recent control message identifiers and suppresses any repeated arrivals. In full RPL deployments, this concept can be extended to use per-neighbor sequence tracking, timestamp validation, or authenticated nonces to improve resilience against forged or time-shifted replays.

—

0.4.1 Advantages

The proposed mitigation mechanism implemented in the `DioReceiverApp` class provides several practical advantages that make it suitable for constrained IoT environments. Although simplified compared to full-scale RPL security extensions, the approach effectively demonstrates how lightweight replay prevention can enhance network robustness without requiring heavy cryptographic operations.

- **Lightweight and Low-Overhead:** The solution completely avoids cryptography and complex packet authentication. Its logic depends solely on maintaining a small, in-memory cache (`m_seenSeq`) of previously received DIO sequence numbers. Each check involves a constant-time lookup and insertion in this structure, imposing negligible processing and memory cost—ideal for low-power IoT nodes.
- **Fast and Deterministic Detection:** Since each DIO is verified against a local record of seen sequence numbers, replayed messages are identified and discarded immediately upon arrival. This ensures instant detection of duplicate control messages, preventing unnecessary processing or routing-table changes.
- **Zero False Positives in Controlled Conditions:** Because the simulation uses strictly unique legitimate sequence numbers, the mechanism achieves perfect accuracy—every duplicate corresponds to a malicious replay. In real-world deployments, the same principle could be extended with sequence aging or per-neighbor context to tolerate legitimate retransmissions without penalizing benign traffic.
- **Effective Resource Protection:** By dropping repeated DIOs early in the reception pipeline, the mitigation conserves both processing cycles and battery power. This preserves the node's availability and prevents local flooding, directly addressing the most harmful consequence of DIO replay attacks.
- **Implementation Simplicity:** The defense integrates naturally within ns-3's `Application` model. It requires only a few lines of additional code and minimal state tracking per node, demonstrating how even a simple duplicate filter can meaningfully strengthen RPL's resilience against replay-based control-plane abuse.

0.5 Methodology

We use the ns-3 network simulator to model and evaluate the DIO replay attack and its mitigation. The methodology directly corresponds to the experimental design implemented in the `main()` function of the two simulation files: `rpl-dio-replay-sim.cc` (attack scenario) and `rpl-dio-replay-sim-without-attacker.cc` (baseline scenario).

0.5.1 Experimental Setup

The simulation environment is configured as follows:

- **Nodes and Topology:** Three nodes are created using the `ns3::NodeContainer`. Node 0 acts as the attacker, while Nodes 1 and 2 act as receivers. Although minimal, this configuration effectively demonstrates how replayed DIOs affect multiple receivers in a local broadcast domain.
- **Network Stack:** Each node is equipped with a standard `InternetStackHelper`, providing basic IPv6 support. Since the goal is to isolate the control-plane behavior rather than simulate physical-layer transmissions, no WiFi or mobility models are attached.
- **Applications:**
  - **Replay Attacker (Node 0):** Runs the `ReplayAttackerApp`, which repeatedly triggers fake DIO transmissions at fixed time intervals to emulate a replay attack.
  - **Receivers (Nodes 1 and 2):** Each node runs an instance of the `DioReceiverApp`, which maintains a local cache of received DIO sequence numbers and applies mitigation logic if enabled.

0.5.2 Attack Simulation

The replay behavior is modeled using scheduled DIO receptions:

1. **Initial Legitimate Messages:** At the start of the simulation, unique DIO messages with sequential identifiers (e.g., sequence number = 1) are delivered to both receivers, representing normal network traffic.
2. **Replayed Messages:** After a short delay, the attacker re-injects the same DIO sequence number, simulating the replay of a previously valid control packet. These replays are scheduled through `Simulator::Schedule()` calls.
3. **Event Logging:** All DIO receptions, whether accepted or dropped, are logged using `NS_LOG_INFO()` to enable detailed observation of the mitigation's effect.

0.5.3 Mitigation Mechanism Implementation

The replay defense mechanism is implemented within the `DioReceiverApp::ReceiveFakeDio()` method:

- **Baseline (Mitigation OFF):**
  - **Command:** `-mitigation=false`
  - **Logic:** The receiver accepts all DIOs without checking for duplicates, emulating an unprotected network.
- **Protected (Mitigation ON):**
  - **Command:** `-mitigation=true`
  - **Logic:** The receiver checks each incoming DIO's sequence number against the local cache `m_seenSeq`. If the number has already been observed, the message is flagged as a replay and dropped immediately.

0.5.4 Evaluation and Metrics

To evaluate the impact of the replay attack and the effectiveness of the mitigation, two distinct simulation runs are compared:

- **Baseline Run:** Mitigation disabled. All DIO messages, including replays, are accepted.
- **Protected Run:** Mitigation enabled. Duplicate DIOs are detected and dropped at the receiver.

The following qualitative metrics are derived from simulation logs:

- `Accepted DIOs:` Count of legitimate (unique) DIOs processed successfully.
- `Dropped DIOs:` Number of replayed DIOs detected and rejected by the mitigation.
- `Reaction Time:` The simulation timestamp at which the first replayed DIO is dropped.

Together, these metrics demonstrate that the simple sequence-number–based check provides immediate protection against redundant control-plane traffic, illustrating the core principle of lightweight replay mitigation in IoT routing protocols.

0.6 Code Implementation

The complete simulation is contained in the `rpl-dio-replay-sim-without-attacker.cc` and `rpl-dio-replay-sim.cc` scratch program. The full source code is provided below.

```
/* rpl-dio-replay-sim-without-attacker.cc
 * ----------------------------------------



#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/ipv6-address.h"
#include "ns3/ipv6-routing-helper.h"
#include "ns3/ipv6-static-routing-helper.h"
#include "ns3/ipv6-routing-table-entry.h"
#include "ns3/applications-module.h"
#include "ns3/udp-socket.h"
#include "ns3/udp-socket-factory.h"

#include <iostream>
#include <map>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("RplDioNoAttackSim");

// ------------------------------
// DioReceiverApp (Same as original, but mitigation is irrelevant
   without attack)
// ------------------------------
class DioReceiverApp : public Application {
public:
    DioReceiverApp() {}
    virtual ~DioReceiverApp() {}

    void Setup(Ptr<Node> node, bool mitigation) {
        m_node = node;
        m_mitigation = mitigation;
    }

    void ReceiveFakeDio(uint32_t seq) {
        double now = Simulator::Now().GetSeconds();

        // Mitigation check is included but will only drop if 'seq' is
   a duplicate.
        // In this 'No Attack' code, all scheduled 'seq' are unique, so
    nothing is dropped.
        if (m_mitigation) {
            // Drop replayed sequence numbers
            if (m_seenSeq.find(seq) != m_seenSeq.end()) {
                NS_LOG_INFO("Node " << m_node->GetId() << " DROPPED
   REPLAY DIO seq=" << seq << " at t=" << now);
                return;
            }
        }
```

```
        m_seenSeq[seq] = true;
        NS_LOG_INFO("Node " << m_node->GetId() << " accepted UNIQUE DIO
    seq=" << seq << " at t=" << now);
    }

private:
    Ptr<Node> m_node;
    bool m_mitigation;
    std::map<uint32_t, bool> m_seenSeq;
};

// ------------------------------
// Main (Attacker components removed)
// ------------------------------
int main(int argc, char *argv[]) {
    double simTime = 5.0; // Reduced time since fewer events are
   scheduled
    bool enableMitigation = false; // Mitigation flag remains, but is
   inactive in this scenario

    CommandLine cmd;
    cmd.AddValue("mitigation", "Enable DIO replay mitigation (inactive
   in this example)", enableMitigation);
    cmd.AddValue("simTime", "Simulation time in seconds", simTime);
    cmd.Parse(argc, argv);

    // Create nodes
    NodeContainer nodes;
    nodes.Create(3);

    // Install internet stack
    InternetStackHelper internet;
    internet.Install(nodes);

    // Create receiver applications
    // We can arbitrarily set mitigation to true or false; it won't
   drop unique messages.
    Ptr<DioReceiverApp> receiver1 = CreateObject<DioReceiverApp>();
    receiver1->Setup(nodes.Get(1), true); // Mitigation set to true for
    demo
    nodes.Get(1)->AddApplication(receiver1);
    receiver1->SetStartTime(Seconds(0.1));

    Ptr<DioReceiverApp> receiver2 = CreateObject<DioReceiverApp>();
    receiver2->Setup(nodes.Get(2), false); // Mitigation set to false
   for demo
    nodes.Get(2)->AddApplication(receiver2);
    receiver2->SetStartTime(Seconds(0.1));

    // --- Schedule ONLY UNIQUE DIOs (Simulating Normal Network Traffic
   ) ---
    // The sequence number (seq) increases to reflect a new, legitimate
    control message.

    // Receiver 1 accepts its first DIO
    Simulator::Schedule(Seconds(1.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 1);
```

```
    // Receiver 2 accepts its first DIO (same seq is fine, as it's a
   different receiver/DODAG structure)
    Simulator::Schedule(Seconds(1.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 1);

    // Receiver 1 accepts a NEW, UPDATED DIO (new sequence number)
    Simulator::Schedule(Seconds(2.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 2);

    // Receiver 2 accepts a NEW, UPDATED DIO (new sequence number)
    Simulator::Schedule(Seconds(2.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 2);

    // Receiver 1 accepts yet another NEW DIO
    Simulator::Schedule(Seconds(3.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 3);

    // Total unique accepted messages will be 5.
    //
   ----------------------------------------------------------------------

    Simulator::Stop(Seconds(simTime));
    Simulator::Run();
    Simulator::Destroy();

    return 0;
}



--------------------------------------------------------------------------------



/* rpl-dio-replay-sim.cc
 * ---------------------------------------
 * Wireless RPL-DIO Replay Attack Simulation
 * ---------------------------------------
 * - Root node sends DIO messages (deterministic or randomized)
 * - Attacker captures and replays them
 * - DRM component detects duplicates, increments suspicion, and
   blacklists
 * - Simulation uses WiFi ad-hoc network, so only nearby nodes receive
   replays
 *
 * Build: ./waf build
 * Run example (attack + mitigation):
 * ./waf --run "scratch/dio --deterministicRoot=true --
   randomizeAttacker=false --disableRootProtection=false --simTime=80
   --attackStart=12 --attackerRate=5"
 */


#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/ipv6-address.h"
```

```
#include "ns3/ipv6-routing-helper.h"
#include "ns3/ipv6-static-routing-helper.h"
#include "ns3/ipv6-routing-table-entry.h"
#include "ns3/applications-module.h"
#include "ns3/udp-socket.h"
#include "ns3/udp-socket-factory.h"

#include <iostream>
#include <map>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("RplDioReplaySim");

// ------------------------------
// ReplayAttackerApp
// ------------------------------
class ReplayAttackerApp : public Application {
public:
    ReplayAttackerApp() {}
    virtual ~ReplayAttackerApp() { Simulator::Cancel(m_event); }

    void Setup(Ptr<Node> node, double intervalSeconds, bool repeat) {
        m_node = node;
        m_interval = Seconds(intervalSeconds); // ns3::Time
        m_repeat = repeat;
    }

    void StartApplication() override {
        Replay();
    }

private:
    void Replay() {
        NS_LOG_INFO("Replay attacker sending fake DIO at " << Simulator
   ::Now().GetSeconds() << "s");
        // In real simulation: we would send fake DIO here

        if (m_repeat) {
            m_event = Simulator::Schedule(m_interval, &
   ReplayAttackerApp::Replay, this);
        }
    }

    Ptr<Node> m_node;
    ns3::Time m_interval;
    bool m_repeat;
    EventId m_event;
};

// ------------------------------
// DioReceiverApp
// ------------------------------
class DioReceiverApp : public Application {
public:
    DioReceiverApp() {}
    virtual ~DioReceiverApp() {}
```

```
    void Setup(Ptr<Node> node, bool mitigation) {
        m_node = node;
        m_mitigation = mitigation;
    }

    void ReceiveFakeDio(uint32_t seq) {
        double now = Simulator::Now().GetSeconds();
        if (m_mitigation) {
            // Drop replayed sequence numbers
            if (m_seenSeq.find(seq) != m_seenSeq.end()) {
                NS_LOG_INFO("Node " << m_node->GetId() << " DROPPED
   replayed DIO seq=" << seq << " at " << now);
                return;
            }
        }
        m_seenSeq[seq] = true;
        NS_LOG_INFO("Node " << m_node->GetId() << " accepted DIO seq="
   << seq << " at " << now);
    }

private:
    Ptr<Node> m_node;
    bool m_mitigation;
    std::map<uint32_t, bool> m_seenSeq;
};

// ------------------------------
// Main
// ------------------------------
int main(int argc, char *argv[]) {
    double simTime = 10.0;
    bool enableMitigation = false;

    CommandLine cmd;
    cmd.AddValue("mitigation", "Enable DIO replay mitigation",
   enableMitigation);
    cmd.AddValue("simTime", "Simulation time in seconds", simTime);
    cmd.Parse(argc, argv);

    // Create nodes
    NodeContainer nodes;
    nodes.Create(3);

    // Install internet stack
    InternetStackHelper internet;
    internet.Install(nodes);

    // Create applications
    Ptr<ReplayAttackerApp> attacker = CreateObject<ReplayAttackerApp>()
   ;
    attacker->Setup(nodes.Get(0), 1.0, true);
    nodes.Get(0)->AddApplication(attacker);
    attacker->SetStartTime(Seconds(0.1));

    Ptr<DioReceiverApp> receiver1 = CreateObject<DioReceiverApp>();
    receiver1->Setup(nodes.Get(1), enableMitigation);
    nodes.Get(1)->AddApplication(receiver1);
    receiver1->SetStartTime(Seconds(0.1));
```

```
    Ptr<DioReceiverApp> receiver2 = CreateObject<DioReceiverApp>();
    receiver2->Setup(nodes.Get(2), enableMitigation);
    nodes.Get(2)->AddApplication(receiver2);
    receiver2->SetStartTime(Seconds(0.1));

    // Simulate DIOs being sent by attacker every 1s
    Simulator::Schedule(Seconds(1.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 1);
    Simulator::Schedule(Seconds(2.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 1); // replay
    Simulator::Schedule(Seconds(1.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 1);
    Simulator::Schedule(Seconds(2.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 1); // replay

    Simulator::Stop(Seconds(simTime));
    Simulator::Run();
    Simulator::Destroy();

    return 0;
}
```

Listing 1: Full source code for dio.cc

0.7 Code Explanation

Both files are small ns-3 simulations (C++) that do *not* implement full RPL protocol stacks. Instead, they simulate the **control-message-level behavior** relevant to a DIO replay attack and a minimal mitigation strategy.

- The "without attacker" version demonstrates only unique DIO acceptance and optionally records seen sequence numbers.
- The "with attacker" version simulates a repeating attacker sending the same DIO sequence numbers and receivers that drop duplicates if mitigation is enabled.

0.7.1 Common includes and ns-3 basics

At the top of both programs you see includes such as:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
```

What these provide:

- `core-module`: ns-3 core types (Simulator, Time, EventId, CommandLine, NS_LOG).
- `network-module`: Node, NodeContainer, NetDevice, etc.
- `internet-module`: Internet stack helpers (InternetStackHelper) and protocol helpers — included here even when not strictly required by the minimal simulation.
- `applications-module`: Application base class and related helpers.

NS_LOG_COMPONENT_DEFINE Each file defines a logging component, e.g.

```
NS_LOG_COMPONENT_DEFINE("RplDioNoAttackSim");
```

This allows runtime control of debug/info logging from ns-3's logging subsystem.

0.7.2 File 1: `rpl-dio-replay-sim-without-attacker.cc`

This program models receivers accepting unique DIOs. It demonstrates the intended mitigation code path but runs with no actual attacker and only unique sequence numbers.

0.7.3 Class: DioReceiverApp

```
class DioReceiverApp : public Application {
public:
    DioReceiverApp() {}
    virtual ~DioReceiverApp() {}

    void Setup(Ptr<Node> node, bool mitigation) {
        m_node = node;
        m_mitigation = mitigation;
    }

    void ReceiveFakeDio(uint32_t seq) {
        double now = Simulator::Now().GetSeconds();

        if (m_mitigation) {
            if (m_seenSeq.find(seq) != m_seenSeq.end()) {
                NS_LOG_INFO("Node " << m_node->GetId() << " DROPPED
   REPLAY DIO seq=" << seq << " at t=" << now);
                return;
            }
        }

        m_seenSeq[seq] = true;
        NS_LOG_INFO("Node " << m_node->GetId() << " accepted UNIQUE DIO
    seq=" << seq << " at t=" << now);
    }

private:
    Ptr<Node> m_node;
    bool m_mitigation;
    std::map<uint32_t, bool> m_seenSeq;
};
```

Responsibilities

- Stores a pointer to the ns-3 `Node` that "hosts" the application (for logging IDs).
- Contains a boolean `m_mitigation` toggling duplicate-detection.
- `m_seenSeq` stores seen sequence numbers (map used as a set).
- `ReceiveFakeDio(seq)` performs the duplicate-check and logs acceptance/dropping.

0.7.4 main()

```
int main(int argc, char *argv[]) {
    double simTime = 5.0;
    bool enableMitigation = false;

    CommandLine cmd;
```

```
    cmd.AddValue("mitigation", "Enable DIO replay mitigation (inactive
   in this example)", enableMitigation);
    cmd.AddValue("simTime", "Simulation time in seconds", simTime);
    cmd.Parse(argc, argv);

    NodeContainer nodes;
    nodes.Create(3);

    InternetStackHelper internet;
    internet.Install(nodes);

    Ptr<DioReceiverApp> receiver1 = CreateObject<DioReceiverApp>();
    receiver1->Setup(nodes.Get(1), true);
    nodes.Get(1)->AddApplication(receiver1);
    receiver1->SetStartTime(Seconds(0.1));

    Ptr<DioReceiverApp> receiver2 = CreateObject<DioReceiverApp>();
    receiver2->Setup(nodes.Get(2), false);
    nodes.Get(2)->AddApplication(receiver2);
    receiver2->SetStartTime(Seconds(0.1));

    Simulator::Schedule(Seconds(1.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 1);
    Simulator::Schedule(Seconds(1.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 1);
    Simulator::Schedule(Seconds(2.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 2);
    Simulator::Schedule(Seconds(2.5), &DioReceiverApp::ReceiveFakeDio,
   receiver2, 2);
    Simulator::Schedule(Seconds(3.0), &DioReceiverApp::ReceiveFakeDio,
   receiver1, 3);

    Simulator::Stop(Seconds(simTime));
    Simulator::Run();
    Simulator::Destroy();

    return 0;
}
```

Explanation of key parts

- `CommandLine`: Allows running the program with flags (e.g., `-mitigation=true`).
- `NodeContainer nodes; nodes.Create(3);` creates 3 ns-3 nodes (IDs 0..2).
- `InternetStackHelper internet; internet.Install(nodes);` attaches the minimal internet stack to nodes, though this demo doesn't use sockets.
- Two `DioReceiverApp` instances are created and started at 0.1 s (just to simulate app lifecycle).
- `Simulator::Schedule` calls inject events at specific absolute times to call `ReceiveFakeDio` with given sequence numbers.
- The simulation stops after `simTime`.

What the demo shows

- Receiver 1 has mitigation enabled; receiver 2 has it disabled. However, because every scheduled seq is unique per receiver in this demo, no replays are dropped. The logs will show accepted unique DIOs.

0.7.5 File 2: `rpl-dio-replay-sim.cc` (with attacker)

This file is the more interesting one: it simulates an attacker and receivers with optional mitigation.

0.7.6 Class: `ReplayAttackerApp`

```
class ReplayAttackerApp : public Application {
public:
    ReplayAttackerApp() {}
    virtual ~ReplayAttackerApp() { Simulator::Cancel(m_event); }

    void Setup(Ptr<Node> node, double intervalSeconds, bool repeat) {
        m_node = node;
        m_interval = Seconds(intervalSeconds);
        m_repeat = repeat;
    }

    void StartApplication() override {
        Replay();
    }

private:
    void Replay() {
        NS_LOG_INFO("Replay attacker sending fake DIO at " << Simulator
   ::Now().GetSeconds() << "s");
        if (m_repeat) {
            m_event = Simulator::Schedule(m_interval, &
   ReplayAttackerApp::Replay, this);
        }
    }

    Ptr<Node> m_node;
    ns3::Time m_interval;
    bool m_repeat;
    EventId m_event;
};
```

Responsibilities

- Periodically triggers `Replay()` which logs a message and re-schedules itself when `m_repeat` is true.

- The destructor cancels the scheduled event to avoid dangling events after object destruction.

Important observation

- The current `Replay()` implementation only logs the attack attempt. It does not actually send packets through NS-3 sockets or NetDevices to other nodes. Instead, the demo directly schedules `DioReceiverApp::ReceiveFakeDio` calls to simulate the effect of a replay arriving at receivers (see main scheduling below).

0.7.7 Class: `DioReceiverApp` (same as earlier)

The same receiver class (with `ReceiveFakeDio` and `m_seenSeq`) appears here; in this file it is used to illustrate acceptance vs. dropping when a duplicate seq is scheduled.

0.7.8 main()

```
int main(int argc, char *argv[]) {
    double simTime = 10.0;
    bool enableMitigation = false;

    CommandLine cmd;
    cmd.AddValue("mitigation", "Enable DIO replay mitigation",
   enableMitigation);
    cmd.AddValue("simTime", "Simulation time in seconds", simTime);
    cmd.Parse(argc, argv);

    NodeContainer nodes;
    nodes.Create(3);

    InternetStackHelper internet;
    internet.Install(nodes);

    Ptr<ReplayAttackerApp> attacker = CreateObject<ReplayAttackerApp>()
   ;
    attacker->Setup(nodes.Get(0), 1.0, true);
    nodes.Get(0)->AddApplication(attacker);
    attacker->SetStartTime(Seconds(0.1));

    Ptr<DioReceiverApp> receiver1 = CreateObject<DioReceiverApp>();
    receiver1->Setup(nodes.Get(1), enableMitigation);
    nodes.Get(1)->AddApplication(receiver1);
    receiver1->SetStartTime(Seconds(0.1));

    Ptr<DioReceiverApp> receiver2 = CreateObject<DioReceiverApp>();
    receiver2->Setup(nodes.Get(2), enableMitigation);
    nodes.Get(2)->AddApplication(receiver2);
    receiver2->SetStartTime(Seconds(0.1));

    // Simulate DIOs being sent by attacker every 1s
```

```
    Simulator::Schedule(Seconds(1.0), &DioReceiverApp::ReceiveFakeDio,
    receiver1, 1);
    Simulator::Schedule(Seconds(2.0), &DioReceiverApp::ReceiveFakeDio,
    receiver1, 1); // replay
    Simulator::Schedule(Seconds(1.5), &DioReceiverApp::ReceiveFakeDio,
    receiver2, 1);
    Simulator::Schedule(Seconds(2.5), &DioReceiverApp::ReceiveFakeDio,
    receiver2, 1); // replay

    Simulator::Stop(Seconds(simTime));
    Simulator::Run();
    Simulator::Destroy();

    return 0;
}
```

Key design choices and their meaning

- `ReplayAttackerApp` is attached to `nodes.Get(0)` (node 0) but the program does not use NS-3 networking primitives to actually transmit packets. Instead, manual `Simulator::Schedule` calls model the delivered DIO messages and replays.
- The scheduled calls show both the original DIO arrival and the later replay; when mitigation is `true`, the receiver will drop the second call for the same sequence number.

0.8 Detailed flow of events (temporal)

For the attacker demo, the scheduled events produce the following timeline (example):

- 0.1 s: Applications start.
- 1.0 s: Receiver1 receives seq=1 (accepted).
- 1.5 s: Receiver2 receives seq=1 (accepted).
- 2.0 s: Receiver1 receives seq=1 again (this is the simulated replay — dropped if mitigation enabled).
- 2.5 s: Receiver2 receives seq=1 again (replay).
- Simulation stops at `simTime`, e.g., 10 s.