EE122 Final Project

Extended Abstract

Subverting Attacks in Dense Decentralized Networks

Mukhamediyar Kudaikulov

Summary

In this project, I explored an attack on dense networks which allow participation of anonymous actors, considered a few defenses, and primarily focused on building a simulation environment to model such scenarios. The goal of the project was to enable the simulation of numerous attack and defense scenarios, and allow for the simulation to be configured according to the user's needs. I successfully simulated the probing technique, the use of supervisory routers and randomized routing to combat the Blackhole Attack.

Background

In this project, I considered decentralized distributed computer networks with dense topologies that allow unrestricted participation of anonymous nodes. Examples of such networks are TOR, MANETs and various VANETs. These networks route traffic through nodes, and every node could be the source, destination or the intermediary hop of some traffic. Such networks by their very nature are susceptible to the infiltration of malicious nodes, which can perform an array of attacks, such as Blackhole Attack [1], Sinkhole Attack [3], Traffic Analysis Attack [4] and many more. In this project, I decided to build a simulation that could be used to model these attacks. Due to time constraints, my focus was primarily on testing the efficacy of defenses against the Blackhole Attack, exploring strategies such as probing, supervisory nodes, and randomized routing.

Attack

Blackhole Attack [1][2] is a kind of attack where the participant is mimicking behavior of a regular node. It advertises the existence of a path to a subset of nodes, but instead drops the packets it receives.

In practice, consider TOR [5], which uses its nodes (TOR relays) to anonymize the traffic passing through them. If there is a sufficient number of Blackhole Attackers present, the service would not be possible as the process of routing and handshake would have an overhead that is too costly to be practical. Another example is such an attacker in VANETs. A blackhole attacker could prevent a vehicle from learning about an upcoming danger on the road.

Defenses

- **Probing**

In the presence of blackhole attackers, a good way to mitigate the attacks is to optimize routing. As such, it is important to identify the adversarial nodes first. I considered the probing technique [6] [7], which sends a ‘bluff’ test packet, and examines the response of the next hop router. The point of probing is that a lying blackhole attacker will communicate a false delivery, which we can check using various metrics. The paper “Bluff-Probe Based Black Hole Node Detection and Prevention” [6] discussed timing based metrics by sending a test packet to a nearby node through the potential blackhole router. In the case of this project, the network assumed is dense, so we can use probing to study the path of a decoy packet directly. In my tests, I used it to have the nodes report the next hop of a packet, and identify the nodes that drop packets.

- **Supervisory nodes**

A useful way to efficiently test ant Supervisory nodes are trusted nodes that are used to aid the workflow of the network. An example of that would be the “directory authorities” in TOR, which report on the topology of the network and the nodes’ metadata. In my tests, I configured the supervisory nodes to be able to route traffic through any available nodes, enabling me to individually assess the adversarial behavior of each node.

- **Randomized routing**

Another way to avoid falling prey to blackhole attackers is to route the traffic with the help of randomized routing [8]. This enables the traffic to avoid always taking the possible honeypot shortest path at the cost of potentially taking a suboptimal path, in the case that the next hop along shortest path was non-malicious. In my tests, I considered a naive routing algorithm that weighs the probability of using a link based on its advertised distance to the destination node.

Simulation Environment

I used python to create the simulation, and created an API to interact with the simulation. It allows the user to configure any and all objects in the network, and provides realistic defaults.

The objects that I considered and built for the simulation are Routers, Links, Packets and Network. I also added child objects of Routers, called Attacker and Defender, which for the purposes of the tests were used as the blackhole attackers and supervisory nodes.

Below are short explanations of the objects and their capabilities. The actual simulation implementation could be found in the Github repository [0]. The interface screenshots for each object can be found in the Appendix A.

Packet

For the purposes of the simulation, packets are a logical unit of traffic. They contain information about the sender of the packet and the destination Router. They have the capability to log their path, and provide a summary (see more in the *testing* section), and the full log of every edge (link) and node visited.
The packets also keep a status, which is one of:

- FRESH: packet is ready to be sent by the source, but not yet put on the next hop's link.
- SENT: packet has departed its source and is in transit.
- ACK: a packet represents an acknowledgement for a packet by the destination.
- DROP: a packet is dropped and is to be discarded by the router/link.
- RECV: a packet has completed a round trip (packet received by source and acknowledged), and no further action needs to be done.

Router

The router objects are the participants of the network under consideration. I enabled them to do multiple actions:

- Store packets,
- Alter packets,
- Wait for acknowledgements,
- Drop packets,
- Calculate best routing paths according to the protocol of their configuration.

The default configuration makes these routers act like simple relays. Any foreign traffic is simply routed to the next hop according to the next hop vector that it calculates using the Dijkstra's Shortest Path Algorithm. Any traffic originating at the Router is waited upon for an acknowledgement, and any traffic that is destined to it is met by an acknowledgement to the source Router. All routers are also referenced by their IPs.

Link

Links are the edges in the network. They serve as the connecting bridge between two nodes. In my tests, links are bidirectional, but this can be changed to be unidirectional. They have the capability to:

- Store packets (simulate packets being in transit)
- Block the packets for extended time ticks (see the network object's time definition)
- Put the packets in the processing queue of different routers to simulate packet delivery.

The link configuration cannot be changed by default, because I do not assume any on
- Link attackers.

Network

Network could be thought of as the manager object. It keeps track of the network state, and enables the user to call functions that will propagate updates to any subset of links and/or routers. It also keeps track of the perceived time in time ticks, and updates the state of the objects within the network on each tick. The network is able to perform an array of actions, including:

- Setting the routing algorithm for all nodes (however, a node could instead opt to run a different algorithm).
- Update the link states and weights.
- Perform DNS lookups.
- Return the current topology of the network in a user - interpretable format (for example, strings for printing).
- Change the topology using an extensive number of formats, including separate lists and dictionaries.
- Schedule packet sends.

The network could be further configured, the details can be examined in the repository.

Attacker and Defender

Attacker and Defender are extended from the Router class. These are pre-configured classes that allow more flexible configuration than typical routers. This enables more unique behavior, such as dropping packets as an attacker, or establishing arbitrary links as a supervisor node.

Testing

- **Probing Technique**

I used the following probing algorithm to test whether a “testSubject” node would drop a packet if it is sent through that router. I used it along the paths which consistently dropped packets. It is a convenient and useful method, because it abstracts away the kind of probing that would be done in practice, possibly requiring several hops, by grouping all trusted nodes into one single node and setting the weights equal to 0. This technique was subsequently used in some way in the tests mentioned below and shown in the Appendix B.

G: network topology = (V,E)

For v in V', require that v reports v'if after using some link (v, v') a packet is dropped (malicious nodes report an innocent node)

Let p = test packet

def generateTrustedNode(v) -> node:

Add trusted v' to G, link (v, v') with weight 0.

return v'.

def Probe (supervisor, testSubject, p):

Let v' = generateTrustedNode(testSubject)

Establish Links (supervisor, testSubject) of weight 0

Set p = Packet(src: supervisor, dst: v')

Put p on network

If p is dropped, testSubject is malicious

- **Testing Environment**

I've also added a testing suite to make testing easier, such as providing default routing algorithms (Dijkstra and its derivatives), and also functions to perform probing on a path that a packet traversed.
The simulation also provides convenient methods to perform a quick network setup. Those include:

- generating a connected network with a connectivity amount of the user choosing (ie, how likely any two nodes are to share an edge between each other),
- installing new nodes post initialization of the network,
- Installing a supervisor node with a link connecting to each router.

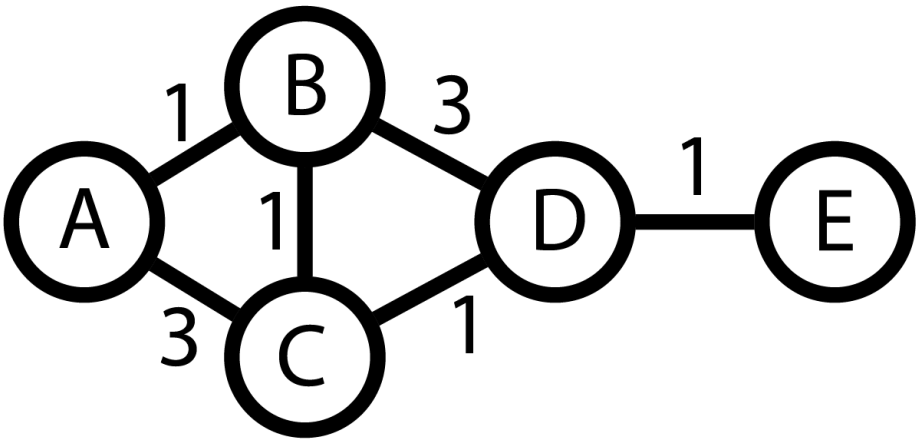
These testing methods were subsequently used to perform tests in Appendix B.

- **Tests**

The tests attached in Appendix B demonstrate some capabilities of the simulation.

In my basic tests, I set up a simple graph like in the diagram below. Then in my tests, I tried multiple scenarios, which included:

- setting node D as a malicious node,
- setting one of the edges to be of infinite weight (to test the dropping and retransmitting functions),
- trying the probing methods to identify that D is a malicious actor.



In the other tests I considered large graphs with 100+ nodes, and different connectivity levels. In all cases, the attacker was successfully identified by using probing, either initiated by the user themselves or through a comprehensive scan conducted by the supervisor node. Also, I tried a randomized routing algorithm, which, however, at times did not converge as fast enough, because it would choose looping paths.
The results of all tests could be found in Appendix C.

Conclusion

This simulation environment offers a very flexible configuration scheme to test a wide range of attacks on dense networks. It has the capability to configure each object in the network individually, and simulate varying behavior of network participants.
While in the project, my focus was on testing the Blackhole Attack, other configurations could allow for testing different behavior of Routers: Sinkhole Attacks, Traffic Analysis Attacks and others.
The tests that I included tested the probing technique, supervisory nodes and Blackhole attackers. These tests ranged from very simple networks to networks with more than a 100 participants.
During the testing portion of the project, I was able to verify empirically that the proposed defenses of probing, the use of supervisory nodes and randomized routing do in fact work, and could be further studied and improved.

References

[0] Simulation Repository: https://github.com/myxamediyar/network-simulation

[1] Fazeldehkordi, Elahe, et al. “A Study of Black Hole Attack Solutions.” Chapter 2 - Literature Review, sciencedirect.com, 2016, doi.org/10.1016/B978-0-12-805367-6.00002-8.

[2] Mastouri, M. A., and Hasnaoui, S. "Effect of the Black-Hole Attack in Vehicular Ad-Hoc Networks." 2021 26th IEEE Asia-Pacific Conference on Communications (APCC), Kuala Lumpur, Malaysia, 2021, pp. 205-210. doi: 10.1109/APCC49754.2021.9609947, https://ieeexplore.ieee.org/document/9609947

[3] Salehi, S. Ahmad, et al. "Detection of Sinkhole Attack in Wireless Sensor Networks." 2013 IEEE International Conference on Space Science and Communication (IconSpace), Melaka, Malaysia, 2013, pp. 361-365. doi: 10.1109/IconSpace.2013.6599496.

[4] Basyoni, Lamiaa, et al. “Traffic Analysis Attacks on TOR: A Survey.” *Readings*, MIT Libraries, 25 Jan. 2023, css.csail.mit.edu/6.858/2023/readings/tor-traffic-analysis.pdf.

[5] Dingledine, Roger, et al. “Tor: The Second-Generation Onion Router.” Design Paper, torproject.org, 16 Feb. 2016, svn-archive.torproject.org/svn/projects/design-paper/tor-design.pdf.

[6] Subathra, P., et al. "Detection and prevention of single and cooperative black hole attacks in mobile ad hoc networks." International Journal of Business Data Communications and Networking, vol. 6, no. 1, Jan.-Mar. 2010, pp. 38+. Gale Academic OneFile, link.gale.com/apps/doc/A232395759/AONE?u=anon~78291843&sid=googleScholar&xid=71a969c7.

[7] Sharma, S., Rajshree, Pandey, R. P., & Shukla, V. "Bluff-Probe Based Black Hole Node Detection and Prevention." 2009 IEEE International Advance Computing Conference, Patiala, India, 2009, pp. 458-461. doi: 10.1109/IADCC.2009.4809054, https://ieeexplore.ieee.org/document/4809054

[8] Sangaiah, Arun Kumar, et al. "Investigating Routing in the VANET Network: Review and Classification of Approaches." Algorithms, vol. 16, no. 8, 2023, article 381. https://doi.org/10.3390/a16080381.

Appendix A

Router

```python
# router object
class Router:
    def __init__(self, name: str = None):...

    def configure(self, name: str = None,
                  ip: int = -1,
                  network: Network = None,
                  packets: list[Packet] = set()):...

    def forwardAll(self):...

    def getNextHopFor(self, name: str):...

    def setHopWrapper(self, func):...

    def __process_basic(self, packet):...

    def checkAck(self):...

    def drop(self, p: Packet):...

    def setDropRandoms(self, yayOrNay: bool):...

    def retransmit(self, p, doRetransmit: bool = True):...

    def recordHop(self, packet: Packet, nextHop: str):...

    def reportHop(self, packet):...

    def __process(self, packet: Packet):...

    def updateRoutingTable(self):...
```

```python
    def updateRoutingTable(self):...

    def setRoutingAlgorithm(self, algorihtm):...

    def setAuxiliary(self, func, *args):...

    def getAuxiliary(self):...


    #region router Name...
    #region router IP...
    #region router Network...
    #region router Links...
    #region router Packets
    def getPackets(self):...

    def setPackets(self, packets: list[Packet]):...

    def addPackets(self, packets: list[Packet]):...

    def addPacket(self, packet):...

    def printNextHops(self):...

    def printNextHop(self, name: str):...

    #endregion

    def preprocess(self):...

    # Additional method to check if the router is destroyed
    def isDestroyed(self):...

    def destroy(self):...
```

Link

```python
# Link object
class Link:
    def __init__(self, u: int, v: int,
                 weight: float = 1,
                 network: Network = None):...

    def addPacket(self, packet: Packet):...

    def getEndpoints(self) -> tuple[Router, Router]:...

    def hasEndpoint(self, ip: int) -> bool:...


    def deliverPackets(self):...

    def setNetwork(self, network: Network):...

    def getNetwork(self):...

    def __repr__(self):...
```

Network

```python
# Network topology
class Network:
    def __init__(self, RTO: int = 20):…

    def updateTick(self):…

    def updateTickN(self, n: int):…

    def updateTickTill(self, packet: Packet,
                       status: Status,
                       stopTime: int = 100):…

    def printAll(self):…

    def changeTopology_l(self, links: list[Link],
                         routers: list[Router]):…

    def changeTopology_rw(self, edges: list[(Router, Router)],
                          weights: list[int]):…

    def changeTopology_nnal(self, node_names,
                            adjacency_list):…

    def refreshDns(self):…

    def triggerNodesExplore(self):…

    def __setLinkMap(self, links: list[Link]):…

    def __setNodeMap(self, nodes: list[Router]):…

    def addNode(self, router: Router, skipChecks: bool = False):…

    def removeNode(self, router: Router):…

    def addLink(self, link: Link, ignoreExisting: bool = False):…
```

```python
    def setLink(self, link: Link):…

    def changeDNSEntry(self, oldname: str, newname: str) -> bool:…

    def getDNSIP(self, name):…

    def tryDNS(self, name):…

    def incrementTime(self):…

    def scheduleSend(self, name: str):…

    def send(self, packet: Packet):…


    def getTime(self) -> int:…

    def getNodes(self) -> list[Router]:…

    def getNode(self, name: str) -> Router:…

    def getRandomNode(self, targetAll: bool = True,
                      failureCond: int = 100,
                      invalid: set = None):…


    def getNodeFromIP(self, ip) -> Router:…


    def setNode(self, node: Router):…

    def getLink(self, id_or_ipTuple) -> Link:…

    def setLinkWeight(self, id_or_ipTuple, w):…

    def getNodeIP(self, name: str) -> int:…
```

Packet

```
class Packet:
    def __init__(self, src: str, dst: str, logBit: bool = False,
                 status: Status = None, network: Network = None,
                 retransmit = True): ...

    def configure(self, status: Status = None, network: Network = None): ...

    def networkCheck(self): ...

    def refresh(self, router: Router): ...

    def setStatus(self, status: Status): ...

    def getStatus(self) -> Status: ...

    def getStatusStr(self): ...

    def incrTimeStamp(self): ...

    def getTimeStamp(self): ...

    def incrTimeSent(self): ...

    def getTimeSent(self): ...

    def incrRTcount(self): ...

    def setNetwork(self, network: Network): ...

    ###LOGGING METHODS
    def log(self, who, msg): ...

    def printLog(self): ...

    def printLogLast(self): ...
    def printLogRec(self): ...

    def printSummary(self): ...
```

Attacker and Defender

```
class Attacker(Router):
    def __init__(self, name: str, attackNum: int = 5,
                 targetAll: bool = True, failureCond: int = 100): ...

    def updateRoutingTable(self): ...

    def setRoutingAlgorithm(self, algorihtm): ...

    def preprocess(self): ...

    def reportHop(self, _): ...

class Defender(Router):
    def __init__(self, name: str, linkWdef: int = 0): ...

    def preprocess(self): ...
```

Appendix B

Basic Test 1: Simple Network

```
def basic_test1(): #basic: should complete round trip
    startMsg, endMsg = startEndTestMsg("Basic Test 1: Simple Network")
    print(startMsg)

    nodes, d = setupBasic()
    net = Network(40)
    net.changeTopology_nnal(nodes, d)
    testPacket = Packet("a", "e", True)
    net.send(testPacket)
    net.updateTickN(100)
    testPacket.printSummary()
    print(endMsg)
```

Basic Test 2: Simple Network with Bad Link

```
def basic_test2(): #basic: should drop
    startMsg, endMsg = startEndTestMsg("Basic Test 2: Simple Network with Bad Link")
    print(startMsg)

    nodes, d = setupBasic()
    net = Network(40)
    net.changeTopology_nnal(nodes, d)
    ip1 = net.getNodeIP('d')
    ip2 = net.getNodeIP('e')
    net.setLinkWeight((ip1, ip2), np.inf)
    testPacket = Packet("a", "e", True)
    net.send(testPacket)
    net.updateTickN(100)
    testPacket.printSummary()

    print(endMsg)
```

Basic Test 3: Attacker Present - Drop

```
def basic_test3(): #should report drop (unknown reasons)
    startMsg, endMsg = startEndTestMsg("Basic Test 3: Attacker Present - Drop")
    print(startMsg)

    nodes, d = setupBasic()
    net = Network(40)
    net.changeTopology_nnal(nodes, d)
    maliciousNode = Attacker('d')
    net.setNode(maliciousNode)
    testPacket = Packet("a", "e", True)
    net.send(testPacket)
    net.updateTickN(100)
    testPacket.printSummary()

    print(endMsg)
```

Probing Test 1: Proof of Concept

```
def probe_test1(): #proof of concept probing
    startMsg, endMsg = startEndTestMsg("Probing Test 1: Proof of Concept")
    print(startMsg)
    nodes, d = setupBasic()
    net = Network(40)
    net.changeTopology_nnal(nodes, d)
    maliciousNode = Attacker('d')
    net.setNode(maliciousNode)
    testPacket = Packet("a", "e", logBit=True, retransmit=False)
    net.send(testPacket)
    net.updateTickN(100)
    testPacket.printSummary()
    res = identifyDropperBasic(net, testPacket)
    print("\nDropper identified to be:", res)
    print(endMsg)
```

Probing Test 2: Randomized Probing in Dense Networks

```python
def probe_test2(): #randomized probing in dense networks
    startMsg, endMsg = startEndTestMsg("Probing Test 2: Randomized Probing in Dense Networks")
    print(startMsg)
    nodes, d = generateConnectedRandomGraph(101, 10, 0.8)
    net = Network(50)
    net.changeTopology_nnal(nodes, d)
    maliciousNode = Attacker('node-mal', 5)
    net.addNode(maliciousNode)
    net.triggerNodesExplore()
    print("\nDropper identified to be:", sendTestPacket(net))
    print(endMsg)
```

Probing Test 3: Probing With a Supervisory Node

```python
def probe_test3(): #supervisory node
    startMsg, endMsg = startEndTestMsg("Probing Test 3: Probing With a Supervisory Node")
    print(startMsg)
    nodes, d = generateConnectedRandomGraph(20, 10, 0.5)
    net = Network(10)
    net.changeTopology_nnal(nodes, d)
    #malicious node installed
    maliciousNode = Attacker('node-mal')
    net.addNode(maliciousNode)
    #superviory node installed
    supervisor1 = Defender("supervisor1", np.inf)
    net.addNode(supervisor1)
    supervisor2 = Defender("supervisor2", np.inf)
    net.addNode(supervisor2)
    net.triggerNodesExplore()
    geneRun = makeSupervisionGene(net, supervisor1, supervisor2)
    print("\nDropper identified to be:", sendTestPacketSupervised(geneRun, net, supervisor1, supervisor2))
    print(endMsg)
```

Random Path Test 1: Proof of Concept

```
def random_path_test1():
    startMsg, endMsg = startEndTestMsg("Random Path Test 1: Proof of Concept")
    print(startMsg)
    nodes, d = generateConnectedRandomGraph(n=10, maxW=2, ⋯
    net = Network(RTO = 200)
    net.routingDefault = ProbabilisticDijkstra
    net.dropRandoms = False
    net.changeTopology_nnal(nodes, d)
    r1 = Router("r1")
    net.addNode(r1)
    net.addLink(Link(r1.getIP(), net.getRandomNode().getIP(), weight = 5))
    net.addLink(Link(r1.getIP(), net.getRandomNode().getIP(), weight = 5))
    net.addLink(Link(r1.getIP(), net.getRandomNode().getIP(), weight = 5))
    net.triggerNodesExplore()
    pack = Packet(r1.getName(), net.getRandomNode().getName(), logBit=True)
    net.send(pack)
    net.updateTickTill(pack, RECV, stopTime=300)
    pack.printLogRec()
    print(endMsg)
```

Appendix C

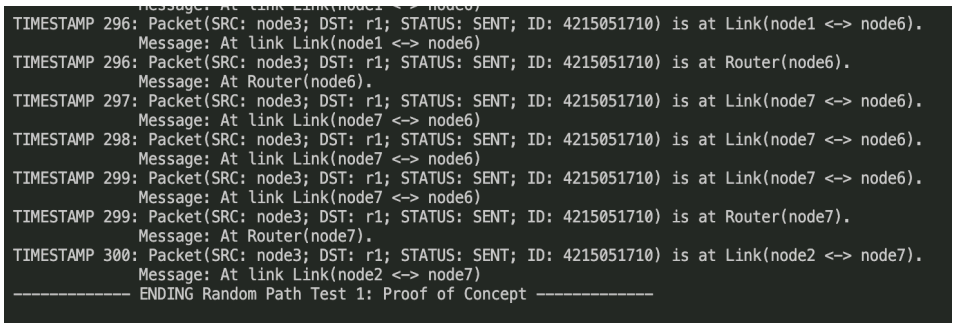
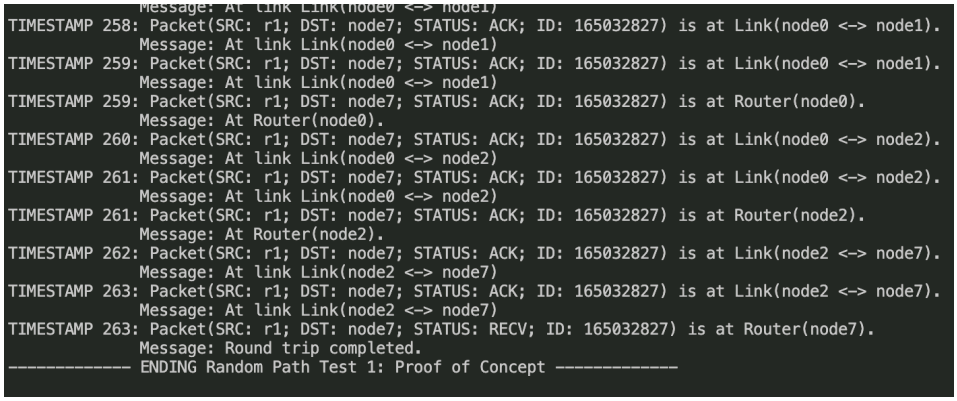
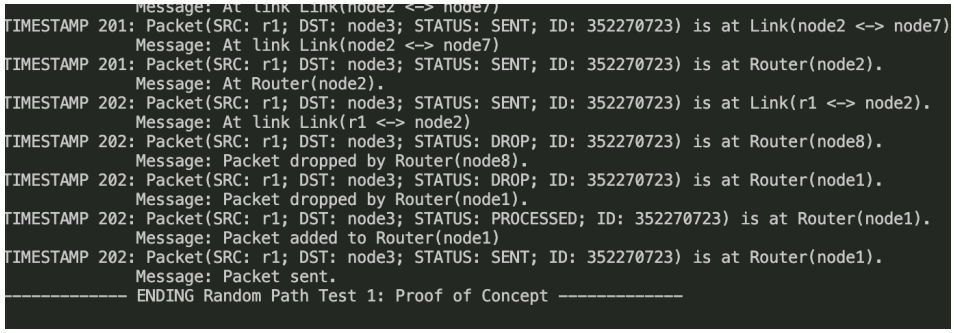
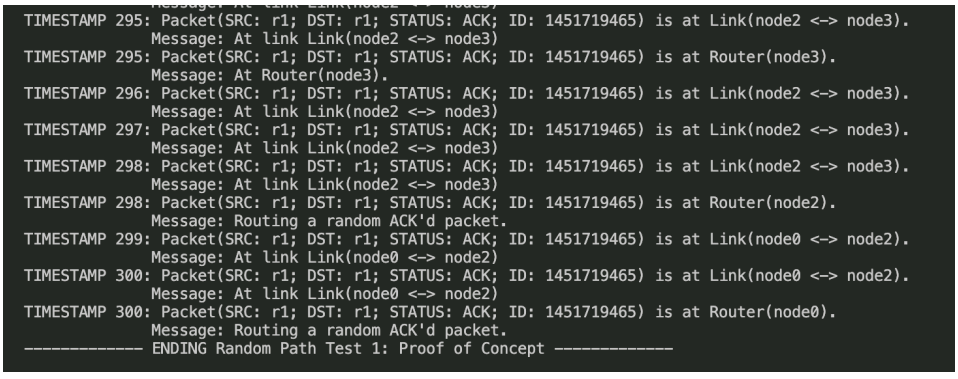
Basic Tests

```
---------------- STARTING Basic Test 1: Simple Network ----------------
Packet(SRC: e; DST: a; STATUS: RECV; ID: 3548450438) summary:
- Packet added to Router(a)
- Visisted nodes: a,b,c,d,e,d,c,b,a
- Round trip completed.
- Time sent: 1
- Last logged time: 17
----------------- ENDING Basic Test 1: Simple Network -----------------


--------- STARTING Basic Test 2: Simple Network with Bad Link ---------
Packet(SRC: a; DST: e; STATUS: DROP; ID: 2535913697) summary:
- Packet added to Router(a)
- Visisted nodes: a,b,c,d
- Packet dropped by source, stopped logging at Link(d <-> e)
- Time sent: 1
- Last logged time: 42
---------- ENDING Basic Test 2: Simple Network with Bad Link ----------


---------- STARTING Basic Test 3: Attacker Present - Drop ----------
Packet(SRC: a; DST: e; STATUS: DROP; ID: 2998269063) summary:
- Packet added to Router(a)
- Visisted nodes: a,b,c,d
- Packet dropped by Router(a).
- Time sent: 1
- Last logged time: 42
------------ ENDING Basic Test 3: Attacker Present - Drop ------------
```

More Comprehensive Tests

```
-------------- STARTING Probing Test 1: Proof of Concept --------------
Packet(SRC: a; DST: e; STATUS: DROP; ID: 869677137) summary:
- Packet added to Router(a)
- Visisted nodes: a,b,c,d
- Packet dropped by Router(c).
- Time sent: 1
- Last logged time: 42

Dropper identified to be: Router(d)
--------------- ENDING Probing Test 1: Proof of Concept ---------------


---- STARTING Probing Test 2: Randomized Probing in Dense Networks ----
Packet(SRC: node45; DST: node55; STATUS: DROP; ID: 2295397967) summary:
- Packet added to Router(node45)
- Visisted nodes: node45,node23,node-mal
- Packet dropped by Router(node23).
- Time sent: 1
- Last logged time: 52

Dropper identified to be: Router(node-mal)
----- ENDING Probing Test 2: Randomized Probing in Dense Networks -----


------ STARTING Probing Test 3: Probing With a Supervisory Node ------
Packet(SRC: supervisor1; DST: supervisor2; STATUS: DROP; ID: 1752956583) summary:
- Packet added to Router(supervisor1)
- Visisted nodes: supervisor1,node-mal
- Packet dropped by Router(supervisor1).
- Time sent: 31
- Last logged time: 42

Dropper identified to be: Router(node-mal)
------- ENDING Probing Test 3: Probing With a Supervisory Node -------
```

Randomized Routing Test

	Non-Convergence: Stopped at a Link
	Convergence: Successful Round Trip
	Non-Convergence: Stopped at a Router
	Non-Convergence: Ack sent Late