# 2IMS30 - Final Assignment
# Wormhole Attack on a RPL Network

Ruben Biskupec
r.biskupec@student.tue.nl

Tarık Doğaner
t.doganer@student.tue.nl

April 16th, 2023

## 1 Introduction

This report focuses on the design of a wormhole attack and defense against the Routing Protocol for Low-power and lossy networks (RPL), a protocol commonly used in the context of the Internet of Things (IoT).

It is divided into two main sections: (i) Attack Design and (ii) Defence Design. In the Attack Design section, we will present the design of a wormhole attack against RPL, while in the Defence Design section, we will focus on developing countermeasures against such an attack. We will provide a defense mechanism relying on minimizing the acceptance chance of unrealistic route advertisements on the candidate parent rank calculation process. We will not be doing the deployment part.

The modifications needed for the attack to work are on the MAC layer, but if the attack is successful, it will change the configuration at the RPL Protocol (Network layer). For the defense design, we are modifying files operating on RPL.

## 2 Attack Design

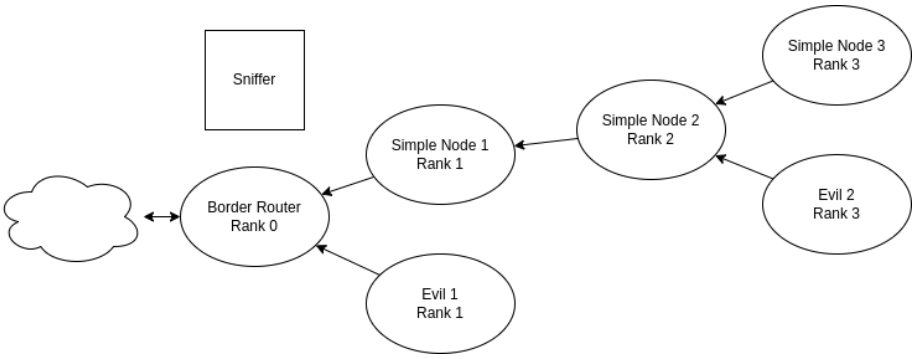


Figure 1: Normal Topology

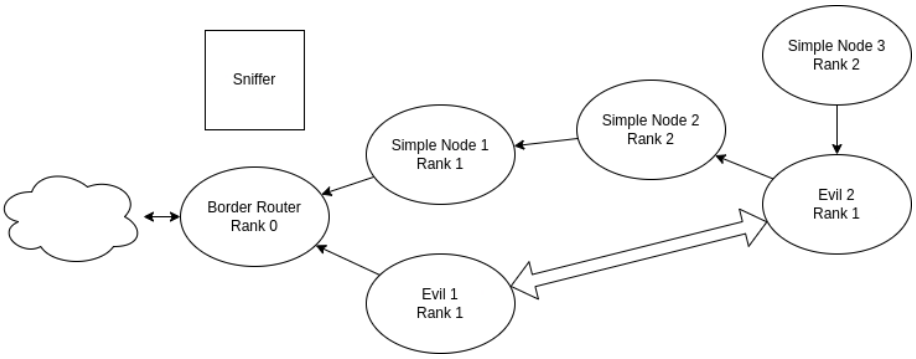


Figure 2: Attack Topology

The wormhole attack is very subtle, it involves the 2 adversary nodes creating a shortcut in the network by capturing packets at one location and replaying them at another through a communication channel, called a wormhole tunnel. The convenience of this new channel the routing service is lured to forward traffic through it, meaning that the malicious actors can potentially control a significant share of the traffic. At this point, the attacker can decide to (selectively) discard packets or eavesdrop on them. This attack can bypass many security mechanisms implemented at the network layer, such as cryptography (end-to-end encryption), making it a serious threat.

In this report, we present the design of a wormhole attack against RPL. Specifically, we focus on the method of attacking the use of a wormhole tunnel. First,

we describe how to design the attack, including the number of devices required and the software necessary to verify that the attack works. Next, we explain how to establish the wormhole tunnel and modify the routing protocol to redirect traffic through the tunnel.

## 2.1 Hardware and Software Planning

To simulate the *Wormhole attack* accurately in a lab environment, 7 nodes (Sensortags CC2650 devices) are required. The first device should act as a border router, being the root node of our network. Then, (at least) three of our nodes should be *simple nodes*, but they should be connected to each other, and not all of them to the border-router, so that we have a topology where we have a rank 3 device which will be misled by the second malicious node's falsely advertised lower rank. As stated just now, we need two malicious nodes so that they are inserted at different points of the network and establish a connection between them. Lastly, we need one device to act as a sniffer, and capture the traffic between all other devices. Thus, (at least) seven devices are needed to carry out the *Wormhole attack.*

In a Wormhole attack, one of the malicious nodes, which is closer to the root node, sends the advertisement packets it received to the other malicious node in the network so that it advertises a shorter route to its neighbors. Thus, it attracts all the traffic from its neighbors and sends it to the other malicious node.

The literature [3] shows that the tunnel can be created in many different ways:

- Encapsulation: In this attack mode, a colluding node at one end of an IoT network intercepts a Route Request (RREQ) packet from the other end and establishes a tunnel to trick nearby legitimate nodes into transmitting their packets through them. However, this may result in delays or lost packets.
- Packet Relay: This attack mode involves attacker nodes relaying packets between two legitimate nodes that are not within direct range of each other. The malicious node acts as a middleman and relays packets between the two valid nodes.
- High Power Transmission: In this mode, a single malicious node with high transmission capacity can launch the attack by overhearing an RREQ and rebroadcasting it with high-level capability, attracting legitimate nodes to overhear and broadcast the packet towards the destination. This allows the malicious node to become part of the network and launch the attack.
- Out of Band Channel: In this type of wormhole attack, the attacker nodes establish a high-bandwidth link between themselves and pretend to be nearby legitimate nodes. This attracts traffic through them and creates an out-of-band channel mode for the attack, requiring specialized hardware to launch.

We also thought of an alternative method to generate the tunnel. We will call this the *in-band* technique. Since both of our malicious devices will be in range of each other, we can proceed by changing the dag creation files in the 2 evil nodes, to allow them to have 2 parent nodes. To achieve this, we need to make a copy of the Contiki-ng operating system folder. Many functions in the /os/net/routing directory need to be changed, in particular the ones regarding dag creation, like rpl-dag.c and rpl-dag-root.c. After completing the changes, both nodes will be able to communicate with each other and also their original parent.

However, for this attack implementation, we will create an out-of-band communication channel as proposed in this paper [5]. Here, the 2 malicious nodes are connected to a laptop via a USB port. They will thus have an alternative communication protocol, since our Texas Instrument devices do not support Ethernet or Wi-Fi connections. The laptops will then communicate with their respective sensortags through the USB connection with custom Python scripts, as shown in the GitHub [1] provided with the paper.

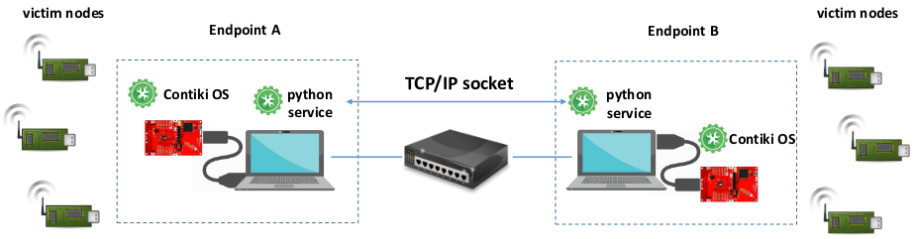


Figure 3: Hardware Setup [5]

## 2.2 Forcing the Topology

From the network layout shown in Figure 1, it becomes clear that we will need to force the topology. In particular:

- *Simple node 2* needs to drop packets coming from the *border router* and *evil node 1*.
- *Simple node 3* needs to drop packets coming from the *border router*, *simple node 1* and *evil node 1*.

To force the topology, we will let those 2 devices drop all packets received at the MAC layer, like explained in Lab 3. To acquire the information of the

[1] https://github.com/darvarr/wormhole

MAC address, we issue the command *tunslip*. Once we know the loopback IPv6 address (the one starting with fe80), we can figure out the MAC address with an online converter. Now, we convert the hexadecimal values into decimal format. We copy the Contiki-ng directory to make some OS changes. Again, as specified in the third lab, we change the code in the file /contiki-ng/os/net/mac/framer/framer-802154.c, and we add the lines to drop all packets coming from the selected MAC addresses.

## 2.3 Deploying the Attack

After establishing the connection between the two malicious nodes with the out-of-band channel, the first step of our attack is for the second node to replay the broadcast message received by the node closer to the root, so that the second malicious node's neighbors, which have higher ranks, should be deceived and change their routing tables to pass through the wormhole. The second step is making the further malicious node act as a sinkhole for its neighbors and attract all the traffic, then forwarding it all to the first malicious node.

At this point, the attacker has many options:

- Selectively discarding packets going through. This could happen with a voluntary delay, causing a denial of service.
- Discarding all packets.
- Eavesdropping.

For simplicity, we will discuss an eavesdropping attack. Again, following the attack explained in this paper [5] we need the following:

- 2 laptops connected to the 2 malicious nodes respectively, with a USB cable. These could also be replaced with more stealthy devices like a Raspberry Pi.
- Implementation of 2 functions, a *sniffer* and a *replayer*. Each function is implemented with 2 processes, one process running on Contiki-ng, one python process on the laptop. The Contiki-ng sniffer captures all the traffic and forwards it to the python sniffer, which will forward it to the other endpoint via TCP socket. The replayer is will receive the frames and retransmit them. These files can be found in the snip_usb_part1.py and snip_usb_part2.py files provided in the GitHub repository. [2]
- In a real-world scenario where all the nodes are spread apart and can not reach directly one another, the latency introduced by the wormhole might cause the nodes to ignore the acknowledgments, making the tunnel less convenient for routing. To circumvent this, the researchers introduced a *Proxy Acker* which generates ACK frames that are transmitted through a real and fast link. Thanks to this, the victim will not increase its ETX estimate.

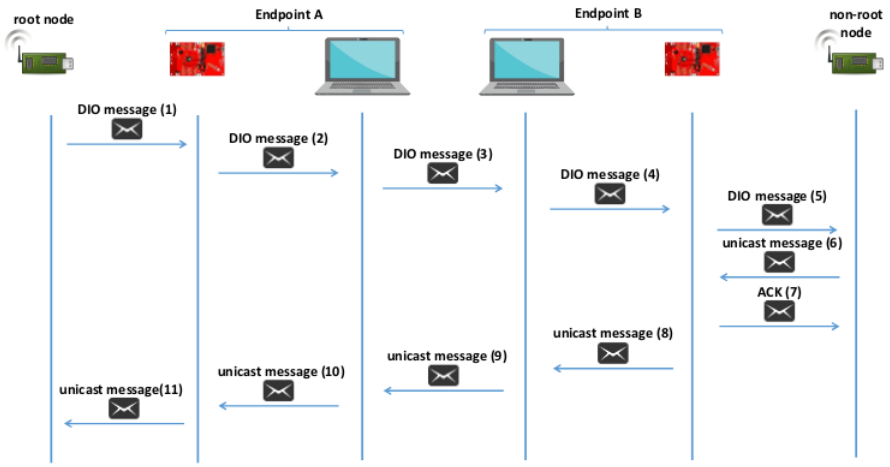


Figure 4: Proxy Acker [5]

In practice, we are going to change the file in /os/net/mac/csma/csma.c at line 89.

```
if(packetbuf_datalen() == CSMA_ACK_LEN) {
  /* Ignore ack packets */
  LOG_DBG("ignored␣ack\n");
} else if(csma_security_parse_frame() < 0) {
  LOG_ERR("failed␣to␣parse␣%u\n", packetbuf_datalen());
} else if(!linkaddr_cmp(packetbuf_addr(PACKETBUF_ADDR_RECEIVER),
                                    &linkaddr_node_addr) &&
    !packetbuf_holds_broadcast()) {
  /* OUR CONTRIBUTION
  Check if the frame is a unicast frame
  AND Check if the destination MAC address
  is on the other side of the wormhole */
  if(packetbuf_attr(PACKETBUF_ATTR_PACKET_TYPE) == FRAME802154_ACKREQ &&
    is_on_other_side_of_wormhole(PACKETBUF_ADDR_RECEIVER) {
    /* Generate and transmit an ACK frame */
    ackdata[0] = FRAME802154_ACKFRAME;
    ackdata[1] = 0;
    ackdata[2] = ((uint8_t *)packetbuf_hdrptr())[2];
    NETSTACK_RADIO.send(ackdata, CSMA_ACK_LEN);
  } else {
    LOG_WARN("not␣for␣us\n");
  }
  /* END OF OUR CONTRIBUTION */
} else if(linkaddr_cmp(packetbuf_addr(PACKETBUF_ADDR_SENDER),
    &linkaddr_node_addr)) {
  LOG_WARN("frame␣from␣ourselves\n");
}
```

This will act as the *Proxy Ack* described in the paper. In particular, the function *is_on_other_side_of_wormhole* will check if the destination address is in our pre-filled table with the MAC addresses on the opposite side of the wormhole. In a positive case, we will immediately send an ACK packet, by doing so, the victim will think we are in close proximity to the root.

Node that this change will be made only on the Contiki-NG [4] OS of the malicious node further away from the root node.

All the traffic going through the wormhole will pass by the Python sniffer program, which will save all the packets on the laptop which will have more than enough memory to store it. This is how eavesdropping will take place.

# 3 Defense Design

In this section, we design a defense mechanism that relies on the candidate parent rank calculation process of the nodes. Before finalizing our strategy, we considered several existing methods. Some approaches include the border router having additional software running and sending extra packets to investigate if the DODAG has some cyclic loops due to the wormholes [1]. Moreover, it is mentioned in [5] that countermeasures using the geographical positions of the nodes could work in theory, however, it requires additional hardware if the nodes are mobile and are vulnerable to GPS spoofing attacks. Therefore, we decided the optimal strategy should operate without any supplementary hardware or software, and it should work in a decentralized way so that each node deploys the countermeasure independently, preventing the attack during the first step. In the following sections, we describe the hardware and software planning and how to deploy the defense, before concluding our report with a discussion of the proposed mechanism's strengths and weaknesses.

## 3.1 Hardware and Software Planning

For the deployment of our defense against the *Wormhole attack*, no additional hardware is required. However, (at least) one node could be deployed as a sniffer to capture and analyze the traffic to observe and recognize that the defense is working accurately. Also, the existing software should be modified to deploy the defense in the victim nodes, but no additional software package is necessary for the implementation.

## 3.2 Deploying the Defense

As stated in the Defense Design section, we want to deploy a defense mechanism that checks the difference between two values, regularly updated with each DIO message received. The first value is Rank_Threshold, which is the difference between the rank of the node's parent and its own rank.

$$Rank\_Threshold = Parent\_Rank - Self\_Rank$$

The second value is Rank_Diff, which is calculated by the difference between the rank of the received DIO message and the rank value of that node.

$$Rank\_Diff = DIO\_Source\_Rank - Self\_Rank$$

After computing these two values after receiving a new DIO message, the node performs the check if $2 * Rank\_Threshold > Rank\_Diff$. If the result is True, then it assumes the DIO message is benign.

However, if $Rank\_Diff >= 2 * Rank\_Threshold$, the node concludes that the received DIO message is malicious and drops the packet. For instance, if our current rank is 20, our current parent's rank is 15, and the new DIO message we receive advertises a rank of 9. The Rank_Threshold value is 5, and

the Rank_Diff value is 11. So, we perform the check and see that $11 >= 10$, $11 >= 2 * 5$. Thus, the DIO packet is dropped since it is detected as malicious. We concluded that a coefficient is necessary because if not, there could not be any room for improvement for nodes to switch to a parent advertising a better route. Furthermore, the value of the coefficient could be fine-tuned if it was deployed with real devices, but we concluded that 2 has a good balance between valid better routes and unrealistic malicious advertisements.
In the following, we are providing a pseudocode block modifying the file *rpl-dag.c* and the function called *acceptable_rank* which is called in the *rpl_process_parent_event* function.

```
acceptable_rank(rpl_dag_t *dag, rpl_rank_t rank)
{
    *** We add this block to calculate the two values needed

    current_rank = dag->rank;
    p = dag->parent;
    parent_rank = p->rank;
    rank_threshold = parent_rank - current_rank;
    rank_diff = rank - current_rank
    is_dio_malicious = rank_diff >= 2*rank_threshold

    ***


    return rank != RPL_INFINITE_RANK &&
        ((dag->instance->max_rankinc == 0) ||
        DAG_RANK(rank, dag->instance) <= DAG_RANK(dag->min_rank +
        dag->instance->max_rankinc, dag->instance \

        //Above is the unmodifed code block, we add this conditional check below

        && !is_dio_malicious));

}
```

The *rpl_process_parent_event* function is called when there is a DIO message received from a new candidate parent node, so it checks if the advertised rank is in the acceptable range with the *acceptable_rank* function. Thus, our defense mechanism is inserted so the function now checks whether the received DIO message is malicious or benign as well.

### 3.3 Discussion

Our proposed defense mechanism works in scenarios where malicious nodes advertise too unrealistic routes (in our case equal to or more than double the current rank difference), so that it is easier to detect a case 'too good to be true'. If a smarter attacker realizes that nodes deploy a threshold mechanism for the rank values in DIO messages and ignores ones that are above the threshold, they could manipulate the rank values in messages and advertise more realistic routes, which means combining a *Rank attack* with a *Wormhole attack*.

To mitigate the extent of such a combined attack, the coefficient may be modified and optimized in the threshold check mechanism, which could be updated depending on the delay time of the packets arriving from the root node. On the other hand, instead of a coefficient, a constant value can be used and the check can be done as follows $Rank\_Diff >= Rank\_Threshold + c$. However, the constant value should be determined and fine-tuned according to the characteristics of the network, and since we as a group are not implementing the deployment with real devices, we could not compare and optimize the constant and coefficient approaches to see which one performs better.

# References


[1] Cheng Chen, Fei Tong, Yujian Zhang, and Ziyan Zhu. A novel detection and localization scheme of wormhole attack in IoT network. In *2022 13th Asian Control Conference (ASCC)*, pages 1983–1988. IEEE.

[2] V. Davydov. Wormhole. `https://github.com/darvarr/wormhole`, 2018.

[3] Snehal Deshmukh-Bhosale and Santosh S. Sonavane. A real-time intrusion detection system for wormhole attack in the RPL based internet of things. 32:840–847.

[4] George Oikonomou, Simon Duquennoy, Atis Elsts, Joakim Eriksson, Yasuyuki Tanaka, and Nicolas Tsiftes. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX*, 18:101089, 2022.

[5] Pericle Perazzo, Carlo Vallati, Dario Varano, Giuseppe Anastasi, and Gianluca Dini. Implementation of a wormhole attack against a rpl network: Challenges and effects. In *2018 14th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 95–102. IEEE.