RPL-based Passive OS Fingerprinting in Low-power and Lossy Wireless Sensor Networks

Master's Thesis

A. R. Güdek

July 1, 2022

RPL-based Passive OS Fingerprinting in Low-power and Lossy Wireless Sensor Networks

Master’s Thesis

A. R. Güdek

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 8, 2022 at 09:00 AM.

TU Delft

This thesis was written for the Cyber Security specialisation of the master degree Computer Science at the Delft University of Technology. The report has been the result of many months of hard work, starting back in November 2021, and culminating in the thesis defence performed on July 8, 2022.

This research lies on the intersection between cyber security and IoT, and may be of interest to anyone with a curiosity for network reconnaissance and fingerprinting in general, or within the area of IoT and 6LoWPAN-based mesh networks. With the investigated standards and protocols still being emerging technologies, there is still room to mitigate possible dangers and flaws before, if any, wide-scale adoption occurs. This research attempts to present one such flaw, namely the leakage of operating system information from RPL network traffic.

I would like to thank my family and friends for their continued support throughout my entire education, providing me with a healthy and nurturing environment in which I could grow and express myself. I would also like to thank my thesis supervisor Georgios Smaragdakis for his weekly supervision and guidance, in addition to the thesis committee members Kaitai Liang and George Iosifidis for all their valuable inputs.

A. R. Güdek
Delft, July 1, 2022

Abstract

RPL-based Passive OS Fingerprinting in Low-power and Lossy Wireless Sensor Networks

Since the introduction of networking protocols for Low-power and Lossy Networks (LLN), many implementations have been created in the form of (real-time) Operating Systems (OS), simplifying their usage and making the technology more widely available. The low cost and low complexity of LLNs allow for large-scaled wireless sensor deployments, making it infeasible to frequently renew the deployed hardware. In addition to this, the low processing and memory requirements of the hardware increase the difficulty of patching their software or adding new security features to it, making them useful targets for hackers. The knowledge of what OS is running on a device is valuable, as each OS will have their own set of security issues and mitigation methods. We investigate the possibilities of OS fingerprinting in the LLN space by simulating networks of devices running a variety of OSes. Focusing specifically on the RPL routing protocol, we build a classifier based on packet headers and communication meta-data, able to differentiate between the investigated OSes. From the results, we illustrate the feasibility of OS fingerprinting based on RPL control frame header contents.

Keywords:
Operating Systems, fingerprinting, Networking, WSN, LLN, RPL, side-channel, meta-data

Contents

1 Introduction

Recent advancements in networking technologies and the introduction of the Internet of Things (IoT), the aim to connect all kinds of "things" to the Internet to allow for centralised control and the creation of smart automation in home, urban and industrial settings, has lead to an explosion in the number of devices connected to the Internet. The speed, latency and setup complexity issues accompanying this rapid expansion has led to endeavours towards easier to use goal-oriented technologies.

One such example, specifically designed for the use of data collection and automation, is the creation of Wireless Sensor Networks (WSNs), also known as Low-power and Lossy Networks (LLNs). WSNs are collections of low-cost and low-power sensor devices (also called nodes or motes) interconnected with the use of wireless antennae, able to collectively and autonomously form and maintain a network. These generally densely deployed networks of nodes often consist, for reasons of financial and operational feasibility, of battery-powered devices with low memory and computation capabilities, therefore requiring protocols and software with low memory footprints making efficient use of the available resources of the underlying hardware.

The core building block of this technology is the 802.15.4 wireless communication standard by IEEE [36]. It defines the physical and Media Access Control (MAC) layers for a low-speed, low-cost and low power consumption wireless communication technology able to reach a transfer rate of 250 kbit/s. Several protocols have been built on top of this technology, such as Zigbee, Thread and 6LoWPAN, the latter being an effort to combine the low cost of 802.15.4-based devices with the power of IPv6 by forming an adaptation layer between them [35]. The packet routing behaviour is decided by RPL [12], the IPv6 Routing Protocol for Low-Power and Lossy Networks. This routing protocol defines the mechanisms required to autonomously set up a network topology and routing paths between the nodes.

Even though these protocols are fairly recent, many researchers and companies have created their own solutions with their own use-cases in mind, leading to the development of multiple Operating Systems (OS) for LLNs. As these technologies become more widely adopted, it also opens up new avenues for those with malicious intent to misuse. The low cost and low complexity of LLNs allow for extremely large-scaled deployments, making it unfeasible to frequently renew the deployed hardware. In addition to this, the low processing and memory requirements of the hardware increase the difficulty of patching their software or adding new security features to it, making them useful targets for hackers. The wide variety of OSes able to run on these devices also adds an additional hurdle to protecting the networks, each having their own set of security issues and mitigation methods.

The knowledge of what OS is running on what device is valuable, as it may assist network owners to better support and defend their networks, but also as it may ease the work of those with malicious intent to infiltrate and exploit them. Attackers may employ device or OS fingerprinting to maximise their knowledge on the system under scrutiny, and thus minimise the time and effort required for successful infiltration. This information can then be used to, for example, quickly identify vulnerabilities and tailor their exploits to the specific OSes identified [8], or insert malicious devices into the network that mimic legitimate behaviour. Because of this, several researches have proposed methods to fingerprint specific devices or identify underlying OSes [5, 28, 41, 43, 59, 72], which may then be used by Intrusion Detection and Prevention Systems (IDS, IPS or IDPS) in order to enforce additional authentication steps or identify intruders in the network [9, 27]. Another interesting use-case for OS fingerprinting in IDS context is to filter out false positive intrusion detections by checking whether the detected intrusion is at all possible on the OS running on that device [31].

While also having adequate reasons for defensive deployment, given the dangers of allowing outsiders to easily fingerprint the OSes running on deployed devices, any such methods should be quickly identified and mitigated. The goal of this study is therefore to identify if any possibility exists for OS fingerprinting in LLNS, particularly focusing on the RPL routing protocol within the 6LoWPAN network stack, and thus answering the research question whether it is possible to perform OS identification by passively observing RPL protocol behaviour in LLNs. We attempt to achieve this by investigating differences in protocols header contents and other high-level behaviour.

This document is outlined as follows. Chapter 2 breaks down the relevant technologies and existing fingerprinting methods in the form of a literature study. Based on this, a research hypothesis is constructed in chapter 3. In Chapter 4, we select the components necessary for our experimentation, the details of which are described in chapter 5. The experiment execution process is detailed in chapter 6, after which the obtained results are discussed in chapter 7, leading into the conclusion provided in chapter 8. Finally, chapter 9 lists any questions remaining open after this study, and provides options for future research.

2 Related Research

We investigate existing research in the area of RPL and fingerprinting, in order to understand the possibilities for OS identification within the context of RPL. First we describe the protocols in the network stack with which RPL is used. We describe the core workings of these protocols and attempt to find any indicators for the ability to create unique OS fingerprints from their behaviour. Next, methods are described with which RPL traffic can be collected in order to be used for fingerprinting. The chapter is concluded with the previous works on OS fingerprinting in general, and its application within the context of RPL.

2.1. RPL

The IPv6 Routing Protocol for Low-Power and Lossy Networks [12] (RPL),is a routing protocol designed by the Internet Engineering Task Force (IETF). It was created as a means to meet the requirements set forth in earlier Request For Comments (RFC) by IETF on use-cases of LLNs under home, building, urban and industrial automation scenarios [11, 20, 22, 66]. This routing protocol defines the mechanisms required to autonomously set up a network topology and the routing paths between nodes in LLNs. Having been designed to comply with the layered architecture of IP, it does not rely on any specific link-layer technology. However, as its conception was brought about by the need for an autonomous, low-power routing protocol in constrained and lossy networks, the creators of the standard adopted the ability to operate within such environments as their main design principle. Therefore, in the context of this study, RPL is used as the network layer routing protocol within the 6LoWPAN network stack, as shown in figure 2.1. Within this stack, RPL is built on top of the 802.15.4 and 6LoWPAN protocols.

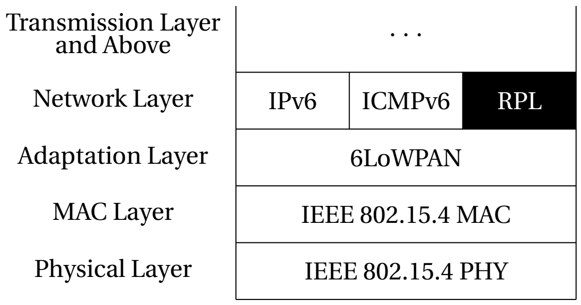


Figure 2.1: The position of the RPL routing protocol in the ISO OSI model within the 6LoWPAN network stack.

2.1.1. IEEE 802.15.4

The IEEE 802.15.4 standard [36] defines a physical layer and MAC sublayer for low-rate wireless communication on cheap, resource-constrained devices. Later becoming the basis for protocols such as ZigBee [1], Thread [38] and 6LoWPAN [35], it's a highly versatile wireless protocol with the ability to generate star and mesh topologies. It is designed to operate within multiple frequency bands, one of which being the 2.4 GHz band. Within this range, the standard defines 16 channels to use, numbered from 11 to 26. While this frequency band is also widely being used by the 802.11 (Wi-Fi) protocol [2], which may cause some radio interference between the two protocols [77], the different physical frame formats and modulation methods set the 802.15.4 protocol widely apart from existing protocols.

The most notable feature of this protocol that makes it suitable for resource constrained devices, is its Maximum Transmission Unit (MTU) of 127 bytes. Its header size and addressing schemes have been designed to fit within 25 bytes, leaving up to 102 bytes for data transmission, however, some optional functionalities may requires additional header space, reducing this payload size further to 81 bytes. 802.15.4 Networks are divided into Personal Area Networks (PAN), each assigned a PAN ID, consisting of Fully Featured Devices (FFD) and Reduced Function Deviced (RFD). While RFDs can only communicate with FFDs in direct range, the more capable FFDs provide additional functionality, such as hop-by-hop message relaying or acting as a gateway for external network access. For it addressing scheme, the 802.15.4 protocol relies on unique 8 byte EUI-64 identifiers. These are generally assigned during the manufacturing of the hardware and is defined as a device's long address by the protocol. Device addressing is then done using the 2 byte PAN ID and 8 byte long address. The nodes can also be assigned 2 byte short addresses which are locally unique within the PAN, either pre-set by the higher layers or assigned when joining the PAN. This way, addressing only requires 4 bytes per address, 2 bytes for the PAN ID and 2 bytes for the short address.

802.15.4 Also defines a link-layer encryption mechanism to secure its communications in the form of symmetric-key cryptography. The standard assumes that the key generation and management is handled by either the upper layers or the network maintainers. It proposes inter-link and inter-group encryption schemes where keys are either shared between two peers or are created for entire groups of devices.

2.1.2. 6LoWPAN

As addressing within 802.15.4 networks differs from the default IP addressing used on the internet and since its 127 byte MTU is way below the minimum MTU of 1280 bytes required by IPv6 [19], the 6LoWPAN compatibility layer [35] was created to be able to connect 802.15.4 networks to the internet over IPv6. This protocol defines several header compression mechanisms with which to losslessly convert IPv6 frames into a format small enough to fit within the payload of 802.15.4 frames. The base IP Header Compression (IPHC) object is shown in figure 2.2, which defines how the IPv6 header information should be extracted and decompressed from the packet. Based on the contents of these fields, the IPv6 header can be populated with values provided directly following the base object (uncompressed), or by default values set by the 6LoWPAN specifications (compressed).

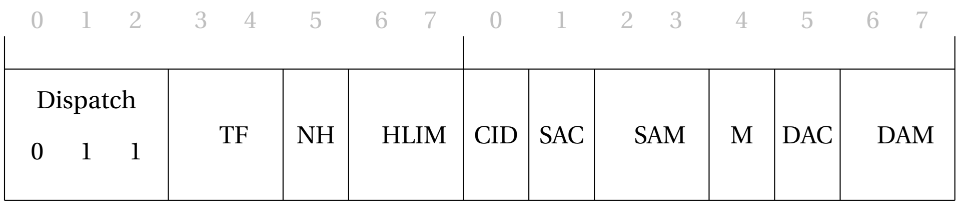


Figure 2.2: The 2-byte base IPHC header in 6LoWPAN.

The IPHC header defines multiple address compression mechanism, both for the source and destination IPv6 addresses. These are defined by the Source Address Compression (SAC), Source Address Mode (SAM), Multicast compression (M), Destination Address Compression (DAC) and Destination Address Mode (DAM) fields. Detailed information about the possible values for these fields and the corresponding address formats can be found in appendix C.

The protocol also provides compression mechanisms for IPv6 extension headers and upper layer protocols through the Next Header Compression (NHC) header. One or multiple NHC headers may follow the IPHC header, allowing for the compression of chained extension headers provided in the IPv6 header. The compressible protocols and extensions are managed by IANA[1]. Of the possible upper-layer protocols, as of this writing, only UDP can be compressed using NHC.

6LoWPAN also defines its own network discovery mechanisms [78], a version of the IPv6 Neighbour Discovery (ND) protocol [58] optimised for use within LLNs.

[1]6LoWPAN NHC header types - https://www.iana.org/assignments/_6lowpan-parameters/_6lowpan-parameters.xhtml#lowpan_nhc

2.1.3. Workings of RPL

RPL defines 3 node types which it forms topologies; hosts, routers and roots. Hosts and routers correspond to RFDs and FFDs from 802.15.4 respectively. Root nodes are routers with a special role of coordinating the network. Topologies within RPL are designed as Directed Acyclic Graphs (DAG) with one or more roots acting as sinks for the network, partitioned into multiple Destination Oriented DAGs (DODAG) having one DODAG per sink. This results in a tree-like structure, as demonstrated in figure 2.3, which, by design, avoids the creation of cyclic routes within the network.

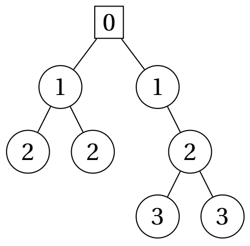


Figure 2.3: A simplified example of a network topology created with RPL consisting of a single DODAG. The square node is the root of the DODAG. The numbers denote a node's rank, in this example calculated as the hop distance to the root.

To uniquely identify a topology in RPL, 4 levels of identifiers are required:

- RPLInstanceID - This identifies a single RPL instance. This instance may contain one or multiple DODAGs that fulfil the routine tasks of the network.
- DODAGID - Each DODAG within a RPL instance is given its own DODAGID as a means to uniquely identify them.
- DODAGVersionNumber - Throughout its lifetime, a DODAG may choose to alter its structure as part of a global repair operation. In such a case, this version number is incremented instead of providing the reconstructed structure with its own DODAGID.
- Rank - Each node within a DODAG version is ranked by its distance to the root node. The rank of a node is higher if it's further away from the root. Calculation of this distance, and the metrics associated with it, depend on the Objective Function (OF) decided on by the network and may not necessarily be the physical distance to the root.

To generate such a topology, in a process called convergence, the protocol makes use of multiple types of control messages. These are introduced as new ICMPv6 messages (Internet Control Message Protocol for the Internet Protocol Version 6 [18]), and thus follow the same frame format. RPL defines the following control messages:

DIS

DIS frames are used optionally to initiate convergence for a newly added node, or in order to inform about other existing networks which a certain node might want to join.

DIO

The DIO contains information describing a RPL instance. Nodes can use the information in this control message to discover instances and learn of its parameters, and make topology decisions based on those parameters.

DAO

The moment a node joins a DODAG, it transmits a DAO to its parent. This indicates to the parent node that a new child has joined, and thus provides another downward route for traffic to flow. DAO frames may also include additional information on the set of reachable addresses via that child node.

DAO-ACK

Optionally, a node joining a DODAG and transmitting a DAO may indicate it expects an acknowledgement from its parent node by setting a specific flag in the sent DAO frame. This acknowledgement is performed in the form of a DAO-ACK frame.

Topology Generation and Maintenance

The topology generation process utilises these control messages to achieve convergence towards fully formed DODAGs. Convergence occurs in three steps, of which two are optional, from which bi-directional links are formed: Upward routes, i.e. routes towards the DODAD roots, and downward routes, i.e. routes towards the leaves. This process is displayed in figure 2.4 for a newly created network.

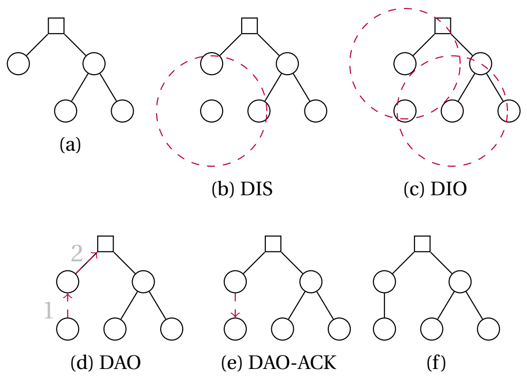


Figure 2.4: The convergence process in RPL.

Prior to setting up the network, certain nodes are given the ability to be DODAG roots by the owner of the network. These nodes start transmitting DIO frames over link-local multicast to inform nodes in the vicinity of their DODAG. Recipients use this information to decide on joining the DODAG and which nodes to use as parents. At this point, upward routes will have been formed, and nodes that have joined the DODAG will be able to communicate with the root, either directly or over multiple hops. Any node not having received a DIO, or desires to learn of other DODAGs in the vicinity, may transmit a multicast DIS, which in turn will trigger a DIO being transmitted by the recipients. Once nodes have decided on their parents and join a DODAG, and if they wish to utilise P2P and P2MP traffic, they transmit DAOs to their parents to inform the parent nodes of their children. Once all DAOs have been transmitted and processed, downward route creation will be complete, and all nodes within the DODAG will have a path defined from which they can be reached. In its most basic form, the convergence process only requires the use of 2 of these control frame types, namely the DIO and DAO.

Although the initial DODAG creation guarantees a non-cyclic topology, the potentially lossy nature of the underlying technology necessitates continuous data-path validation and loop detection. To avoid having to actively perform these checks and waste resources, the protocol has opted to insert a RPL Packet Information object in data packets containing the Rank of the transmitter. As Rank values are dependent on the height of the nodes in the tree, inconsistencies between the travel direction of the packet and the ranks of the sender and receiver nodes can immediately be identified. This will in turn trigger a local repair mechanism in order to fix the observed inconsistency.

As mentioned previously, RPL also supports a global repair mechanism. This mechanism is initiated by the DODAG root, and is triggered by incrementing the DODAGVersionNumber and transmitting a DIO with the new DODAGVersionNumber. On receiving a DIO with an incremented version number, child nodes will drop their previous routes and create a new DODAG based on the newly received DIO control frame. Even though the specifications define this behaviour, the events warranting global repair is left out of scope and is left to implementers of the protocol to decide.

Secure Mode

As mentioned in the control frame definitions, RPL provides the option for a secure mode. The specifications define 2 secure modes on top of the standard "unsecured" mode. In "preinstalled" mode, all nodes contain a pre-defined security key that is needed to generate encrypted RPL messages and to join the network. In the stricter "authenticated" mode, the pre-defined key only allows joining the network as a leaf node. To obtain privileges for becoming a router node in this mode, the specification mentions the usage of an authentication authority, but leaves the design of this authority and the key authentication process as out of scope. Research [62] and further detailed investigation into RPL implementations has indicated that no known OS provides support for these secure modes in their stable release versions. Because of this, the details of the security header and CC message type, and the operation of RPL under secure mode has been left out of scope for this research.

2.2. Research on RPL

Thanks to its area of use, the RPL routing protocol has also attracted a lot of interest in academia. A large number of research papers have been published investigating RPL, attempting to shed light on its performance and shortcomings regarding, among other, its routing performance, resource usage and security [44, 46, 51, 63]. We look into existing research to find any indication of a research gap.

There are several indicators of weakly defined specifications in existing research [17, 45, 62].

To optimise control packet delivery, RPL makes use of the Trickle algorithm [53] for DIO control frame scheduling, dynamically adjusting transmission frequencies based on the stability of the network's topology. Trickle requires three parameters to function:

- I_{min} - Minimum interval size
- I_{max} - Maximum interval size, indicated as the number of times the interval should be doubled
- k - Redundancy constant

The Trickle algorithm then uses these parameters by starting to transmit DIO frames in an interval of I_{min}. After every interval, the interval length is doubled until it reaches $I_{min} * 2^{I_{max}}$. Before every transmission, the transmitter compares the number of frames received during the previous interval that are consistent with the network's current state, compare it with k, and suppresses the transmission if k is exceeded. If at any point, a frame is received that indicates an inconsistency, intervals are reset to I_{min} and the entire process is repeated. The RPL specifications explicitly mentions four scenarios that reset the Trickle timer:

- An inconsistency is identified through the RPL Packet Information header.
- A multicast DIS is received without the Solicited Information option.
- A multicast DIS is received with the Solicited Information option and the current node passes the criteria in the option.
- A new DODAG is joined.

While providing default values for its parameters, the protocol gives implementers the freedom to change these in order to better fit their use-cases. The specifications also indicate that this timer should be reset under certain conditions, but this list of reasons is stated to be non-exhaustive and that implementers may extend this list as needed, leading to observable differences in control traffic [33, 47]. While transmission scheduling for DIO control frames follows the Trickle timer, a similar approach is not taken for DAO and DIS frames, and scheduling of these control frames is left for implementers to decide [17, 45].

The specifications also allow for variability in the selection of OFs, depending on the requirements of the network. The selection of an appropriate OF is important in order to avoid constantly altering and repairing the network topology, as this will generate unnecessary control traffic overhead [37]. Similar freedom is given when choosing between storing and non-storing modes of operation, where the specifications only require any node to support at least one of the modes. Additionally, of the option headers defined by RPL, certain options are also never explicitly mentioned being used and their use is left optional. Any implementation may therefore choose to use these by default, setting them apart from others.

Contiki OS defaults to a queue size lower than its alternatives [33]. When this queue fills up, which, for example, may occur during periods of high network activity, the device will no longer accept new packets until the former are processed. As a result, Contiki OS might be more prone to transmission failures, translating into observable retransmission behaviour.

These differences have also presented themselves affected the network's performance in several interoperability studies, where researchers have shown that networks perform more poorly when nodes with different OSes are used together in mixed RPL networks [33, 49].

From the aforementioned examples it follows that nodes within a RPL network will perform differently, depending on the underlying OS. As these differences are mostly behavioural in nature, it should be possible to observe them passively without joining the network and from this data to infer information the running OSs of a node.

2.3. Data Collection Methods

Before we can analyse and fingerprint RPL traffic, this data has to be collected from RPL networks. Within the context of the 6LoWPAN network stack, there are different ways of collecting network traffic data. Most existing research on RPL networks make use of simulators, which allows in-depth analysis of the behaviour of the simulated devices [3, 14, 30]. Simulators provide a network-wide packet-level view of all traffic within a network [56, 64], and can thus be used as a quick and simple method for data generation and collection.

For fingerprinting to be useful in real-world scenarios, however, methods are required to collect 802.15.4-based RPL network traffic between real wireless devices. For this task, a receiver is needed able to listen to 802.15.4 radio traffic, along with necessary software to decode and store the observed data frames. This receiver can either be a Software Defined Radio (SDR) [68], or any other transceiver with 802.15.4 protocol support [26, 50, 64]. With the correct software, it is also possible to reuse off-the-shelf node hardware as an eavesdropping device module [64]. By using multiple receivers with a common backbone, traffic flowing through the entire WSN can be monitored, even in very large low-density networks [80].

Another thing to consider is the possible ambiguity between RPL network traffic and traffic generated by other protocols without using RPL. Mistaking and attempting to fingerprint any non-RPL network traffic as a RPL OS may lead to undesired results. Ideally, any such traffic should be filtered out prior to fingerprinting, or classified as unknown. Investigating each layer of our selected network protocol, we find that such ambiguity can only happen under two scenarios.

The first scenario is when 802.15.4 is used with a payload that does not consider the first byte as a protocol identifier for the next layer. As opposed to the wider known Ethernet protocol which has a type field in its header indicating the protocol of its payload, 802.15.4 does not support such a mechanism. In the case of 6LoWPAN, the first byte of the 802.15.4 payload is used as a dispatch type that indicates whether this payload is 6LoWPAN (if first two bits of dispatch is not 00), and in what configuration it is being used. However, other protocols may not follow this format, and may thus produce payloads that look similar to 6LoWPAN frames. The encapsulated protocol must therefore be heuristically identified. Thankfully, the 6LoWPAN header, and subsequent RPL and UDP headers, contain several fields that are used to indicate the current and encapsulated protocols, like 6LoWPAN's *dispatch* and *next header* fields, or ICMPv6's *type* field. This reduces the odds of another protocol being mistaken for the packets we are interested in, down to negligible levels.

The other scenario in which the obtained traffic data could be ambiguous, is when 6LoWPAN is used to transfer data traffic, but with a routing protocol other than RPL. In this case, if we observe a data packet without a RPL packet information object, this might mean one of two things. Either the packet is transmitted without the use of RPL, or the OS from which it originated does not utilise this loop detection mechanism of RPL. For this research, we are only interested in the latter, while the former should be discarded. This issue can be circumvented by designing the classifier in such a way, that only data packets are considered that were transmitted by devices of which we know they make use of RPL, by having observed a RPL frame transmitted by that device within the duration of the network capture.

With these considerations, we can guarantee that all traffic obtained during the capture and used for fingerprinting is of the correct type, and no other kind of traffic is mistakenly classified.

2.4. Network Reconnaissance and Fingerprinting

Reconnaissance is the act of information gathering prior to engagement and is the first step of the cyber kill chain. By collecting as much info as possible, attackers can more easily find avenues for exploitation.

Fingerprinting is a component of reconnaissance which entails identification of features that differentiate systems from each other. Through these differentiation, important insights can be gained on the system under investigation, like the hardware model or software running on the system. Defenders can also use fingerprinting, for example for intrusion detection [8, 31].

Active vs Passive methods exist. Active methods require triggering responses from a target, while passive fingerprinting observes information a target publicises on its own. Because active methods require interaction with the systems under investigation, the targets can become aware of this and take measures against being fingerprinted, making it counter-productive for attackers.

2.4.1. Fingerprinting in WSNs

Existing research of fingerprinting within WSNs show that many available methods are active. Prior fingerprinting efforts for hardware or sofware identification are very limited. On the physical and MAC layers, Jenkins et al. [43] have shown possibility of hardware fingerprinting using mangled L2 frames, while Yan et al. [75] performed hardware identification via non-standard frame handling. Beck et al. [8] shows possibility of OS fingerprinting using the IPv6 Neighbour Discovery Protocol by observing responses to forged control frames. This protocol can also be used within the 6LoWPAN network stack, and may thus also allow for identification of WSN OSes.

Using passive methods, Prates et al. [65] perform device and sensor identification by creating device fingerprints from packet timing and response times observed between the gateway and the external network. While this method allows someone to infer that a certain traffic sample corresponds to a specific device of which previously a fingerprint was created, it does not provide additional insights into the technologies used within that device.

We can observe that there is a lack of research regarding passive OS identification methods within the 6LoWPAN network stack.

2.4.2. Passive OS fingerprinting

Investigating existing passive OS fingerprinting options, we find that there are three types of passive analysis [57]; Singleton, Sample and Stimulus-Response.

Singleton analysis focuses on the contents of a single packet. Protocol implementations may opt to choose different default header values if these are not explicitly enforced by their specifications, or make use of optional header options. The effectiveness of this method has been demonstrated in past with protocols like IP, ICMP, TCP, and more [57, 59]. However, not all header values are useful for this kind of analysis, so careful consideration is required when selecting our feature-set. These include values that are based on external factors or are unique for every node, such as node addresses, but also timers and counters that change on every frame.

Sample observes behaviour and changes within a set of packets. The inter-arrival time between network traffic and control frames has been shown to be effective in fingerprinting, for example within the context of 802.11 [28, 72]. Header field values that depend on timers or counters can also leak information through, for example, the values they are incremented by or how they are derived from other variables [57]. Sample-based analysis also has its difficulties. Counters may behave differently during retransmissions, but retransmission identification can be complicated, since it may be impossible to discern low-level retransmission from those that were initiated by upper layers of the network stack. Furthermore, timing-based analysis may be affected by network congestion, caching, or other causes of delay.

Stimulus-Response looks at behaviour triggered by the reception of other packets. Implementations may choose not to respond to certain packets, or respond differently based on the stage of the communication. Response behaviour of a device can provide insights into the internal state-machines of protocol implementation, which can be used for fingerprinting and identification. Here too, similar to sample analysis, dropped packets due to network congestion or traffic overloading can cause issues with these features [57].

Seeing that all types of features have both their advantages and disadvantages, we should ideally use a combination of Singleton, Sample and Stimulus-Response analysis.

3 Research Hypothesis

From the information conveyed in chapter 2, it is possible to hypothesise the possibility of performing OS identification using behavioural fingerprints from activity at the RPL layer of the network stack. This section attempts to summarise the core findings leading to this hypothesis, alongside with providing clear research questions, which by means of answering, will result in proving or disproving it.

3.1. Summary of Findings

[62] lists several underspecified elements of the RPL standard. Most notable are the underspecification of DAO transmission scheduling and events triggering Trickle timer resets. In addition, [45] mentions that DIS scheduling is also not explicitly defined in the specifications. Several studies have shown in other networking protocols that underspecification leads to variation in implementation among manufacturers, which can then be used to uniquely identify them by means of active or passive fingerprinting techniques [8, 28, 43, 75]. In RPL, this difference in implementation between OSes is also hinted at and demonstrably shown by means of textual analysis [47, 62] or interoperability studies [33, 49]. However, a search for existing research on passive OS identification in LLNs has yielded no results, clearly indicating a research gap in the area of OS fingerprinting in the LLN space.

3.2. Main Research Question

Is it possible to perform OS identification by passively observing RPL protocol behaviour in LLNs?
The main goal of this research is to assess the feasibility of performing OS identification by passively observing control messages and accompanying meta-data distributed by the RPL protocol. There are 3 avenues from which this question can be approached. The first, and simplest, is by observing header values used in control messages and identifying differences in default header values between OSes. A second approach is investigating how implementations react to header data they receive. The final option is looking at differences in scheduling of control messages for which the scheduling behaviour is underspecified in the specifications. Being able to successfully perform classification using any of the aforementioned methods will be enough to prove the possibility of passive OS identification.

3.3. Research Sub-Questions

Answering the main question can be done in multiple ways. Therefore, in this section, the main question is dissected and several sub-questions are provided. The main research question can only be answered with a 'no' if all methods below fail to provide valid results, whereas it is only necessary to successfully perform OS identification with one of the methods to be able to declare it feasible.

Is there a difference in default header options and values being used by the RPL implementations?
Having been created independently, and given the under-specifications and extension options in the RPL standard, the RPL implementations may display slight differences in the contents of packets they generate. An implementation may, for example, opt to use options objects by default in the options header that is not being used by the others. Another possibility is that reserved fields, which are generally ignored by the

recipients, are populated differently than defined in the specifications. Implementations may also differ in the way they respond to received control frames, either choosing to (not) respond, or respond with header contents that differ among implementations.

Is there a difference in scheduling behaviour of control frames between the RPL implementations?
A more protocol-agnostic method for fingerprinting that could also be attempted here, is the usage of meta-data of the observed network traffic as features. This includes factors such as frame scheduling, traffic flows, error-state behaviour, and more. Since we know that there is underspecified scheduling behaviour for DIS and DAO control frames, and DIO scheduling parameters are not fixed by the specification, these may result in observable difference in, for example, control frame timing and retransmission behaviour. This means that this method of fingerprinting could also prove useful within a RPL environment.

4 Information Gathering

Before we can start investigating differences in RPL implementations in the form of an experiment, we have to investigate the available OS candidates, as well as the means of gathering data necessary for fingerprinting. We describe the methods used to gather this information, along with the final choice of operating systems and experiment environments selected for the continuation of this study. This chapter also lists the fingerprinting features selected to be investigated.

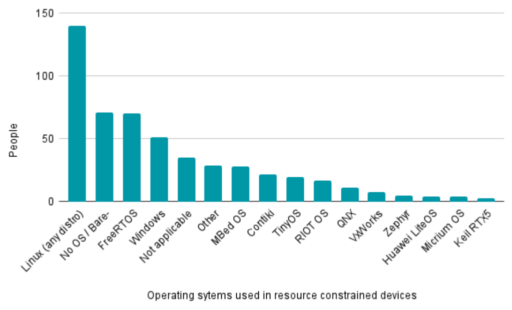
4.1. Existing RPL Implementations

We find out which OSes with RPL support by observing several sources, ranging from research papers and surveys to online sources related to IoT development. In chapter 2 we described several papers in which specific RPL implementations were investigated, giving us our first look into which OSes with RPL support exist and are used within academic circles, namely Contiki OS [21] and TinyOS [52]. In order to find other existing OSes, we take a more comprehensive look at other sources, such as OS survey papers, developer surveys, websites and code repositories.

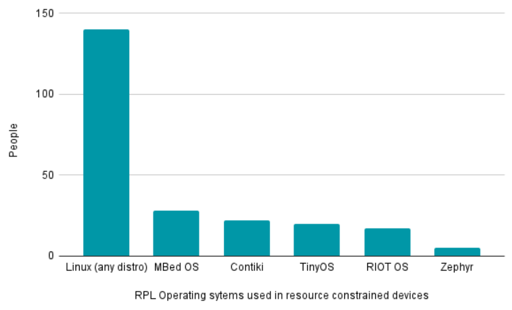
As a first step, we list all OSes mentioned in existing IoT, 6LoWPAN and RPL OS survey papers [16, 42, 69, 76] and go through a verification process to ensure they are suitable for our research. First we make sure these are open-source and still available by visiting their websites. We find that several OSes have been discontinued and their websites removes, so they can no longer be obtained (Nano-RK, MantisOS, SOS OS, NanoQPlus). Next, we make sure these OSes support the protocol stacks in question by searching for the RPL implementation code within the source code. This can be done easily by using the search functionality on GitHub or other git-based repositories. For other sources, a manual inspection is performed to find a RPL implementation within the code. Here too, we find that a number of mentioned OSes do not meet our requirements (LiteOS, NuttX, Erika, MansOS, RETOS, ChibiOS/RT). After eliminating all non-viable OSes from these surveys, all that remain are Contiki OS, TinyOS, RIOT [7], OpenWSN, ARM Mbed OS and Zephyr.

To better understand what OSes exist outside of academic use and their popularity, we take a look at the 2018 IoT developer survey[1] by the Eclipse foundation. In this survey, participants were asked which IoT technologies they used, with one of the questions being which OSes they used within their resource constrained devices. Even though this survey is performed yearly by the Eclipse Foundation, and more recent versions of this survey exist, we opt to use the 2018 version, as (to our knowledge) this is the most recent version for which the raw survey data has been published. The question regarding OSes used within resource constrained devices was answered by 362 people, of which the results can be seen in figure 4.1a. Once again we investigate which of these OSes are suitable for our research, resulting in the OSes visible in figure 4.1b.

[1]Key Trends from the IoT Developer Survey 2018 - https://blog.benjamin-cabe.com/2018/04/17/key-trends-iot-developer-survey-2018



(a) All IoT operating systems



(b) IoT operating systems with RPL support

Figure 4.1: Most used OSes within IoT on resource constrained devices based on the Eclipse 2018 developer survey.

These results show that Linux is the dominant OS used for IoT purposes, even for resource constrained devices. We find, however, that there is still no official RPL support provided by the Linux kernel as of the writing of this report. To achieve RPL operation wihtin Linux, the most proposed method is by running one of the other RPL OSes on top of Linux as an application using their native compilation option, and using a Serial Line IP (SLIP) antenna module that handles the 802.15.4 frame transmission and reception. Since this method makes use of the network stacks of the other OSes instead of having a distinct implementation, we expect the behaviour of Linux to be the same as the behaviour of the OS running on top of it. For this reason, we do not consider using Linux for our experiments.

Of the remaining OSes, we find that Zephyr only supports RPL up to version 1.13, discontinuing its support due to lack of maintainers since version 1.14. Since versions 1.13 and earlier have a smaller set of supported devices which are also relatively outdated, we deem this OS no longer useful for RPL use and remove it from our set of selected OSes.

Finally we also notice that Contiki OS has recently been replaced by Contiki-NG and is no longer being maintained. From existing research we do find, however, that most studies make use of the older versions of Contiki, and that it is still widely being used [70]. For this reason, we decide to include both versions in our experiments.

Considering all these factors, we are left with the OS selection seen in table 4.1. Next we will investigate what other requirements these OSes have and whether we can practically use them for our research.

Selected Operating Systems
Arm Mbed OS
Contiki OS
Contiki-NG
RIOT
TinyOS
OpenWSN

Table 4.1: Initial operating systems selected for investigation.

4.2. Experiment Environments

In existing research on WSNs, a plethora of methods have been used to create, deploy or simulate the behaviour under question [15]. These methods can be split into two main groups; using simulation software or deploying on real hardware. This section investigates and compares these methods, in order to find the most suitable environment with which to run our experiments.

As it is desirable to perform experiments as accurately as possible, the most straight-forward way to achieve this would be to set up sensor networks with real hardware and play out real-world scenarios. Several papers exist in which researchers have done so to evaluate the performance of WSNs in realistic settings [17, 23, 29, 33, 47, 49], the largest of which consisting of 69 nodes placed in grid formation within an area of 1km^2 [17]. This, while displaying the possibility of real-world deployment for experimentation, also makes obvious its infeasibility due to its cost and operational requirements. Researchers also raise the issue of time requirements and reliability of deployment, and the lack of repeatability of experiment, as not all parameters might be in the hands of the experimenters, such as outside interference [15, 60]. Also, while using real hardware gives us the most flexibility for platform support, we will be limited by which hardware we will be able to obtain, mainly due to a limited budget or availability of the hardware for purchase.

To alleviate some of the difficulties associated with test-bed setups, multiple experimental labs have been set up that allow researchers to quickly and easily deploy and execute large-scale WSN experiments [4, 6, 25, 71, 74]. A downside of this method is that the network traffic capturing is performed by hardware modules individually connected to each node, rather than from a single point with a sniffer module, thus being less representative of real-world scenarios. We are also, once again, limited by hardware platforms to only those provided by these labs.

A more popular approach for sensor network deployments in research is the usage of simulators, as can be noted by the large number of papers in which they are being used [3, 14, 17, 30, 33, 34, 39, 45, 48, 49, 54, 55, 70, 73, 79]. While testing with real hardware provides the most realistic outcomes, it also has its own set of complications, requiring specific domain knowledge to set up the necessary hardware, suffering from a lack of repeatability, and high space and cost requirements [15, 56]. Simulators alleviate these difficulties by providing capabilities for easily setting up empirically repeatable large-scale sensor networks. Using simulated environments also gives us the option to automate our tasks and run them in parallel, greatly reducing the time required for this step. Even at the cost of losing some of the environmental complexities inherent to real-world scenarios due to operating on approximations of the physical world, the usage of simulators is still encouraged as an initial method of research for testing new algorithms and creating proof-of-concepts [61]

Cooja

From surveys on existing LLN simulators, the Cooja simulator stands out with its full-stack simulation capabilities and multi-OS support [15, 24, 48, 67]. It allows its users to simulate multiple nodes as part of an 802.15.4-based wireless network. By making use of the MSPSim and Avrora harware emulators, Cooja attempts to mimic real hardware behaviour as closely as possible. This does, however, limit its out-of-the-box platform support to MSP and AVR-based hardware.

The simulator collects UART output, radio activity and packet transmission of all nodes, and contains a script execution module with which to automate interaction with the nodes using that data. This scripting environment can also be used for testing, for example by testing whether a specific output string is observed within a certain period of time.

Figure 4.2 shows the Cooja GUI during the simulation of an 8 node network, with all default plugins enabled. In addition, the simulator also includes a headless mode. In this mode, the GUI is not started and all tasks are run in the background. This allows for easy parallel execution, and automation of testing and data collection.

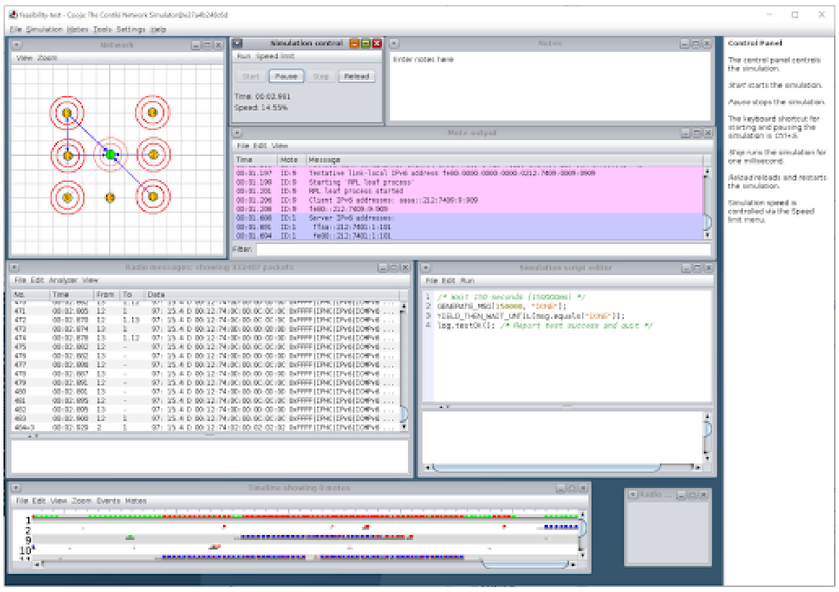


Figure 4.2: The Cooja simulation environment in GUI mode, running a network of 8 nodes.

Renode

The Renode simulator[2] is one of the more recent simulation frameworks, with a wide selection of features. It supports the emulation of a large number of hardware modules from architectures like ARM, RISC-V and x86. The emulations can interact with each other through the simulation of the physical mediums connecting them, like wired connections or wireless radio signals.

Figure 4.3 shows a 3 node simulation running on 3 emulated hardware modules within the Renode simulator. The window in the top left is the console from which simulations can be started and managed. Each emulated node creates a window in which the UART output of that node is displayed.

Renode allows for detailed management and inspection of the performed simulations. Apart from the possibility of attaching debugging tools to the emulated hardware, it has detailed logging capabilities of its hardware emulation, firmware execution and network simulation. It also provides an automated testing framework with which execution, testing and data collection can be automated and parallelised.

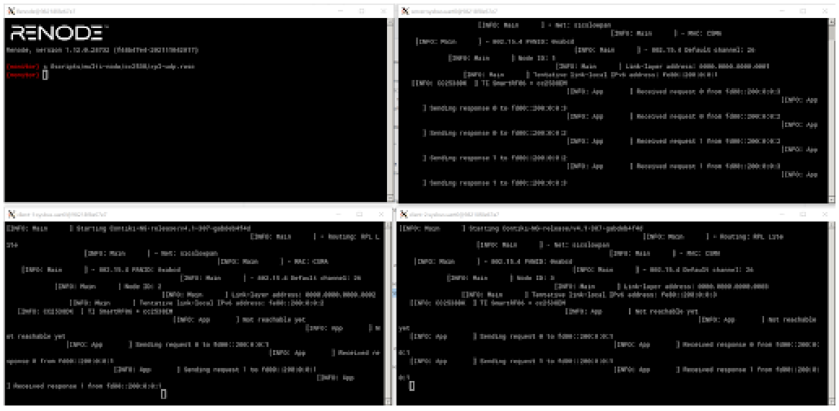


Figure 4.3: The Renode simulation environment running a network of 3 nodes.

Hardware

For hardware-based simulation we opt to use real hardware, which requires at least 3 hardware modules: 1 root, 1 leaf and 1 radio traffic sniffer. For this study, we are able to obtain 3 nRF52840 dongle modules

[2]Renode - https://github.com/renode/renode

by Nordic Semiconductor[3]. These are cheap hardware modules (< $10) with 802.15.4 protocol support and freely available programming software, which can be used for our experiments out-of-the-box without any additional hardware requirements.

Nordic Semiconductor also provides a free 802.15.4 sniffer software[4] that can be used with the same nRF52840 dongle modules. This software performs raw radio sniffing on a single channel, and sends the captured frames through its USB connection to a computer running Wireshark packet capture software with a special bridging driver. Thanks to this, any 802.15.4 traffic transmitted over the selected channel from within the sniffer's range can conveniently be logged to any file format supported by Wireshark. Figure 4.4 shows the setup required for this process.

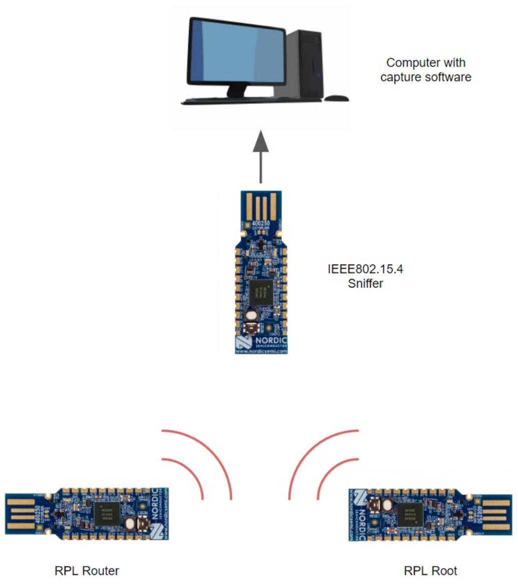


Figure 4.4: Necessary hardware setup to simulate a 2 node RPL network and collect its radio traffic.

OS Support

A detailed look into all of these options shows that they all have their limitations. Cooja has limited platform support, only allowing emulation of MSP and AVR-based hardware. These platforms are not supported by ARM Mbed OS, since this OS is created only for the ARM platform. While RIOT does partially support the MSP platform, its RPL implementation is too large to fit on any of the boards implemented by MSPSim.

Renode is still heavily under development, to the level that the emulation of some platforms are incomplete. The examples provided by Renode are generally based on Contiki OS, which correctly works out-of-the -box for many of the emulators. When attempting to emulate other OSes, however, we find that emulations either do not start, or do not generate any network traffic.

Using a hardware-based environment is difficult to automate, requiring manual uploading of the firmware, running the nodes and capturing of traffic. We are also limited due to the number of hardware modules we have on hand, thus hampering parallelisation.

Seeing all these limitations, we choose to use a combination of hardware and simulator-based methods. We test out all environment options per OS to make sure we can continue using them. For ARM mbed OS we find it has very specific hardware requirements for using 802.15.4 and RPL, which does not include any platforms available in any of our environments. Furthermore, while OpenWSN supports platforms available in all of the environments, attempting to run the OS results in failure to execute entirely or generate any traffic. This leaves us with only 4 OSes, and the continuation of the project will only make use of those 4. Table 4.2

[3]Nordic nRF52840 dongle - https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle
[4]nRF Sniffer for 802.15.4 - https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-802154

contains the final list of OSes with which the experiments will be performed, together with their environment support.

OS	**Cooja**	**Renode**	**Hardware**
Contiki OS	✓	✓	
Contiki-NG	✓	✓	✓
TinyOS	✓		
RIOT			✓

Table 4.2: Environment support of the selected OSes. The hardware environment makes use of ARM-based nRF52840 modules.

4.3. Feature Selection

We investigate possible features within the three types of passive analysis; Singleton, Sample, and Stimulus-Response. For Singleton analysis, we consider the header values available in the control frame headers as features, including the usage of optional header options. Sample and Stimulus-Response analysis allow for more freedom in features. For these we can observe, among others, changes in header values, behaviour that leaks information on workings of internal mechanisms and timing and scheduling patterns.

We should be selective in which header values we observe. Not all of them will be suitable for Singleton analysis. Retransmission identification can be difficult, complicating Sample analysis. Congestion or traffic overloading can cause issues with Stimulus-Response features. So, ideally a combination of Singleton, Sample and Stimulus-Response should be used.

In this section, we describe the contents of RPL frame headers and options objects, and list the header fields we select for our analysis. We also describe the Sample and Stimulus-Response-based analyses we will perform on the obtained data.

4.3.1. Control Frame Headers

For Singleton, we use all header fields defined by the specifications, except for fields that are designed to be different for each node (address, rank, etc.) This also includes usage of optional header options and the fields within those sub-headers. This section lists all relevant RPL headers and the features among them we select for fingerprinting.

DIS

The DIS control frame is the smallest of the RPL control frames, its base form only consisting of 2 bytes, as shown in figure 4.5. The two fields it defines are all reserved for future use and are set to 0 by the specifications. Due to this, we can not define any features based on the contents of the DIS base object. As part of its optional component, this control frame supports 3 options: Pad1, PadN and Solicited Information.

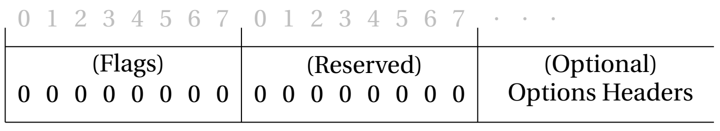


Figure 4.5: The DIS Base Object.

DIO

The DIO contains information describing a RPL instance. Nodes can use the information in this control message to discover instances and learn of its parameters, and make topology decisions based on those parameters. Its base object, shown in figure 4.6, contains information that identifies the DODAG the transmitting node is part of, among with some of the DODAG's parameters. The DIO control frame supports 6 options: Pad1, PadN, DAG Metric Container, Routing Information, DODAG Configuration and Prefix Information.

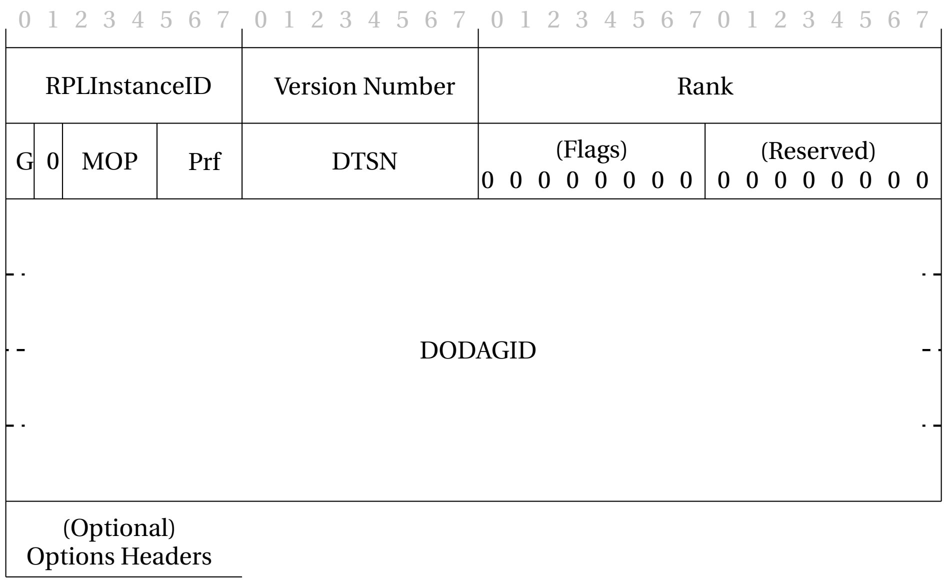


Figure 4.6: The DIO Base Object.

RPLInstanceID
This 8-bit field is an identifier for a RPL instance, which uniquely identifies a network instance. No default value for this field is provided in the specifications, so each implementation may have opted for a different default value. OSes implementers may also have chosen not to make this value editable, since this field's value does not affect the functioning of a RPL network if all nodes are supposed to only be a part of a single network, and the selected OSes currently do not support multi-DODAG networks.

Furthermore, a distinction is made between local and global RPLInstanceIDs, where global IDs should be unique for the entire network, while local sub-networks may be created by any node and identified through a local ID. Whether a RPLInstanceID, and consequently the network, is local or global, is indicated by the uppermost bit of the ID; 0 if the instance is global and 1 if it is local. This differentiation and how it affects the networks' behaviour is unexplored in the specifications, which makes it an interesting value to investigate and potentially use as a feature.

Grounded
This 1-bit flag indicated whether a DODAG is grounded or floating. Being grounded is defined in the specifications as *"being able to satisfy the application-defined goal"*. While no further details are given on what this goal may be, the implication is made that a DODAG is grounded if it can provide access for its children nodes to an external network. The vagueness fo this definition, however, may have led to different behaviour in OS implementations, so we include this field as a feature.

Mode of Operation
RPL currently defines 4 modes of operation indicated by the 3-bit MOP field, as depicted in table 4.3. Each of these modes effect the way routes are used and managed by participants of the network. The specifications give freedom to implementers to choose which mode to use, thus each OS may have chosen a different default, so we select this value as a feature.

MOP	**Description**
0	No downward routes
1	Non-storing mode
2	Storing mode, no multicast
3	Storing mode with multicast

Table 4.3: Modes of operation defined by RPL.

DODAGPreference

This 3-bit value defines the preference value for this DODAG, compared to other DODAGs present in a RPL instance, depicted as a value between 0 (least preferred) and 7 (most preferred). The default value for this field is defined as 0 by the specifications, but implementers are free to use other preference values, and may thus be implementation specific.

DAO

The DAO conveys information about the node attempting to join a RPL DODAG. Its base object, as seen in figure 4.7, has a size of 4 or 20 bytes, depending on the presence of the optional DODAGID field. This control frame supports 5 options: Pad1, PadN, RPL Target, Transit Information and RPL Target Descriptor.

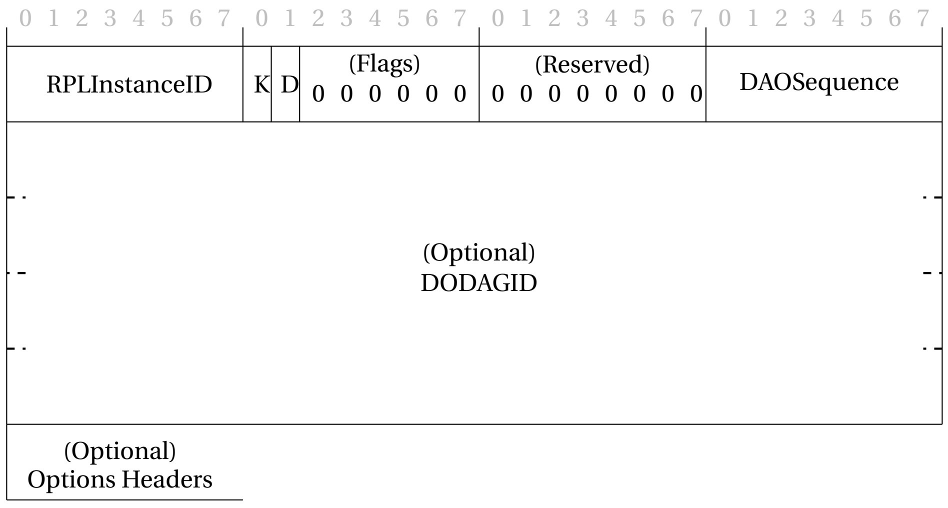


Figure 4.7: The DAO Base Object.

RPLInstanceID

This field is the same as the RPLInstanceID provided in the DIO base object. We use this field as a feature for DAO frames based on the same considerations mentioned earlier for the field in the DIO.

K

This 1-bit flag indicates whether the transmitting node expects to receive a DAO-ACK as a reponse to this frame. Since DAO-ACK frames are optional in the convergence process, the choice to use it is left to implementers, meaning that the value of this field may differ between the selected OSes.

D

This 1-bit flag indicates the presence of the optional DODAGID field within the DAO base object. The DODAGID must be provided if the RPL instance is local, but is not required to be omitted if the instance is global. Due to this ambiguity, this value may differ per implementation.

DAO-ACK

The DAO-ACK acts as a protocol-level acknowledgement for when a node joins a DODAG and mirrors the DAO frame that triggers it. Its base object is depicted in 4.8, and has a size of 4 or 20 bytes, depending on the presence of the optional DODAGID field. This frame does not define any optional header options.

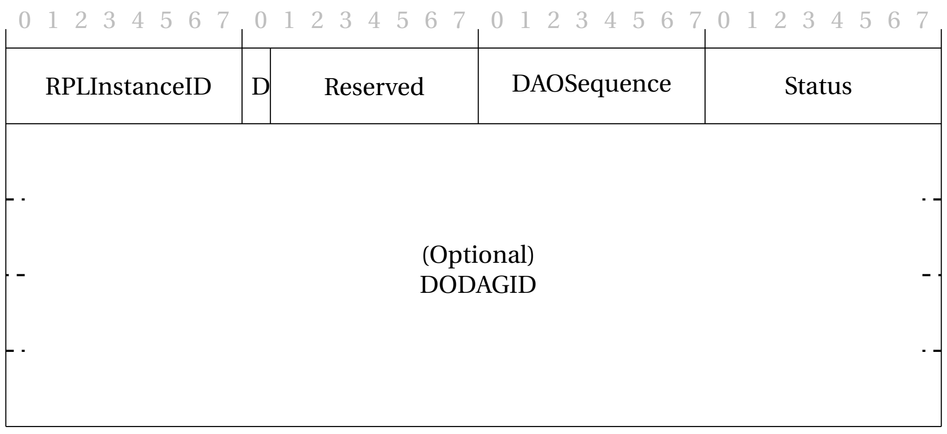


Figure 4.8: The DAO-ACK Base Object.

RPLInstanceID

This field is the same as the RPLInstanceID provided in the DIO base object. We use this field as a feature for DAO-ACK frames based on the same considerations mentioned earlier for the field in the DIO.

D

This field is the same as the D field provided in the DAO base object. We use this field as a feature for DAO-ACK frames based on the same considerations mentioned earlier for the field in the DAO.

Status

The status field is used as a status code indicating the state of acceptance of the DAO sender as a child, as depicted in table 4.4. More detailed status codes are left out of the scope of the original specifications, and which specific value should be used in case a child is not fully accepted is not explicitly defined. For this reason, this value may differ between implementations.

Status	**Description**
0	Accepted
1-127	Temporarily accepted, recipient is suggested to find another parent node
127-255	Rejected

Table 4.4: Status codes defined by RPL, used to indicated the status of a node attempting to join another as a child.

4.3.2. Options Headers

Per optional options header that is supported by that frame type, we introduce a boolean feature indicating the header's presence or absence. Some options may occur multiple times, but we consider an options header present if it occurs at least once. Additionally, we consider the contents of these optional headers as possible features, if they have any.

Pad1

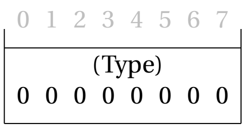


Figure 4.9: Pad1 option header.

Fixed-length 1 byte padding, depicted in figure 4.9. Contains no fields to be used as a feature.

PadN

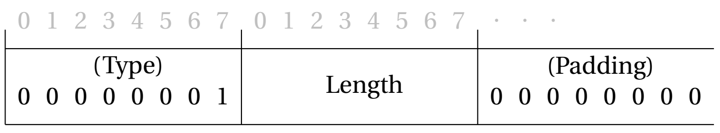


Figure 4.10: PadN option header.

Variable-length padding of N bytes, depicted in figure 4.10.

Length
The length field indicates the number of zeroes appended to the base options object as padding, allowed to have a value between 0 and 5.

DAG Metric Container

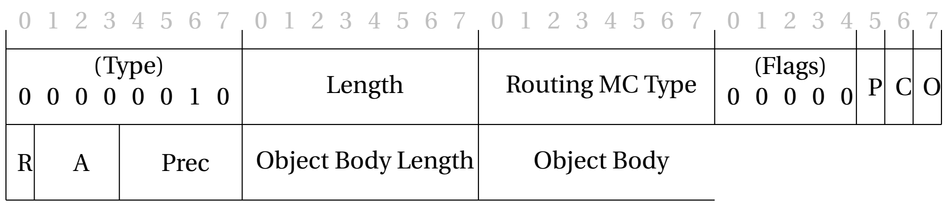


Figure 4.11: DAG Metric Container option header.

This options object provides details on the link metrics used by nodes to decide on the most ideal routes. The metric container may contain multiple metrics, each with its own routing metric type header and object body. Its contents are shown in figure 4.11.

Routing-MC-Type

This field defines the metric used by the node for its routes. The possible values for this field are managed by IANA[5]. Which metric is used depends on the network owners and no default metric is enforced by the specifications, thus each OS may have opted to use a different metric as default.

Prec

This field indicates the precedence of the current metric object over other metrics provided within the same DAG metric container options header, ranging from 0 (highest) to 15 (lowest). Since metric selection is implementation dependent, the precedents assigned to each metric may vary.

Routing Information

The contents of this options header will vary depending on the composition of the network, so none of the fields are usable as features.

DODAG Configuration

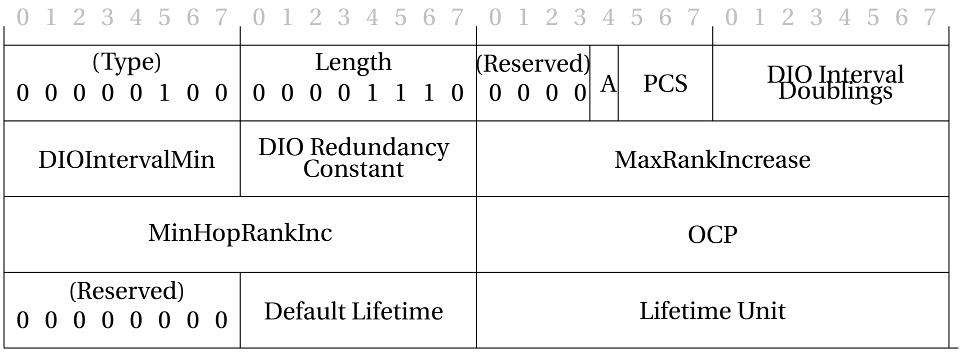


Figure 4.12: DODAG Configuration option header.

DIOIntervalDoublings

The DIOIntervalDoublings field is used to indicate the I_{max} Trickle parameter used by the DODAG. While the specifications propose a default value of 20 for this field, corresponding to a maximum interval size of 2.3 hours, implementers are given the freedom to change this value to fit their needs.

DIOIntervalMin

The DIOIntervalMin field is used to indicate the I_{min} Trickle parameter used by the DODAG, calculated as $2^{DIOIntervalMin}$ ms. While the specifications propose a default value of 3 for this field, corresponding to a minimum interval size of 8 ms, implementers are given the freedom to change this value to fit their needs.

DIORedundancyConstant

The DIOIntervalMin field is used to indicate the k Trickle parameter used by the DODAG. While the specifications propose a default value of 10 for this field, implementers are given the freedom to change this value to fit their needs.

MaxRankInc

This field is used to indicate a DODAG's allowed maximum increase in node rank during local repairs, in order to limit volatility in the topology which could reduce the network's performance. No default value is provided for this parameter, and may thus differ per implementation.

[5]RPL Routing Metric types - https://www.iana.org/assignments/rpl-routing-metric-constraint/rpl-routing-metric-constraint.xhtml#rmc-type

OCP

This field indicates the Objective Function (OF) used by the nodes in this DODAG to calculate node ranks. An OF describes how route metrics are converted into rank values that are used for ideal route selection. Possible values for this field are managed by IANA[6] and no default value is provided by the specifications.

MinHopRankInc

This field describes the granularity of route distance calculated using the DAGs OF and the relation between that distance value and a node's determined rank, where the rank is calculated as $Rank = floor(distance/MinHopRankInc)$. In essence, the MinHopRankInc determines the position of the decimal point of a calculated route distance value, in which the rank is determined by the integer part of that value. This allows an OF to provide higher precision distance values from which an ideal route selection can be performed.

The default value for this field is defined as 256, which results in an 8-bit integer part. This limits the maximum number of hops in a DODAG to 2^8, thus implementers are required to use a different value if larger DODAGs are needed.

Default Lifetime & Lifetime Unit

These fields define the default lifetime in seconds of routes created within a DODAG after which a route is no longer considered valid and should be renewed, calculated as $defaultlifetime * lifetimeunit$ s. No default values are provided in the specifications for these fields.

RPL Target

The contents of this options header will vary depending on the composition of the network, so none of the fields are usable as features.

Transit Information

The contents of this options header will vary depending on the composition of the network, so none of the fields are usable as features.

Solicited Information

Contains conditions for a DIS recipient to respond with a DIO. Is used to perform neighbour node probing with smart filtering in order to limit the number of responses that are less useful.

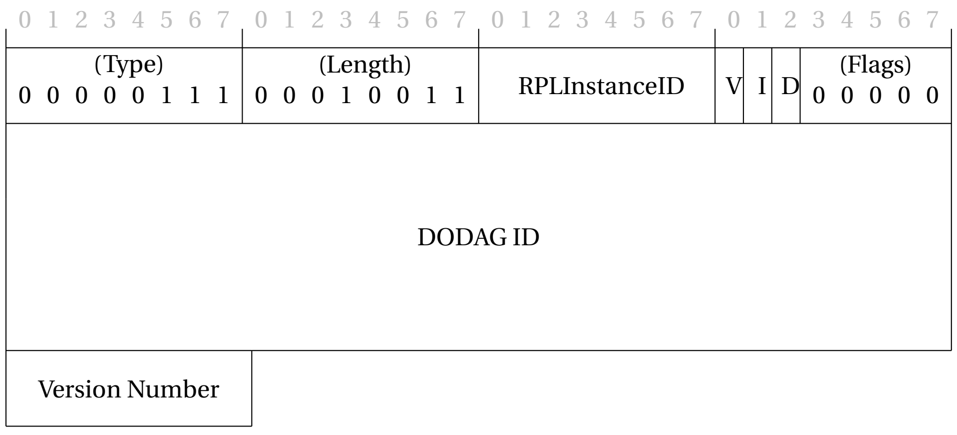


Figure 4.13: Solicited Information option header.

V

If this flag is set, only nodes with a DODAG version number that match the Version Number value provided in the Solicited Information option header should respond to this DIS frame.

[6]RPL OCPs - https://www.iana.org/assignments/rpl/rpl.xhtml#ocp

I

If this flag is set, only nodes with a RPLInstanceID that match the RPLInstanceID value provided in the Solicited Information option header should respond to this DIS frame.

D

If this flag is set, only nodes with a DODAG ID that match the DODAG ID value provided in the Solicited Information option header should respond to this DIS frame.

Prefix Information

The prefix information option lets nodes inform neighbours of IPv6 prefixes reachable through that node, allowing for stateless address configuration.

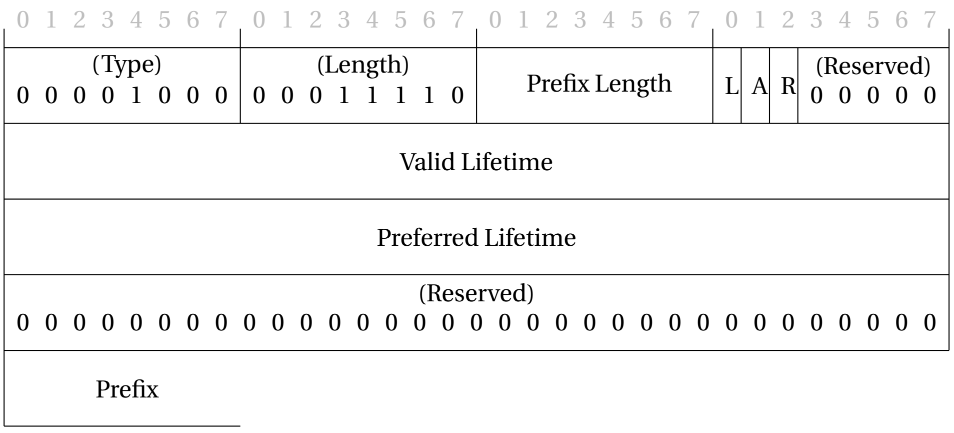


Figure 4.14: Prefix Information option header.

Valid Lifetime

This field defines the default lifetime for prefixes assigned through this option header to be considered valid. No default value is provided for this field and thus may differ per implementation.

Preferred Lifetime

This field defines the lifetime for prefixes assigned through this option header to preferred for use. No default value is provided for this field and thus may differ per implementation.

RPL Target Descriptor

The contents of this options header will vary depending on the composition of the network, so none of the fields are usable as features.

The Packet Information Header

This option is used in data frames for loop avoidance purposes, as described earlier in chapter 2. Its format is depicted in figure 4.15.

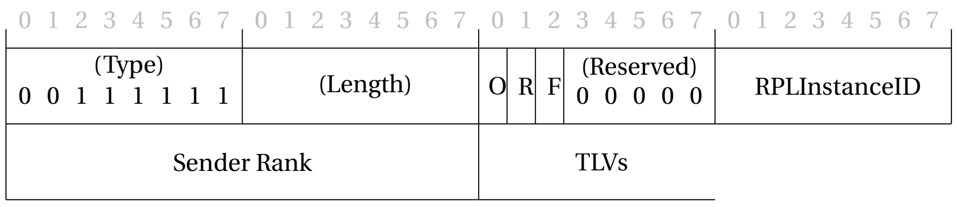


Figure 4.15: RPL Packet Information header.

RPLInstanceID
This field is the same as the RPLInstanceID provided in the DIO base object. We use this field as a feature for DAO frames based on the same considerations mentioned earlier for the field in the DIO.

4.3.3. Other Features

For Sample, we extract retransmission counts, retransmission intervals, usage of certain features throughout the network lifetime, how those features are used and how the field usage evolves through time. Stimulus-Response looks at similar features, but specifically in response to received packets.

In order to avoid over-fitting our classifier to the specific composition of nodes in our simulations,

DIS Retransmission Count
Average number of retransmissions of DIS frames.

Recognising retransmissions is difficult due to no mechanism being available in the frame header for this purpose, so we rely on heuristics to extract this information. A frame is considered to be retransmitted if any of the following requirements are met.

- Last transmitted frame was of the same control frame type
- ICMPv6 checksum matches last transmitted frame
- Last transmitted frame occurred less than 1 second ago
- Time between last and current transmission is within a factor of 5 of the average interval observed until this point

DIS Retransmission Interval
Average interval between retransmissions of DIS frames. The same heuristic method is used as for the DIS retransmission count.

DIO Retransmission Count
Average number of retransmissions of DIO frames. The same heuristic method is used as for the DIS retransmission count.

DIO Retransmission Interval
Average interval between retransmissions of DIO frames. The same heuristic method is used as for the DIS retransmission count.

DAO Retransmission Count
Average number of retransmissions of DAO frames. The same heuristic method is used as for the DIS retransmission count.

DAO Retransmission Interval
Average interval between retransmissions of DAO frames. The same heuristic method is used as for the DIS retransmission count.

DAO-ACK Retransmission Count
Average number of retransmissions of DAO-ACK frames. The same heuristic method is used as for the DIS retransmission count.

DAO-ACK Retransmission Interval
Average interval between retransmissions of DAO-ACK frames. The same heuristic method is used as for the DIS retransmission count.

DAO Average Delay
Delay between a DIO and its DAO response.

Calculated by first locating DAOs then crawling back to the first non-retransmitted DIO sent by the DAO recipient. The same heuristic method for retransmission detection is used as for the DIS retransmission count.

DAO-ACK Average Delay
Delay between a DAO and its DAOACK response.

Calculated by first locating DAO-ACKs then crawling back to first non-retransmitted DAO sent by the DAO-ACK recipient. The same heuristic method for retransmission detection is used as for the DIS retransmission count. Additionally, the DAO-ACK echoes the DAOsequence field of the matching DAO.

DAO-ACK Requested
Whether the K flag is present at least once in a DAO frame. This indicates whether the DAO transmitter desires DAO-ACKs to be used.

DAO-ACK Used
Whether a DAO-ACK frame is used at least once. This indicates that DAO recipients support and use DAO-ACKs.

DAO-ACK Request Followed
Whether a DAO-ACK is sent at least once when one is requested through the K flag in the received DAO. This feature connects the DAO-ACK Requested and DAO-ACK Used features for the DAO recipient. It is needed as a separate feature, since DAO-ACK Requested is a feature defining the behaviour of the DAO transmitter, rather than the recipient.

DIO DODAG Configuration Option Used
Whether a DODAG Configuration Option header is used at least once in a DIO frame. While this options header is optional, it contains critical information for the configuration of the DODAG. For this reason, we are interested in knowing whether any OS elides it.

DIO DODAG Configuration Option Always Present
Whether a DODAG Configuration Option header is present in all DIO frames transmitted by that node. While the information in this options header may be important for configuring the DODAG, once the DODAG is joined and correctly configured, there is no need to keep transmitting it with every DIO frame. This feature depicts whether any OS makes this consideration.

DIO DTSN Increment Type
How DTSN evolves after every transmitted DIO. When this sequence number is incremented, all children receiving that DIO must respond with a new DAO. This can be used for local maintenance and route refreshing. How this value should be incremented is not specified, and thus increments may happen in different intervals. We define the following increment methods:

- 0) Static - (1, 1, 1, ...)
- 1) Monotonic - (1, 2, 3, 4, 5, ...)
- 2) Linear - (1, 3, 6, 10, ...)
- 3) Random - (1, 15, 278, ...)

Intermittently incrementing (1, 1, 2, 2, 2, 3, ...) is not easily identifiable due to retransmissions. Here, instead of applying a heuristic method, we remove duplicate DTSN values before assessing.

DAO Sequence Follows DTSN
Whether the DAO Sequence evolves as a response to DIO DTSN values. According to the specifications, if a node transmits a DAO with new information, it should increment its DAO sequence number. It may, however, also increment this value even if the DAO contains the same information. This feature tracks whether the implementation also chooses to increment its DAO sequence when it receives an incremented DTSN.

Difference Between DAO Sequence and DTSN
Difference between the DAO Sequence and DIO DTSN values. Depicts the offset between the received DTSN and transmitted DAO sequence, and the ratio of DAO sequence incrementing.

Directed DIO Present
Whether a directed DIO frame is used at least once. Normally during convergence, DIO control frames are transmitted to a multicast address, but the specifications also allow for unicast DIOs. These can be used, for example, when a DIO is specifically requested by another node, or as a probing mechanism by the transmitting node.

Directed DIO Transmission Interval
Average interval between directed DIO frames. Here we disregard to possibility of a frame to be retransmitted and consider the average interval between all DIO frames with a unicast target address.

4.3.4. Summary of Features

Figure 4.5 shows a summary of all features mentioned above, separated by feature type, which will be used for this research.

Singleton DIS Features	Singleton DIO Features	Singleton DAO Features	Singleton DAO-ACK Features	Singleton Data Features	Sample / Stimulus-Response Features
Option 0 present	RPLInstanceID	RPLInstanceID	RPLInstanceID	HBH option present	DIS average retransmission count
Option 1 present	Grounded	k	d	HBH option option type	DIS average retransmission interval
Option 7 present	MOP	d	status	RPLInstanceID	DIO average retransmission count
Option 1 length	DODAGPreference	opt0present			DIO average retransmission interval
Option 7 V	Option 0 present	opt1present			DAO average retransmission count
Option 7 I	Option 1 present	opt5present			DAO average retransmission interval
Option 7 D	Option 2 present	opt6present			DAOACK average retransmission count
	Option 3 present	opt9present			DAOACK average retransmission interval
	Option 4 present	opt1len			DAO average delay
	Option 8 present	opt9descriptor			DAOACK average delay
	Option 1 length				DAOACK requested
	Option 2 metric 1 present				DAOACK used
	Option 2 metric 2 present				DAOACK request followed
	Option 2 metric 3 present				DIO DODAG configuration option used
	Option 2 metric 4 present				DIO DODAG configuration option always present
	Option 2 metric 5 present				DIO DTSN increment type
	Option 2 metric 6 present				DAO sequence follows DTSN
	Option 2 metric 7 present				Difference between DAO sequence and DTSN
	Option 2 metric 8 present				Directed DIO present
	Option 2 unknown metric present				Directed DIO average interval
	Option 2 metric 1 precedence				
	Option 2 metric 2 precedence				
	Option 2 metric 3 precedence				
	Option 2 metric 4 precedence				
	Option 2 metric 5 precedence				
	Option 2 metric 6 precedence				
	Option 2 metric 7 precedence				
	Option 2 metric 8 precedence				
	Option 4 DIOIntervalDoublings				
	Option 4 DIOIntervalMin				
	Option 4 DIORedundancyConstant				
	Option 4 MaxRankInc				
	Option 4 MinHopRankInc				
	Option 4 OCP				
	Option 4 default lifetime				
	Option 4 lifetime unit				
	Option 8 valid lifetime				
	Option 8 preferred lifetime				

Table 4.5: All features selected, separated by feature type.

5 Experiment Design

In order to answer the stated research questions, we design an experiment and consequent analysis to prove the ability of OS fingerprinting using RPL. In this experiment we simulate a large number RPL networks per OS and collect all network traffic transmitted between the nodes during the simulation. By analysing the obtained network traffic, we attempt to create fingerprints that uniquely identify each OS, with which OS identification can be performed. Here we give a clear definition of the project scope, outlining any assumptions made prior to experimentation, followed by a detailed description of the experiment.

5.1. Security Assumptions

This experiment aims to display the feasibility of OS fingerprinting with RPL traffic as a concept. Because of this, we have chosen to keep the experiment environment as simple as possible and have made several assumptions regarding the network creation and data collection.

Since the protocol we are interested in only operates within the local PAN and is not transmitted outside the WSN, data capturing has to happen within radio range of the devices under investigation. In practice, this could be achieved in several ways, as outlined in chapter 2. These methods would potentially have limited visibility of the entire network, depending on the transmission ranges of the nodes being observed, or would require a more sophisticated distributed data collection approach to observe all available nodes in the network. However, for this study we assume that the WSN under investigation is small enough to be fully visible by our sniffer.

For the selection of OSes to investigate in this research, we choose to only consider open-source operating systems. Such OSes are more easily accessible and will have their source code available for deeper inspection, if necessary. We assume that no alterations are made to the core code that makes the protocol work. The OSes must be put in use as their developers intended and occurrences of such alterations in the wild are assumed to be rare enough to be left out of scope. OSes do generally allow adjusting certain parameters outside of the source code, either via user-level API calls, or parameters provided during compile time.

Some parameters in RPL need to be chosen correctly, in order to obtain the best network performance results. This means that not all combinations of parameters are as likely to be observed in the wild. Default parameters of the OSes are generally already optimised, so changing away from the default may be undesirable for the users, decreasing the likelihood of observing different parameter combinations in the wild. This does not, however, forbid users of the OSes to change these parameter. Because of this, we assume that parameters may be changed from the default parameters set by the OSes, and investigate how these changes will affect the classification efforts. Any parameters relating to protocols on other layers are left as is, except for the 802.15.4 radio channel, which we arbitrarily set to channel 25. The knowledge of which channel is utilised is necessary for hardware-based simulations, as the sniffer needs to be tuned to the same channel as the nodes to be observed.

Another simplifying assumption we make is that we keep our networks homogenous, so all nodes in the DODAG are from the same OS and have the same parameters. Since some control frame value are decided by the DODAG root, and are copied and propagated by its children, it may be difficult to ascertain the true parameters of children nodes before joining a DODAG. While it would be interesting to observe how network heterogeneity might affect our fingerprinting effort, this is left out of scope due to time constraints.

Finally, we assume that no encryption methods are used that hide the contents of RPL control frames, since this provides us the most information to investigate.

5.2. Experimental Setup

After having created and tested the firmware, we design an experiment to generate RPL traffic for fingerprinting. We simulate RPL networks for each OS, using the corresponding environment, and capture all generated traffic into PCAP files following the methods described earlier in section 4.2. The simulated networks consist of 2 nodes, a root node and a router node, placed at a distance within each other's radio range. The root node acts only as a PAN coordinator within the local network and is not connected to a secondary external network.

At the start of the simulation, both nodes are booted at the same time. The router node connects to the root as a child node following the RPL convergence process. Once the initial convergence is complete and a DODAG has been formed, the DODAG is kept maintained via the default RPL maintenance methods. Previous efforts have shown that small RPL networks ($\leq$ 40 nodes) complete the convergence process in under 100 seconds [45, 47], so we arbitrarily choose the duration of each simulation to be 150 seconds. This duration is long enough to ensure our networks are fully converged and additional time is given to observe the behaviour of nodes after this process. We test for two specific scenarios.

Scenario 1 - Control Frames
In the first scenario, only the DODAG is set up and maintained. This will result in traffic consisting only of RPL control frames, allowing us to observe the behaviour of RPL without it being influenced by the transmission of other packets. We achieve this behaviour by triggering the RPL convergence process once the nodes are booted, after which all control traffic is automatically generated and handled by the OS kernels.

Scenario 2 - Data traffic
Scenario 2 starts similarly to the previous. Following the initial convergence after which a DODAG is formed, the nodes maintain their topology via de default RPL maintenance process. Once the DODAG is created, the nodes now also send UDP data to each other in periodic intervals. This allows us to both investigate the effects of data layer traffic on RPL control traffic, and also observe the usage of RPL Packet Information objects within these data packets.

In order to facilitate the sending of UDP data, we create an UDP server on the root node and an UDP client on the router. The client stores a counter that will be converted into its string representation and transmitted to the server. The counter is incremented after every transmission. The first data frame is transmitted 5 seconds after convergence. The server responds to the reception of the data by sending the string *"OK"* to the client, after which the client schedules the next data frame to be transmitted randomly within an interval of 10 seconds. This behaviour is illustrated in figure 5.1 below.

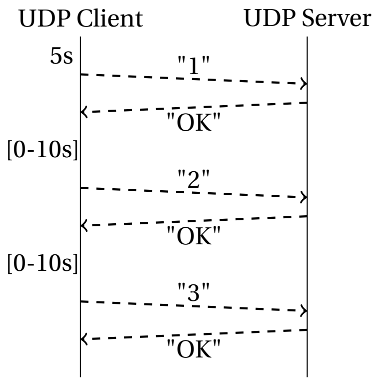


Figure 5.1: Visualisation of the UDP data transmission timeline.

6 Experimentation

This chapter describes the practical steps taken during the execution of the experiment, outlining the process of setting up the simulation environments, building the necessary software, running the simulations, and processing the obtained data. We conclude with the technical limitations observed throughout the design and execution process.

6.1. Building the firmware

Before we can perform our experiments, we have to program our nodes with an application that will generate the desired network traffic. This section describes the process of setting up the necessary toolchains, design and implementation of the code, and how to program and run the nodes.

6.1.1. Setting up Toolchains

To ease the setting up of toolchains of each OS, we utilise container environments provided by Docker. Containerisation allows for the creation of highly replicable and portable environments with little to no manual input required by the user. Both RIOT and Contiki-NG have official Docker containers for their development environments that we can use. The compilation toolchains required by Contiki OS are also present for Contiki-NG, meaning that we can use the Contiki-NG container for both OSes. The only selected OS without an official container is TinyOS. For this OS, we create our own by following the tutorial provided in the TinyOS code repository[1].

The usage of containers does mean that all compilation tasks should be performed within these containers. To simplify the editing and compilation process, we make use of the Shared Folder functionality in Docker, allowing us to edit the code locally and our edits to directly be visible to the containerised environment.

6.1.2. Implementation of Desired Behaviour

In order to obtained the desired behaviour from the OSes for our experiments, we need to create some applications to run on our OSes. These applications will be responsible for setting up the RPL networks and communicating application-layer data when required. Luckily, most of the behaviour is already implemented in parts in example code provided by the OSes, which we use as a base from which we build the rest of our application. We describe the changes required to the example code and provide diagrams for each OS depicting the application behaviour.

Contiki OS

As a start-off point for our implementation we use the RPL-based example code provided in the *examples/ipv6/rpl-udp* folder. This example implements a UDP client and server similar to our scenario 2 described in chapter 5, where a RPL network is formed with the UDP server as root, with the UDP client sending periodic messages to the server, and the server responding with a simple reply string.

[1] TinyOS setting up Debian development - https://github.com/tinyos/tinyos-main/blob/master/doc/00c_Setting_Up_Debian_Development

Contiki OS is a multi-threaded event-driven operating that handles most of the core protocol behaviour within the kernel. Thanks to this, we only need to write a custom thread within which we setup our UDP server and client, and define the UDP message scheduling behaviour. High-level diagrams for the behaviour of the UDP client and server threads are given in figures 6.1 and 6.2 below. The reception and transmission of UDP messages are triggered by timer and kernel-level events, as indicated by the red arrows. The handling and transmission of RPL control frames is not shown, as this is performed opaquely by the underlying OS kernel.

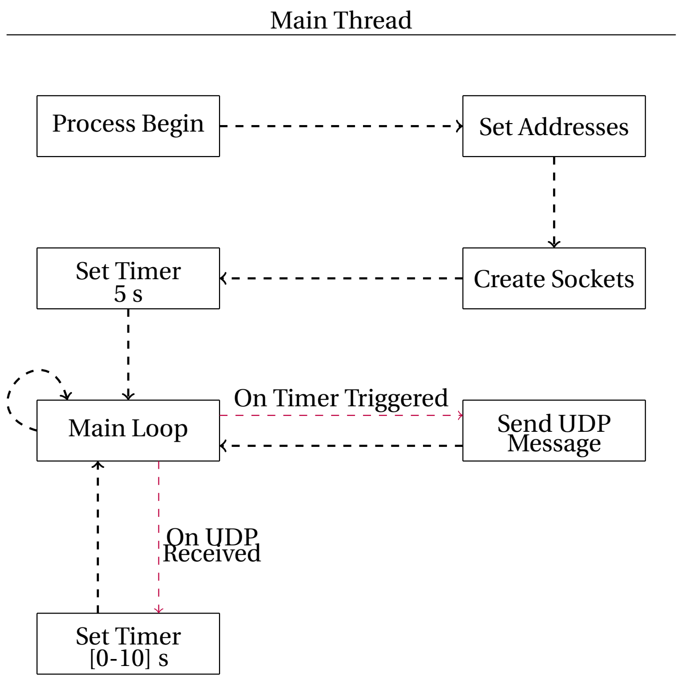


Figure 6.1: Diagram of the Contiki OS UDP Client application behaviour.

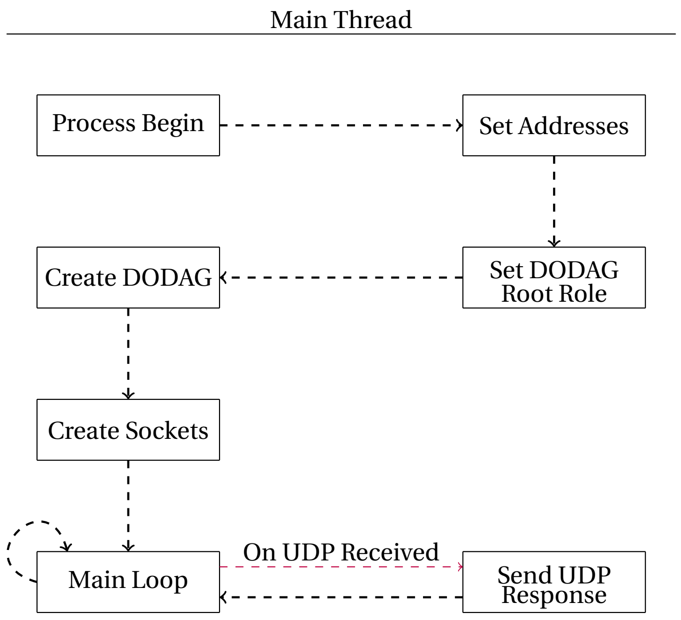


Figure 6.2: Diagram of the Contiki OS UDP Server application behaviour.

To simplify the application, IPv6 addresses of the client and server statically assigned. The server creates a UDP socket on port 5678, and the client opens a socket on port 8765 to be able to receive responses from the server.

The example also contains some functionality that is not needed for our experiments, such as logging of the nodes' energy consumption using Powertrace in the client, and allowing for the triggering of global DODAG repairs by pressing a hardware button in the server. We remove this functionality in order to reduce the nodes' resource usage.

To implement scenario 1, we simply remove the lines of codes responsible for receiving and transmitting UDP packets, and leave the rest as-is.

Contiki-NG

Being a continuation of Contiki OS, Contiki-NG relies on the same programming paradigms as its predecessor, resulting in diagrams similar to those for Contiki OS, as depicted in figures 6.3 and 6.4. The only difference is that Contiki-NG provides easy methods to obtain the IP address of the DODAG root, so we do not have to statically assign any addresses. Once again, the RPL control frame scheduling and handling is performed internally by the OS kernel.

For this version we edit the RPL-based example provided in the *examples/rpl-udp* folder, containing a UDP client and a UDP server implementation with behaviour similar to scenario 2. Once the DODAG convergence is initiated by setting the server's role to that of a root node, sockets are opened on ports 5678 and 8765 by the server and client respectively, and the client starts sending periodic messages to the address of the DODAG root. The server responds to the address obtained from the received packet.

To implement scenario 1, we simply remove the lines of codes responsible for receiving and transmitting UDP packets, and leave the rest as-is.

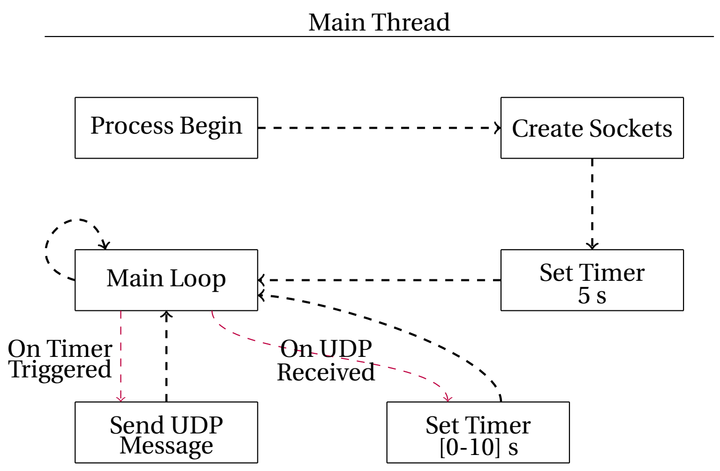


Figure 6.3: Diagram of the Contiki-NG UDP Client application behaviour.

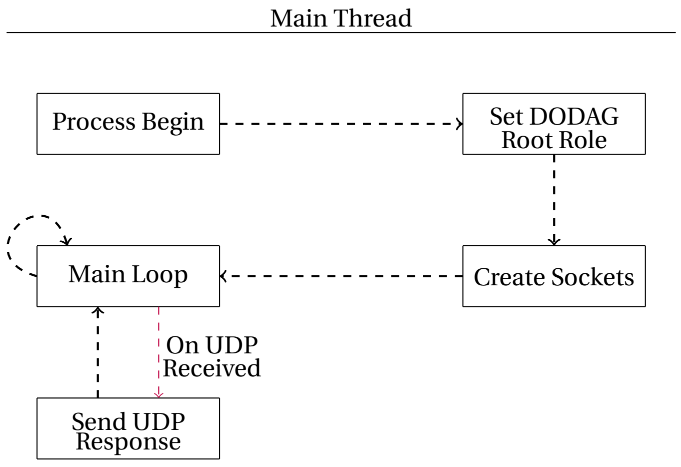


Figure 6.4: Diagram of the Contiki-NG UDP Server application behaviour.

RIOT

For RIOT, we start off with the networking example provided in *examples/gnrc_networking*. This example imports all necessary networking-related libraries and boots into a shell that allows the control and management of RPL networks, which it requires to be performed manually through shell commands[2].

For debugging purposes, we keep the shell component of the example application. However, in order to automate the shell interaction and setup the RPL network immediately after boot in the root node, we extend the example utilising the GNRC networking and RPL APIs. First we add a static IPv6 address to the root node on the 802.15.4 network interface by calling
gnrc_netif_ipv6_addr_add(<interface>, <address>, 128, 0). Next we create a RPL DODAG with a DODAG ID equal to that address with *gnrc_rpl_root_init(0, <address>, true, true)*. The node will now create a RPL DODAG and automatically generate control messages for other nodes to be able to join the DODAG. Once the other node receives a DIO from this node, it will automatically join the RPL network without any additional code or manually shell interaction required. These changes will allow for the creation of a simulation following the first scenario.

In order to implement the second scenario, we extend the application with the necessary UDP client and server functionality. For the UDP client, we create a thread that handles the UDP packet transmission and reply reception, and hard-code the UDP server IPv6 address into it for the sake of simplification. In RIOT, this address is automatically derived from the 48-bit hardware address assigned to the used hardware, which we obtain by manually running the default networking example provided by RIOT and extracting the IPv6 address from the captured network traffic. The server functionality is implemented by creating a similar thread that creates a socket on the desired port and replies on message reception from the client.

This results in the diagrams shown in 6.5 and 6.6.

[2]Tutorial RPL in RIOT - https://github.com/RIOT-OS/RIOT/wiki/Tutorial:-RIOT-and-Multi-Hop-Routing-with-RPL#initializing-rpl

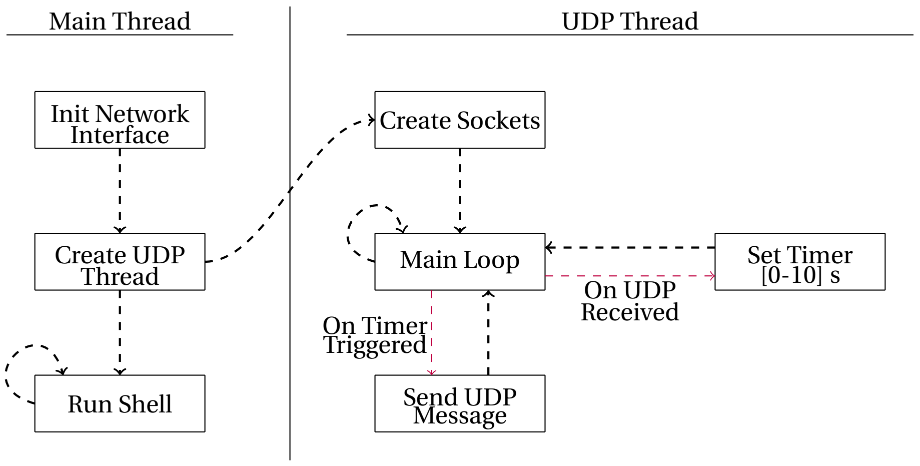


Figure 6.5: Diagram of the RIOT UDP Client application behaviour.

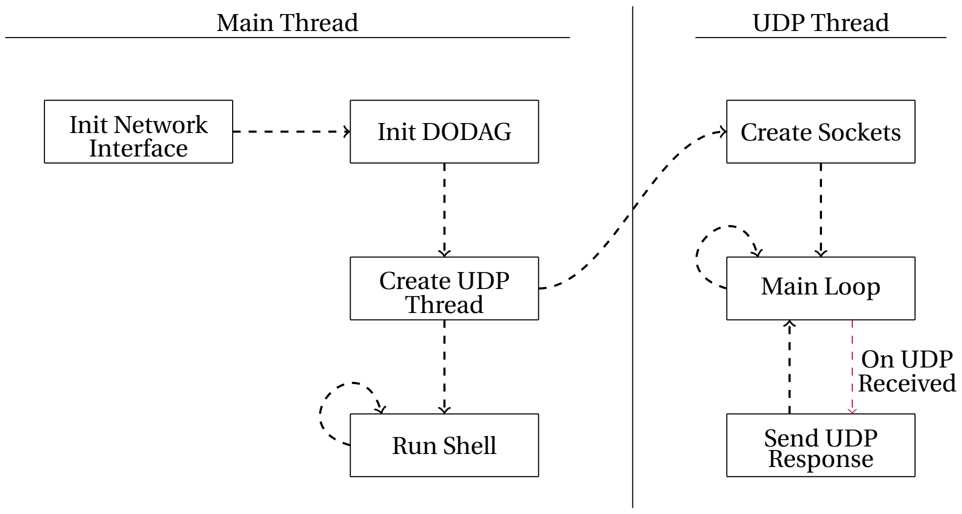


Figure 6.6: Diagram of the RIOT UDP Server application behaviour.

TinyOS

TinyOS is a fully event-driven OS, which sets it slightly apart from the other OSes described above. Instead of allowing the creation of worker threads that are then further scheduled by the kernel, TinyOS expects all tasks to be performed within event callbacks. The *apps/UDPEcho* folder in TinyOS contains an example implementation of a RPL router running a UDP echo server that we can use as a base for our application. We remove the LED control and statistics monitoring code from the example and implement the UDP client and server behaviour as according to scenario 2 within network event callbacks. For the root node, we also call the *setRoot* function of the *RPLRoutingC* interface, in order to trigger the convergence process with that node as the DODAG root. The diagrams depicting this behaviour can be found in figures 6.7 and 6.8 below.

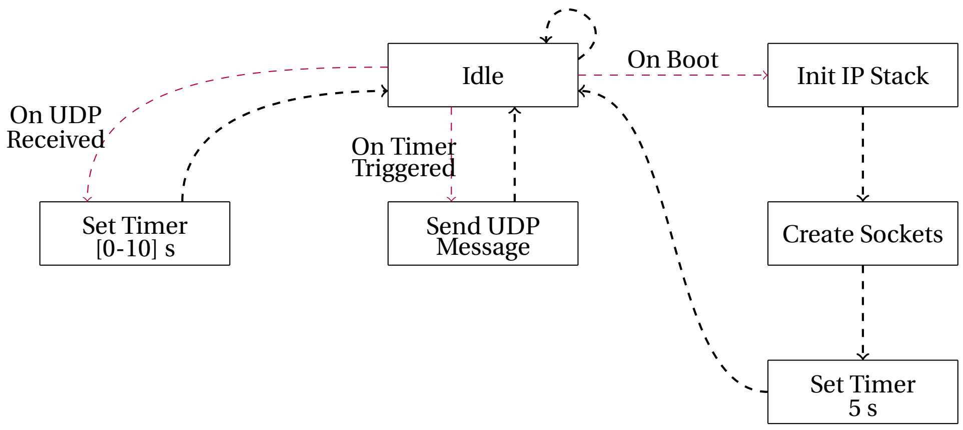


Figure 6.7: Diagram of the TinyOS UDP Client application behaviour.

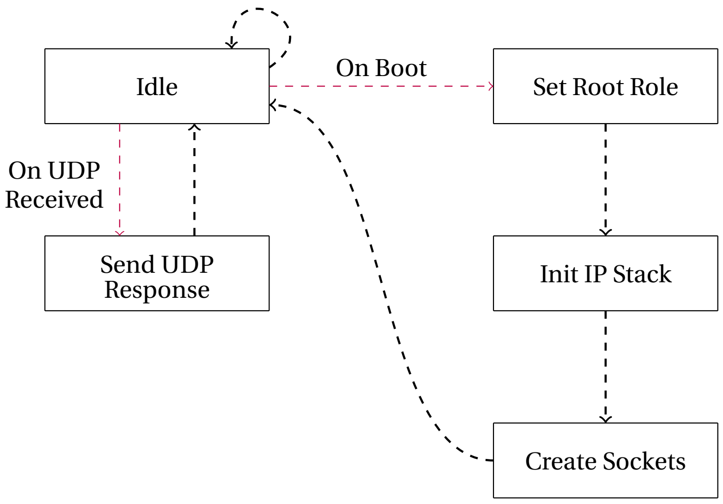


Figure 6.8: Diagram of the TinyOS UDP Server application behaviour.

Sample Composition and Generation

We create around 80 samples in total per OS, resulting in over 300 samples for the entire dataset. Of these, 60 include data traffic, while the rest only consists of RPL control traffic. For each OS, we have 2 samples with default parameters, one with and one without data traffic, while for the rest the RPL parameters are randomised.

We manually compile versions of the OSes with default parameters. For versions with custom parameters, however, since there is a wide range of OS parameters that can be assigned, we create a test generation pipeline that automatically generate firmware for the selected OSes with randomised sets of parameters. For this, we investigate the code-bases of the selected OSes and extract all editable parameters related to RPL, including the range of values each parameter can be set to. The list of editable parameters for each OS can be found in appendix A.

Using this information, we generate samples with random parameter configurations. Per configuration, we assign each editable parameter a 50% chance to be kept at its default value. For the parameters selected to be randomised, we uniformly randomly select a value from within its range of possibilities. The changed parameters are then converted into a compilation command string with all necessary flags set, depending on the OS.

Each OS has its own method of allowing compile-time parameter selection, as shown below, where <*K*> stands for the name of the parameter and <*V*> for the value assigned to that parameter. Some options can also be defined without a value, in the form of a flag <*F*>.

Contiki OS and Contiki-NG
$DEFINES = " <K>=<V>, \quad <K>=<V>, \quad <F>, \quad <F>, \quad ..."$

RIOT
$DEFINES = " -DCONFIG\_<K> \quad <V> \quad -DCONFIG\_<K> \quad <V> \quad ..."$

TinyOS
$PFLAGS = " -D<K>=<V> \quad -D<K>=<V> \quad ..."$

While TinyOS does support assigning parameters at compile time through the $PFLAGS$ argument, doing so will overwrite any default compiler parameters defined in the project Makefiles, which consequently will cause the compilation to fail unless all necessary default parameters are also passed through the $PFLAGS$ argument. To avoid this issue, we define our own custom parameters corresponding to the editable parameters provided by TinyOS, which we pass to the build through the $CFLAGS$ argument. In the application Makefile, we check for the presence of these new custom flags, and if present, apply their values to the original editable parameters and append them to the $PFLAGS$ variable in that file.

Additionally, we define a custom parameter $DATA\_TRAFFIC\_UDP$ for all OSes, which we use to indicate whether we are compiling for scenario 1 or 2. If this parameter is defined, we include the UDP-related code into the build, corresponding to scenario 2. This way, we can limit code duplication and implement both scenarios within the same files.

We also alter Contiki-NG's $RPL\_CONF\_SUPPORTED\_OFS$ parameter to be more console-friendly. Originally, the value assigned to this should be a C++ list initialiser containing pointers to the functions implementing the OF's behaviour ($\{\&function1,\ \&function2, ...\}$). Instead, we define the parameters $RPL\_CONF\_OF\_OF0$, $RPL\_CONF\_OF\_MRHOF$ and $RPL\_CONF\_OF\_ALL$, corresponding to $RPL\_CONF\_SUPPORTED\_OFS = \{\&rpl\_of0,\}$, $RPL\_CONF\_SUPPORTED\_OFS = \{\&rpl\_mrhof\}$ and $RPL\_CONF\_SUPPORTED\_OFS = \{\&rpl\_of0,\ \&rpl\_mrhof\}$ respectively. This allows us to more easily set the necessary value for this parameter at compile-time.

One important component of parameter randomisation is that some parameters require specific values to be able to work correctly. Because of this, we will check the obtained results manually to ensure that the samples still work as expected. A configuration is considered broken if any of the following are observed in its output within the duration of the simulation.

- No DIO control frames are transmitted by any of the nodes.
- If no DAO control frames are transmitted by the child nodes, while MOP is set to any value other than 0.
- If no data frames are transmitted by any of the nodes in the samples where data traffic is expected.

Since we keep our simulations limited to 150 seconds, we can also encounter these conditions with parameters that would normally be acceptable. One such example is the DAO delay parameter allowed to be changed by RIOT, which delays the transmission of a DAO control frame after the reception of a DIO. If this value is set high enough to exceed the duration of our experiments, no DAO frame will be observed within this time frame. For this reason, we set some bounds to the range of values certain parameters can have, in order to maximise the number of usable samples we generate. The specific parameters for which this is the case can also be found in appendix A.

6.2. Environment Setup

We ready the selected experiment environments in order to run our simulations. Both Cooja and Renode support simulation definition files that can be used to quickly load in and start running a simulation. For Cooja, this is an XML file with a "*.csc*" extension, and contains information regarding the number and positioning of nodes, the firmware they use, tests to run during the simulation, and general configuration of Cooja plug-ins activated when the file is loaded in. This file can be easily created by setting up the simulation manually through the GUI and using the *save simulation* option in the GUI menu.

Through the GUI, we create a sample simulation with two Tmote Sky nodes positioned within each other's radio range. We enable the Cooja scripting window, and in it, we create a script that stops the simulation after 150 seconds. For network traffic capture, we find that the built-in radio logging plug-in does not automatically start capturing the network traffic once the simulation starts. For this reason, we install the *radiologger-headless*[3] plug-in for Cooja, which automatically start the network capture, even when the simulation is run without a GUI.

Once this base simulation definition file has been created, we can create copies of this file and replace specific sections to obtain simulation files for all firmware samples we have generated. Specifically, we require changes to three locations in the file. The first two are paths to the firmware files of the root and router nodes, found within the */simconf/simulation/motetype/firmware* XML tags. The other change is required in the configuration of the *radiologger-headless* plug-in, within the *plugin_config/pcap_file* XML tag, which defines the path where the network capture file is saved to.

Finally, we find that Cooja requires the binaries to be executed to have a specific extension when running, where the file extensions corresponds with the platform to emulate. Compiling Contiki OS and Contiki-NG automatically generates the files with the necessary extension, but the same is not the case for TinyOS. However, this can easily be fixed by renaming the TinyOS compilation output file into the corresponding file extension [67]. For the Tmote Sky platform which we are emulating for TinyOS, the required file extension is *.sky*.

Renode provides test automation through integration with Robot Framework[4]. Several example test files are provided, of which we use the robot file *tests/platforms/CC2538/cc2538_rpl-udp.robot* as a base for our own simulation definition. This example test creates a RPL network of 3 nodes, and tests whether they can send and receive UDP packets to each other. We alter this file by removing one of the nodes, in order to fit our simulation scenarios, and change the test cases to log all network traffic and terminate after 150 seconds.

The main way of terminating a test within this framework is by waiting for specific node output. A test fails if expected output is not seen within the assigned timeout duration. However, if the test string is mistakenly outputted by any of the nodes during the simulation, tests will terminate too early. Since this emulator does not provide any graphical means of verifying correct execution of our firmware, we can not rely on removing all node output to avoid this issue, since we may need to use this output for debugging purposes. For this reason, we extend Renode's Robot Framework integration with a new command *WaitForTimeout*, which terminates a test once the timeout is reached, without the need for a test string to match against.

For its network traffic logging functionality, Renode makes use of Wireshark to capture traffic to files, but only makes this possible through Wireshark's graphical interface, which requires manual clicks to start and save captures. To be able to automate this process, we extend Renode with a new *TsharkPlugin* plugin. This plug-in makes use of Tshark[5] to directly capture to PCAP files. In the robot file, we enable this new plug-in by executing the *CaptureWirelessTraffic* command we defined for this purpose. This command also accepts a parameter indicating the save location of the created network capture file.

Once again, once this definition file has been created, we create copies for each sample we generated that needs to be run within Renode.

The simulations with real hardware are performed manually. We program the hardware modules using the programmer tool nRF Connect[6], provided by the manufacturer of our hardware modules. After programming both the root and router nodes, we disconnect both modules, connect the sniffer module to the computer, and start a network capture in Wireshark. Then we simultaneously power up the modules running the root and router firmware and wait for 150 seconds, after which we stop the Wireshark capture, and save the obtained capture file. This process is then repeated for all samples generated for the RIOT OS.

[3]Cooja radiologger-headless plug-in - https://github.com/cetic/cooja-radiologger-headless
[4]Robot Framework - https://robotframework.org/
[5]TShark - https://www.wireshark.org/docs/man-pages/tshark.html
[6]nRF Connect for Desktop - https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop

6.3. Data Processing

At the end of the data generation process, we obtain a total of 203 network captures among all four OSes, split as shown in table 6.1. We have chosen to generate smaller set of samples for scenario 2, since there are no editable RPL parameters specifically relating to the contents of data frames, thus there being less variation in the contents of those specific frames in this scenario. In the case of TinyOS, which has the smallest set of samples in total, only 2 editable parameters are provided to the end-user, only allowing for 4 unique configurations of the OS. This makes it unnecessary to generate as many samples as the other OSes, as all variation is already captured by this smaller set. For RIOT, the relatively smaller number of samples is caused by the lack of automation in testing, as we were required to tun these simulations manually with hardware.

	Scenario 1	**Scenario 2**	Total
Contiki OS	61	7	68
Contiki-NG	59	9	68
RIOT	35	9	44
TinyOS	14	9	23
Total	169	34	**203**

Table 6.1: Number of network traces generated, split by scenario and OS.

Next, we extract and investigating the frames present in these generated samples. Table 6.2 shows the number of frames present of each frame type per OS. We observe that our data is unbalanced. Most notable is the disparity between DIO frames transmitted by Contiki OS and the others, averaging a factor of 125 times the number of DIO frames sent by other OSes per sample. Manual observation of these samples shows that Contiki OS likes to retransmit control frames a large number of times in a short amount of time, the reason for which we are unable to explain, so we assume this to be the default behaviour for this OS.

Another discrepancy we find is the large difference in data frames transmitted by TinyOS. While there is an element of randomness to the data frame scheduling by design of the scenario, which would cause the transmission numbers to not align perfectly, the data frame count of TinyOS lies over a factor of 7 higher than the rest. We were unable to determined why this behaviour occurs.

	DIS	**DIO**	**DAO**	**DAO-ACK**	**DATA**
Contiki OS	989	216775	339	48	241
Contiki-NG	423	4037	58	37	194
RIOT	92	959	242	188	271
TinyOS	49	414	345	0	1710

Table 6.2: Number of frames present in the complete dataset, split by frame type.

Traffic data is stored in PCAP files in PCAPNG format, which is the most used format for network traffic with wide support. We have two options for PCAP parsing; PyShark and Scapy. PyShark is a Python wrapper around TShark that uses the Wireshark dissectors to parse PCAPs, which has support for the 802.15.4, 6LoWPAN and RPL protocols. Scapy is a Python library that performs the parsing and header extraction by itself, also supporting all necessary protocols out-of-the-box. Since Pyshark is a wrapper round a native application, it greatly outperforms a pure Python implementation in terms of speed. Wireshark dissectors are also more mature, with wider protocol support. Pyshark is, however, more limited in the number of header values it provides to the user for further inspection. Since the actual parsing is being performed in an external application, we are also not able to access the missing data directly or easily extend the capabilities of the dissectors. For this reason, we choose to use Scapy for the parsing and feature extraction tasks.

When using Scapy, we find that it lacks several functionalities that we require to fully analyse all collected RPL data. Packets captured with the nRF52840 sniffer contain 802.15.4 TAP headers[7]. Scapy does not support 802.15.4 TAP out-of-the-box. We extend Scapy with a 802.15.4 TAP parser implementation.

[7]IEEE 802.15.4 TAP protocol - https://github.com/jkcko/ieee802.15.4-tap

Moreover, RPL control frames can contain multiple options. The Scapy RPL parser only auto-parses the first option header in any RPL frame. We manually parse RPL control frame options headers by looping over all options headers until no unparsed data is left.

We also see that RPL control frames containing IPv6 prefixes fail to parse due to invalid length of packed IP address string. This is caused by incorrect prefix length calculation by Scapy in options headers using the _IP6PrefixField class, specifically in the Route Information, RPL Target and Transit Information options. Additionally, this prefix field is optional in these headers and may be omitted completely, while it is always expected to be present in the Scapy implementation. We fix the _IP6PrefixField length calculation to use the length field provided in the header and make this field conditional on the length value being higher than the length of the required fields.

Next we find that IPv6 extension headers fail to parse correctly. Anything past the base IPv6 header is left unparsed as raw payload if any IPv6 extension header is present. This behaviour is a result of an incorrect 6LoWPAN NHC header decompression implementation, which fails to correctly populate *next header* fields of the base IPv6 header and its extensions.

Finally, Stateful IPv6 address decompression issues in the 6LoWPAN IPHC layer parsing cause a single node address to be converted into multiple distinct IPv6 addresses for statefully compressed 0-bit source addresses ($SAC = 1$ and $SAM = 3$). We fix the decompression mechanism by taking the final part of the IPv6 address from the layer 2 802.15.4 header.

6.4. Technical Limitations

Apart from the scoping and assumptions defined in section 5.1, we observed certain technical problems limiting the extent of our research, which we were unable to overcome either through incompatibilities or a lack of time and resources.

The main limitation for the study is the set of available operating systems we can use. As stated in section 4.2, we were unable run and collect data for the ARM Mbed OS and OpenWSN OSes due to a lack of time and hardware resources, limiting the number of usable open-source OSes to four. Furthermore, the assumption of only considering open-source OSes for this research, as mentioned in section 5.1, also has its technical reasons. The lack of access to proprietary OSes and the tooling required to run them, makes it infeasible to simulate them or run on custom hardware. A future research could attempt to include these proprietary OSes by finding off-the-shelf commercial products utilising them, and performing an analysis on fingerprinting those consumer devices.

Another limitation lies in the selection of features for the Sample and Stimulus-Response components. The list of features selected for this study are non-exhaustive, and other behavioural traits may achieve better fingerprinting results. Here we are limited by time constraints and further research is needed to better encapsulate the different kinds of behaviour displayed by these OSes.

7 Analysis

Following the execution of the experiment outlined in the previous chapter, we analyse the obtained data and describe the results. This chapter starts with a description of the classifier we use to analyse the data, and the methods used to evaluate its performance. Next, we detail the specific results obtained by the classifier. Finally, we test for the effects of any confounding variables possibly present in our data, to ensure our previous analyses are valid.

7.1. Classifier Design

Given the set of features we decided on contains singleton type features for each control frame class, and a sample and stimulus-response-based analysis, we create the classifier design shown in figure 7.1, consisting of multiple components accommodating for each type of feature.

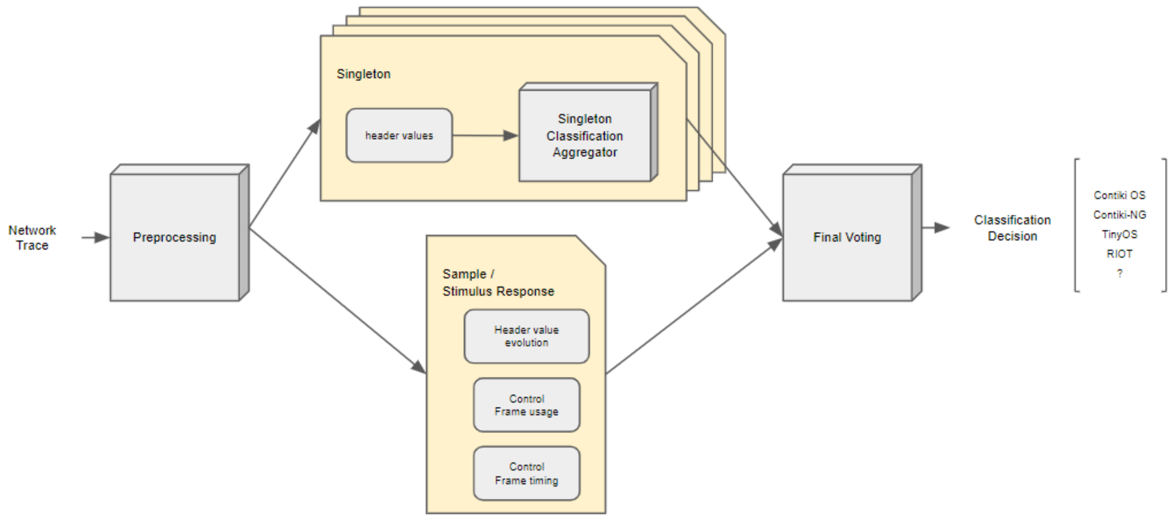


Figure 7.1: Classifier design.

We have several Singleton components, one for each control frame type, and a Sample/Stimulus-Response (SSR) component that analyses the whole traffic stream. The results of these components are combined using a voting-based ensemble method, resulting in a classification decision of one of the 4 OS classes or the unknown class.

Our classifier should not be rule-based, in order not to over-fit our classifiers [10]. Since OS parameters are editable, such methods would perform poorly. For our classifier components, the multi-class multinomial logistic regression model is chosen using a cross-entropy loss function. We attempt to minimise over-fitting by regularising our data using $\ell 2$ regularisation. The classification decision is made by selecting the class with the highest classification score.

The goal of the final classifier is to assign a label to a node corresponding to the OS it is running, based on a trace of its network traffic. Therefore, the classifier must accept a network trace as its input, and, since a network trace may contain traffic from multiple sources, should result in a label and node address per node observed in the trace.

For the singleton components, however, we obtain multiple feature vectors per node from a single trace, as a trace may contain more than one frame of that type for each node. Therefore, assuming m frames transmitted by node N of control frame type C contained within a single network trace, a classification output per N from these components is obtained as follows. For each N with frames $f \in \{f_{CN0}, f_{CN1}, ..., f_{CNm}\}$ we generate m feature vectors $v_{CN0}, v_{CN1}, ..., v_{CNm}$, which we pass on to the singleton component responsible for that control frame alongside with that node's node address a_N. That component then attempts to classify each v_{CN} by assigning class probabilities for each of our n output classes O, resulting in probability vectors $p_0, p_1, ..., p_m$ of the form $(p_{O1}, p_{O2}, ..., p_{On}, a_N)$. Using these probabilities, we assign that vector v_{CNx} to class O_y with the highest p_{Oy} in p_x. The individual vector results are then grouped by node addresses a_N and the class assignments are counted. The class with the largest number of occurrences is selected as the final output decision for N, the final class probability being calculated as the average class probabilities of the individual decisions.

The SSR component requires a much simpler approach. Here we create a feature vector v_N per node N in a trace, and thus only obtain a single output class per N from the component.

The classification decision is made based on the class with the highest class probability. These results are combined in the final step via a voting process, where a decision is made based on a majority vote. The results of the singleton components are combined with the result of the SSR component by matching the node addresses. The final vote will then be performed with one vote per component, which can be assigned different weights depending on the component performance, if necessary.

To decide on the unknown class, we use a simple decision boundary method. If the classification score is below a certain boundary, we do not accept that classification and mark it as the unknown class. We find this boundary empirically by training the classifier for 3 of the 4 classes, also called the one-vs-rest approach, and testing the inlier and outlier probability scores.

7.2. Initial Results

Prior to training, we reserve 25% of our samples for our testing set and use the remaining 75% to train our classifier with. We attempt to classify the samples from our test set and compare the results with the labels we expect. Using these results, we are able to evaluate our classifier's performance by observing its precision, recall and f1-score, according to the following definitions.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$F1score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Since our dataset is unbalanced among its classes in certain cases, we do not consider accuracy as a metric for performance, but rather look at the balanced accuracy, which for multiclass classification is the same as the macro average of the classifier's recall for each class.

Before testing the full classifier, we analyse the Singleton and SSR components individually in order to assess their effectiveness and role in the final classification task. The component performances are used to select for the most performant components and exclude any that underperform, since these will degrade the performance of the final classification as well. For this, we split our samples into training and test sets, perform the standard preprocessing and scaling necessary using all features, and then train the classifier as expected, up to the point where the results of the individual components get combined. At this point, we can obtain the classification results of the individual components, which we then compare with the expected labels. Table 7.1 shows the classification performance per component in the form of accuracy, precision and recall, from which can be observed that singleton DIO performs exceptionally well. The other components, however, are not as performant on their own, and thus not suitable for use within our classifier. Because of this, those components are excluded from the final version, resulting in a final classifier design as shown in figure 7.2.

Component	**Precision**	**Recall**
DIS	0.25	0.25
DIO	0.82	0.88
DAO	0.63	0.57
DAO-ACK	0.47	0.35
Data	0.48	0.50
Sample/SR	0.58	0.59

Table 7.1: Classification performance of individual components of the classifier.

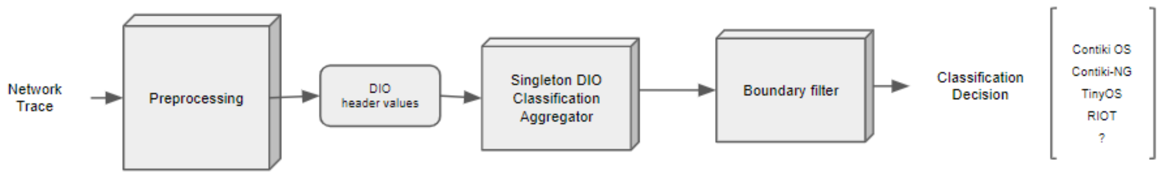


Figure 7.2: Full classifier design after excluding underperforming components.

7.3. Feature Analysis

Prior to building the final classifier, we use various methods to analyse our features and reduce the dimensionality of our feature-set. First we remove all features whose data show zero variance between all classes present. These features will have no effect on the classification task, since they will be equal among all data points. The features removed from this step can be found in appendix B.

Next, we perform a collinearity analysis in order to find and remove any highly multi-collinear features still present in our feature-set, as these may impact the regression of our model. This allows us to reduce the feature-space without too great a hit on performance, since the trend of the value is also included in other features within the dataset. We calculate a Spearman correlation matrix and perform hierarchical clustering on highly correlated features. Then we pick one feature per cluster as representative of that cluster of features. Clustering is performed using a cluster boundary that is decided after visual inspection of the distance values, choosing a boundary that will eliminate highly collinear features, while not removing too many and affecting the performance of the classifier. Figure 7.3 shows the Spearman correlation matrix and the corresponding distance values for each feature in the form of a dendrogram. The lower the distance value is between two features, the more collinear they are.

From these distance values we observe that the DIO feature-set contains highly collinear features, with an obvious cluster boundary around 0.1. Clustering for this boundary value allows us to reduce the number of features from 20 to 14.

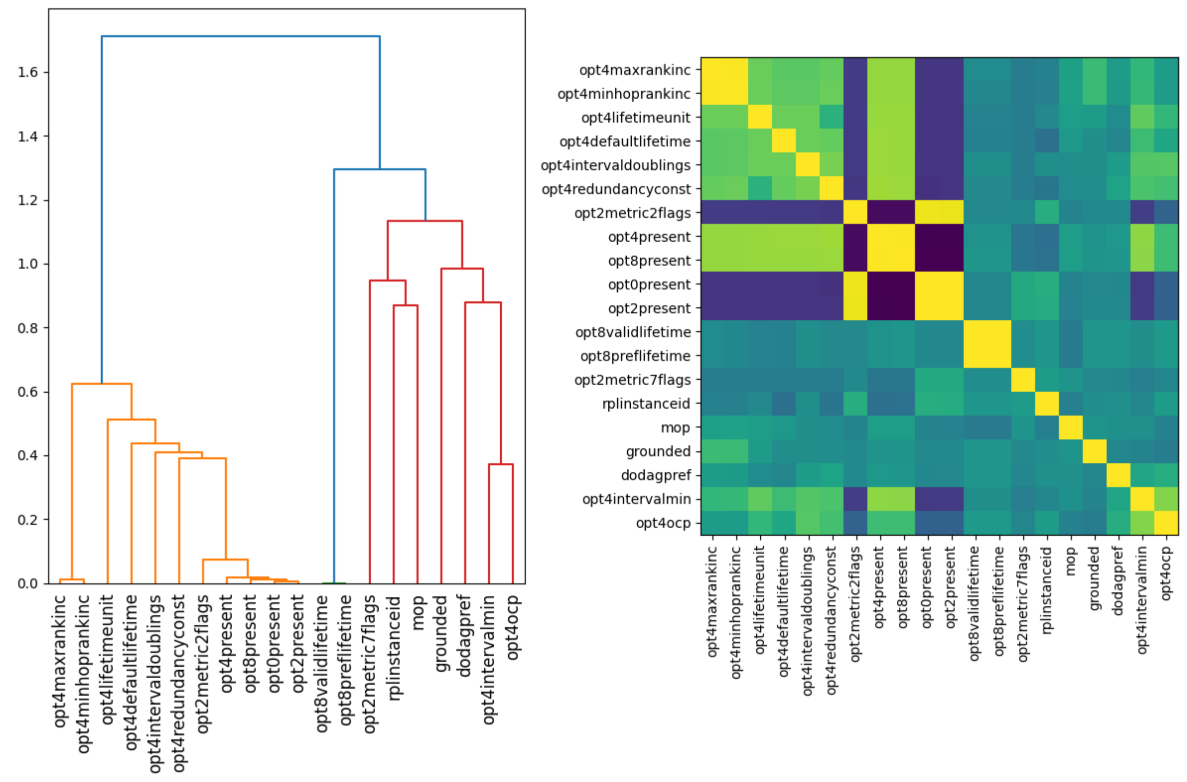


Figure 7.3: Spearman correlation coefficients matrix (right) and dendrogram of the resulting distance linkage (left) between features of the Singleton DIO classifier component

Finally, we observe the regression coefficients for each feature and class, obtained by training our multinomial logistic regression model. This gives us the usefulness of each feature for the classification decision for each class, from which we can find the features that do not effect the classification decision due to their ambiguity, and remove them from our feature-set.

The coefficients in a multinomial logistic regression model are obtained per feature per class. We consider a feature useful if the coefficient values per feature are high for any of the classes. Figure 7.4 shows the maximum coefficient value of each feature among all classes per component. From the DIO feature-set, we remove any features with a coefficient value below 2.5, allowing us to drop 4 more features. The list of remaining features can be found in table 7.2. Table 7.3 shows the performance results of the DIO component after the dimensionality reduction, showing a considerable increase in performance after the process.

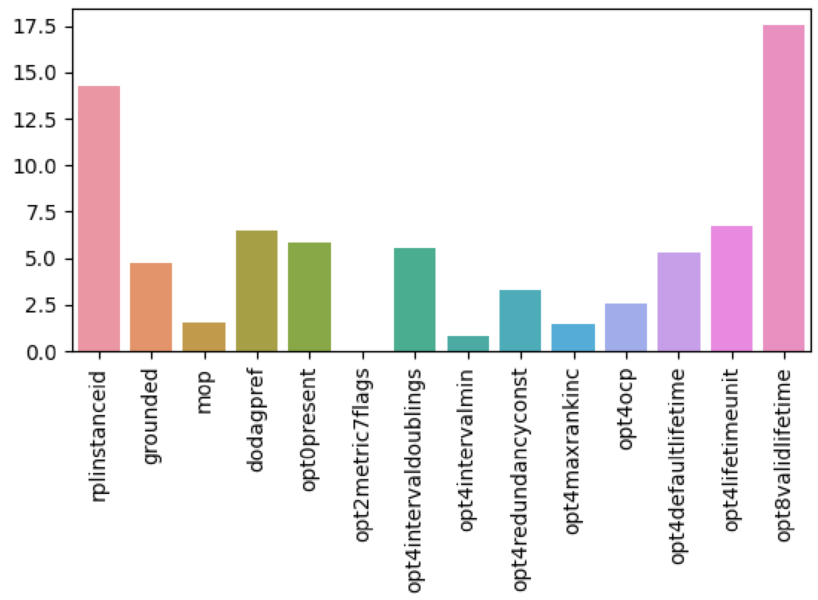


Figure 7.4: Logistic regression coefficient values of the DIO classifier component. The value indicates a feature's importance for the classification of a feature vector as at least one of the classes.

Remaining Features
RPLInstanceID
Grounded
DODAG preference
Option 0 Present
Option 4 IntervalDoublings
Option 4 Redundancy Constant
Option 4 OCP
Option 4 Default Lifetime
Option 4 Lifetime Unit
Option 8 Valid Lifetime

Table 7.2: DIO component features remaining after the reduction steps.

Component	**Precision**	**Recall**	**F1 Score**
DIO	1.00	0.96	0.98

Table 7.3: Classification performance of the DIO component.

7.4. Finding the Decision Boundaries

As mentioned in section 7.1, in order to label a certain sample as the unknown class, we need to decide on boundaries under which it is considered too ambiguous and is labeled as unknown. Since we do not have any data from an unknown class, we have to mimic this behaviour by using a one-vs-rest method, where we train our classifier by excluding one of the classes and using it as an unknown class. This process can be repeated by using each of the other classes as the unknown class, in order to obtain a more generalised view of the effects of the decision boundaries.

We attempt to find the optimal boundary at which to consider a feature vector as an outlier, based on the classifier's prediction score for that vector. For each class separated as an outlier, we obtain the prediction probabilities for the vectors in our test set and plot for all boundaries the prediction performance for the inlier and outlier class, in the form of precision and recall. Since our dataset is highly unbalanced, we balance the dataset in two steps. First we sample from each inlier class a number of feature vectors equal to a third of the size of the selected outlier class. Next, if the total size of the subsampled inlier dataset is smaller than the outlier set, we subsample the outlier set to equalise the sample sizes.

The results of the boundary analysis can be found in figure 7.5, with the precision scores on the left and recall score on the right. We find that the optimal boundary at which both precision and recall is maximised lies around 0.95.

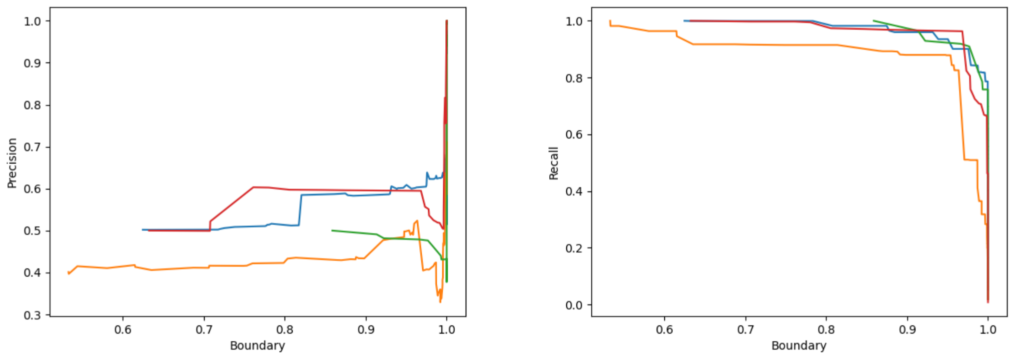


Figure 7.5: Precision (left) and recall (right) vs. unknown class boundary.

7.5. Classifier Performance

Before enforcing the selected decision boundary and evaluating its performance, we break down the results obtained from the full classifier without decision boundaries. From table 7.4 we observe a balanced accuracy of 97% for inlier classes. The only misclassifications happen between Contiki-NG and Contiki OS, as indicated by a 91% precision for Contiki OS and an 89% recall for Contiki-NG. This amount of ambiguity between those OSes is understandable, given that Contiki-NG is based on Contiki OS.

	precision	**recall**	**f1-score**
contiking	1.00	0.89	0.94
contikios	0.91	1.00	0.95
riot	1.00	1.00	1.00
tinyos	1.00	1.00	1.00
macro avg	0.98	0.97	0.97
weighted avg	0.96	0.96	0.96

Table 7.4: Classification performance of final classifier.

Next we incorporate a decision boundary of 0.95 for the outlier class and investigate whether our classifier's performance degrades for the inlier classes. As seen in table 7.5, the balanced accuracy drops from 97% to 74%, due to drops in recall for RIOT and TinyOS. To better understand why this drop in performance happens, we observe the confusion matrix, shown in table 7.6. Here we see that RIOT and TinyOS samples are being classified as unknown (50% of the time for RIOT and 43% of the time for TinyOS), resulting in the degradation in performance. This shows us that the selected decision boundary is not appropriate for all our classes, and should be lower for these two classes.

	precision	recall	f1-score
contiking	1.00	0.89	0.94
contikios	0.91	1.00	0.95
riot	1.00	0.50	0.67
tinyos	1.00	0.57	0.73
macro avg	0.98	0.74	0.82
weighted avg	0.96	0.85	0.89

Table 7.5: Classification performance of classifier with unknown class boundary at 0.95.

Real \Predicted	Unknown	Contiki-NG	Contiki OS	RIOT	TinyOS
Unknown	0	0	0	0	0
Contiki-NG	0	40	5	0	0
Contiki OS	0	0	50	0	0
RIOT	8	0	0	8	0
TinyOS	6	0	0	0	8

Table 7.6: Expected vs. predicted results with unknown class boundary of 0.95.

We also investigate the classifier's unknown class identification capabilities, using the one-vs-rest method earlier described. For each class introduced as outliers, table 7.7 lists the outlier detection performance of this classifier. From these results we find that the outlier detection performance of the classifier is poor, with and average recall of 25%, but also dependent on the outlier class.

Outlier Class	Outlier Precision	Outlier Recall	Outlier F1-score
Contiki-NG	0.78	0.38	0.51
Contiki OS	0.68	0.26	0.37
RIOT	0.00	0.00	0.00
TinyOS	0.67	0.35	0.46
Average	**0.53**	**0.25**	**0.34**

Table 7.7: Outlier class classification performance of the full classifier using a boundary of 0.95, tested for each class introduced as outlier class, displayed as macro precision, recall and f1-score averages.

7.6. Alternative Classifier

We attempt to improve the poor outlier classification results by creating a new classifier based on a different model. Rather than using a multinomial logistic regression model, we create n binary logistic regression models, one per class c, and use them for the final classification with a One vs Rest approach: Model n_c is used to assess whether a feature vector f is of class c, resulting in 1 if $f \in c_x$ and 0 otherwise. By default, the model assumes a decision boundary of 0.5, where a score over this value results in a decision of 1. Using the classification scores of each binary model, the final decision is made based on the model n_x with the highest score. A diagram of this classifier is shown in figure 7.6.

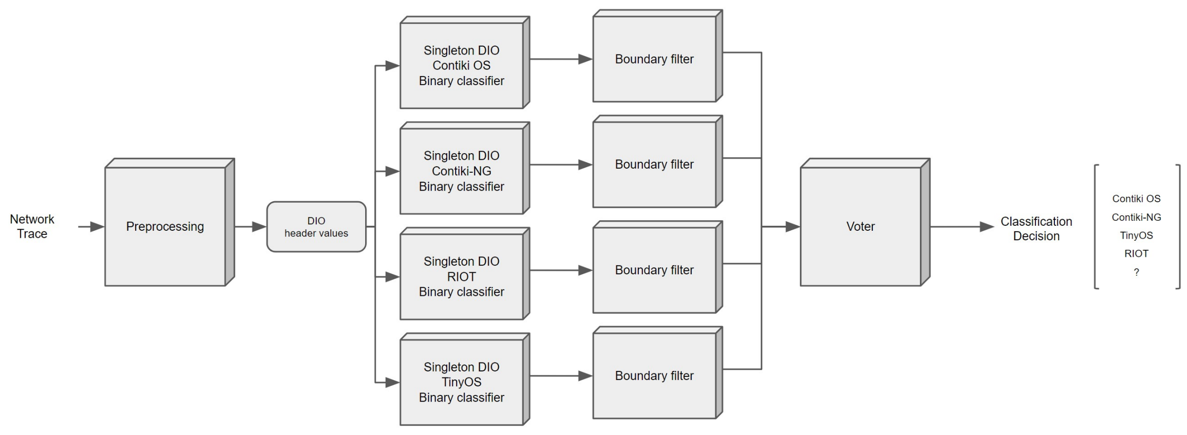


Figure 7.6: Classifier redesign taking a One vs Rest binary classification approach.

We first assess this classifier by calculating its performance in the DIO component, found in table 7.7a, and the performance of the full classifier based on this approach, visible in table 7.7b. The results are in line with the performance of to the previous classifier.

Precision	**Recall**	**F1 Score**
1.00	0.93	0.96

(a) DIO component One vs Rest performance

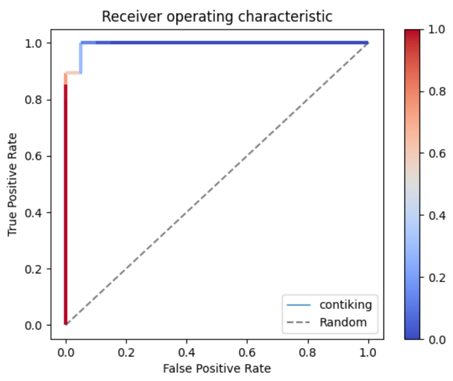
Precision	**Recall**	**F1 Score**
0.98	0.98	0.98

(b) Full One vs Rest classifier performance

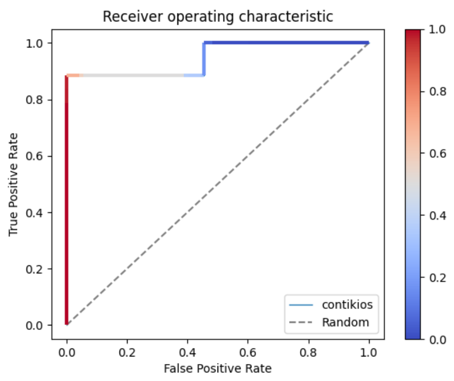
Figure 7.7: Classification performances of the DIO component and full classifier using the One v Rest approach, displayed as macro precision, recall and f1-score averages.

Next we apply boundary shifting for each model by first graphing their ROC curves and finding the optimal boundaries. If the classification score of a model falls below its selected boundary, we disregard this model's prediction and assign it a score of 0. This way, the class prediction is performed by selecting the class with the highest score that is above its individual decision boundary.

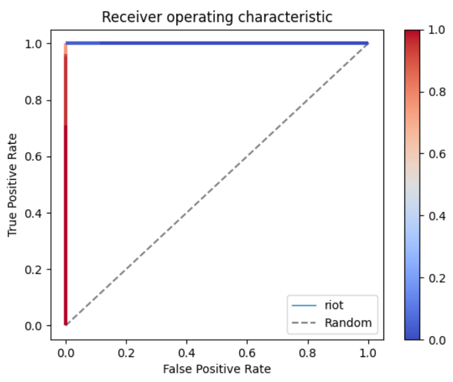
The ROC curves are shown in figure 7.8 for each model/class, with the colours of the lines corresponding to the boundary values. From these, we decide on the boundary values maximising the true positive rate, while keeping the false positive rate below 1%, resulting in the values in table 7.8. Before evaluating the outlier detection performance of this classifier, we make sure our boundary shifting has not negatively affected our classification performance by comparing the classification results for our test set both with and without the boundaries. As shown in tables 7.7b and 7.9, our classification performance is not affected, except for a negligible drop in recall.



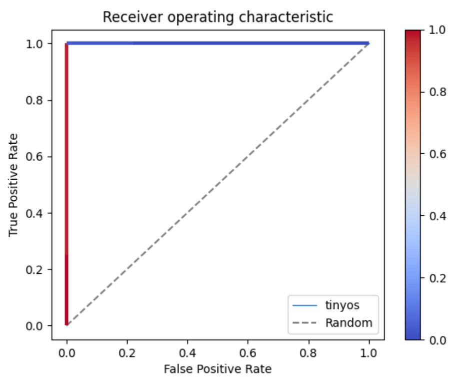
(a) Contiki-NG



(b) Contiki OS



(c) RIOT



(d) TinyOS

Figure 7.8: ROC curves of One vs Rest binary logistic regression models. The decision boundary values are indicated by the colours of the graph.

Positive Class	**Decision Boundary**
Contiki-NG	0.74
Contiki OS	0.68
RIOT	0.70
TinyOS	0.55

Table 7.8: Decision boundaries selected for the One vs Rest models.

	Precision	**Recall**	**F1 Score**
Full classifier	0.98	0.97	0.98

Table 7.9: Classification performances of the full classifier using the One v Rest method with the selected decision boundaries, displayed as macro precision, recall and f1-score averages.

Once again, we investigate the classifier's unknown class identification capability using the one-vs-rest method. For each class introduced as outliers, table 7.10 lists the outlier detection performance. This classifier also performs poorly for outlier classes, with an average recall of 30%.

Outlier Class	**Outlier Precision**	**Outlier Recall**	**Outlier F1-score**
Contiki-NG	1.00	0.39	0.57
Contiki OS	0.86	0.10	0.18
RIOT	0.00	0.00	0.00
TinyOS	0.91	0.70	0.79
Average	**0.69**	**0.30**	**0.39**

Table 7.10: Outlier class classification performance of the full classifier using the One v Rest method with the selected decision boundaries, tested for each class introduced as outlier class, displayed as macro precision, recall and f1-score averages.

7.7. Effects of Environment and Platform

As we have opted to use multiple environments and differing hardware platforms during data collection, it is important to ensure our data is not influenced by this decision. It is possible that an OS may behave differently due to the underlying properties of the execution environment. To test this, we generate two datasets for the same OS, generated on separate environments and hardware platforms, and attempt to differentiate between the feature-sets resulting from them.

The possible OS-environment mappings were already mentioned previously in table 4.2. We find that both Contiki OS and Contiki-NG can be used in multiple environments. We choose to use Contiki OS for this task, generating samples and running them within the Cooja and Renode environments. For the dataset to be run with the Renode environment, we use the samples with data traffic generated previously for the ARM CC2538 platform. To run the OS in Cooja, we create samples with the same parameter configuration as the other dataset, but for the MSP430-based Tmote Sky platform. This allows us to simultaneously test for environment and platform differences, and observe how they affect the selected features.

In order to evaluate the effects of these choices, we extract feature vectors of both datasets and compare them by calculating the Euclidian distance between vectors of the first set against the vectors of the second. If this distance is 0, the feature-sets will be perfectly overlapping and no features will be affected. We calculate the Euclidian distance between feature vectors of both classes in a crosswise fashion and obtain a distance of 0.0, indicating that the all vectors are identical. This shows that environment and platform selection does not effect the dataset for the features used in our DIO-based classifier.

We also attempt to build a classifier that can distinguish between these two datasets, using the same features selected for our original classifier. If the datasets are unaffected by environment choices, we expect this classifier to perform poorly, with an accuracy nearing 50%, similar to random choice. The results of this classifier can be found in table 7.11 below. As expected, the accuracy of this classifier approaches that of random choice with a value of 44%, with all samples being assigned to a single class.

Environment & Platform	**Precision**	**Recall**	**F1-score**
Cooja & Sky	0.00	0.00	0.00
Renode & CC2538	0.51	0.88	0.64
Macro average	**0.25**	**0.44**	**0.32**

Table 7.11: Classifier performance to differentiate between environment and platform differences while running Contiki OS.

8 Conclusion

From the findings in chapter 7, we are able to conclude that OS identification from RPL behaviour can feasibly be performed. Enough distinctive features can be found within the header contents of the DIO control frame to achieve a balanced classification accuracy of 97%. The other control frame types, however, only publicise a limited number of features, making them less useful for fingerprinting. Using traffic meta-data is also complicated by the lack of retransmission controls within the protocol specifications.

Additionally, efforts to detect unknown classes by using decision boundaries have proven unsuccessful, only achieving an average recall of 30% on outlier classes, thus limiting the usefulness of RPL OS classification in the wild unless data on all existing OSes are collected and included in the training phase of the classifier.

8.1. Discussion

While the results of this study show that OS identification using RPL is indeed a possibility, it should only be considered an initial proof-of-concept, mainly due to the simplifying assumptions made during the data collection and experimentation stages. A more detailed research is required into RPL-based OS identification in which not only more operating systems should be considered, but the scenarios under which the networks are being investigated should be greatly expanded upon.

First, the size, shape and density of the RPL networks are all variables we have kept mostly constant. It is unknown whether the protocol would react differently to larger networks where collisions and congestion happen more often. The protocol also has mobility support, a concept we have not considered in our experiments, which could cause nodes to behave in ways different than when all of them are static. These factors would mostly manifest themselves in features based on timing meta-data.

Another point of contention may stem from our assumption that all nodes within our created networks follow the same RPL parameters. Since DIO parameters of the DODAG are defined and propagated by the root node, header contents of this control frame transmitted by other nodes may be different than our classifier expects. This could cause misclassifications if nodes within a network run different OSes than the root node.

Finally, our assumption of nodes not making use of any encryption mechanisms, while simplifying our access to header contents and increasing our avenues toward correct classification, also makes our experiments less representative of real-world scenarios. In the wild, network administrators may opt to use encryption mechanisms on one or multiple layers of the network stack, in the form of link-layer encryption or RPL Secure, or using application layer protocols that support encrypted communication. While application-layer encryption would not pose any difficulties for our methods, RPL Secure and link-layer encryption make observing the contents of the control frames used within this research impossible. In these cases, classification would have to be performed either using header data of the secure variants of the control frames, or fully rely on meta-data analysis.

9

Future Research

Since this research only aimed at providing a proof of concept for OS fingerprinting using RPL, its scope was kept fairly limited. For this reason, this section outlines certain aspects not considered during our experiments and provides a set of future research possibilities.

Network Size and Composition

As mentioned in the discussion in chapter 8, there are several aspects of the RPL environment we have left out of scope during this research. A logical continuation of the current research could therefore be one where these considerations are taken into account.

One such example is the usage of heterogeneous networks, where nodes within the network use different parameters or are of distinct OSes. While DIO parameters of the nodes within the DODAG are defined and propagated by the root node, it is perfectly possible that certain OSes disregard the DODAG-wide parameters altogether and use their own, or only take these into effect partially. Previous interoperability studies for mixed-OS networks using Contiki OS and TinyOS nodes have shown some compatibility issues and a loss of performance [33, 48, 49]. While there is an indication that these incompatibilities may be caused by mismatched parameters in the MAC layer or other reasons outside of RPL [49], whether any observable differences occur in the routing layer should be more thoroughly investigated.

Another important factor to look into is the effect of network shape and size on the routing behaviour. If the network composition affects our feature vectors in a way that makes generalisations on the underlying OS impossible, this will in turn make performing OS identification in the wild infeasible, as networks in the wild may differ widely in size and composition.

Finally, networks in the wild might also utilise encryption mechanisms that make it impossible to observe headers contents in the way we used within the current research. This would require a deeper investigation into the possibility of meta-data analysis than has been performed within this research, and make better use of timing-based side channels observable in RPL-based traffic.

Additional Features

The current research has also understandably been limited by the level at which current OSes have implemented the RPL specifications. In the future, however, it is reasonable to expect that other parts of the standard and its extensions will be available to end-users, in which case the efforts required to perform OS identification may greatly vary.

Current implementations, for example, do not support RPL Secure from the original standard, which make use of encrypted RPL control frames, but rather use link layer encryption provided by IEEE 802.15.4. RPL Secure introduces 5 new control frames which, if ever implemented, could provide additional possibilities for OS identification. Many extension of RPL have been proposed as well, since the original conception of the protocol, that define addition behaviour and control frame types that can potentially be used for fingerprinting using similar methods as those in the current research [13, 32, 40].

More RPL Implementations

There is also always the possibility of more OSes being created implementing RPL, apart from the current set of OSes chosen for this research. Since we have not been able to successfully perform outlier detection with our current dataset and classifier, it is important to extend the dataset as much as possible, as soon as a new implementation emerges.

This research has only included open-source OSes with RPL support, but a similar investigation could be carried out using existing proprietary RPL implementations. Some of these may even be based on existing open-source variants [62], possibly resulting in very similar observable behaviour, making it an interesting challenge for future research.

Research on the current set of OSes could also be extended to attempt to distinguish not only between the OSes themselves, but also between different versions of the RPL implementations within the OSes. Being able to identify the exact version of a software running on a device can be extremely valuable in establishing whether that device is exposed to a certain vulnerability, in cases, for example, where that vulnerability has been fixed in later versions of that software.

Other Technologies

Fingerprinting of RPL OSes is potentially not only limited to RPL behaviour, given that these OSes also all provide their own implementations of the network and MAC layers on which RPL relies. There have been examples within previous research where some sort of fingerprinting was possible with these lower layer protocols [43, 75], but a fully passive OS identification method using these lower layer protocols within the context of RPL-based OSes is still missing, and would be a possible avenue to pursue for further research.

Finally, given the fact that RPL was designed to be usable within different network stacks, with wildly varying underlying technologies, another interesting investigation could be to use RPL over different physical and MAC layers, and see how this affects RPL behaviour.

A
Editable RPL Parameters

All OSes used in this study provide the ability to adjust the parameters of their RPL protocol implementations. This chapter lists all editable parameters of each OS, in addition to their default values and the ranges of values accepted for those parameter.

Contiki OS

RPL_CONF_OF
Default value: 1 (MRHOF)

Options: 0, 1

RPL_CONF_DAG_MC
Default value: 0 (No Metric/Constraint used)

Options: 0, 2, 7

Contiki OS contains the following comment:
When MRHOF (RFC6719) is used with ETX, no metric container must be used; instead the rank carries ETX directly.
So, the value of this parameter should be 0 if RPL_CONF_OF is set to 1.

RPL_CONF_DEFAULT_INSTANCE
Default value: 0x1e

Options: 0x00 - 0xff

RPL_CONF_DAO_SPECIFY_DAG
Default: 1

Options: 0, 1

RPL_CONF_DIO_INTERVAL_MIN
Default: 12

Options: 0 - 0xff

For feasibility, we limit the value to be between 0 and 0x0f (32 seconds). If higher, we might not be able to observe any DIO traffic within the time-frame of our simulations.

RPL_CONF_DIO_INTERVAL_DOUBLINGS
Default: 8

Options: 0 - 0xff

RPL_CONF_DIO_REDUNDANCY
Default: 10

Options: 0 - 0xff

RPL_CONF_DEFAULT_LIFETIME_UNIT
Default: 0xffff

Options: 0x0000 - 0xffff

RPL_CONF_DEFAULT_LIFETIME
Default: 0xff

Options: 0x00 - 0xff

RPL_CONF_PREFERENCE
Default: 0

Options: 0 - 7

RPL_CONF_GROUNDED
Default: 0

Options: 0, 1

RPL_CONF_DAO_ACK
Default: 0

Options: 0, 1

RPL_CONF_MIN_HOPRANKINC
Default: 256

Options: 0 - 0x2492

$RPL\_MAX\_RANKINC = (7 * RPL\_MIN\_HOPRANKINC)$ should be below 0xffff.

RPL_CONF_MOP
Default: 2

Options: 0, 1, 2, 3

RPL_CONF_DEFAULT_ROUTE_INFINITE_LIFETIME
Default: 0

Options: 0, 1

RPL_CONF_INSERT_HBH_OPTION
Default: 1

Options: 0, 1

RPL_CONF_WITH_PROBING
Default: 1

Options: 0, 1

RPL_CONF_PROBING_INTERVAL
Default: 120s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0xfff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_PROBING_EXPIRATION_TIME
Default: 600s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0xfff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_LEAF_ONLY
Default: 0

Options: 0, 1

RPL_CONF_DAO_LATENCY
Default: 4s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0xfff (31 seconds). If higher, we might not be able to observe any DAO traffic within the time-frame of our simulations.

RPL_CONF_MULTICAST
Default: 0

Options: 0, 1

RPL_CONF_MCAST_LIFETIME
Default: 3

Options: 0 - 0xff

RPL_DIS_INTERVAL_CONF
Default: 60

Options: 0 - 0xffff

For feasibility, we limit the value to be between 0 and 0xfff (31 seconds). If higher, we might not be able to observe any DIS traffic within the time-frame of our simulations.

Contiki-NG

RPL_CONF_MOP
Default: 1

Options: 0, 1

RPL_CONF_OF_OCP
Default: 1

Options: 0, 1

RPL_CONF_SUPPORTED_OFS
Default: &rpl_mrhof
Options: &rpl_of0, &rpl_mrhof, &rpl_of0, &rpl_mrhof

The OF supported by including it with this parameter should correspond with the value assigned to *RPL_CONF_OF_OCP*. If the OCP is set to 0, but rpl_of0 is not supported, or if OCP is 1 and rpl_mrhof is not supported, RPL will not work.

RPL_CONF_DIO_INTERVAL_MIN
Default: 12

Options: 0 - 0xff

For feasibility, we limit the value to be between 0 and 0x0f (32 seconds). If higher, we might not be able to observe any DIO traffic within the time-frame of our simulations.

RPL_CONF_DIO_INTERVAL_DOUBLINGS
Default: 8

Options: 0 - 0xff

RPL_CONF_DIO_REDUNDANCY
Default: 0

Options: 0 - 0xff

RPL_CONF_DEFAULT_LIFETIME_UNIT
Default: 60

Options: 0 - 0xffff

RPL_CONF_DEFAULT_LIFETIME
Default: 30

Options: 0 - 0xff

RPL_CONF_WITH_MC
Default: 0

Options: 0, 1

RPL_CONF_DAG_MC
Default: 0

Options: 0, 2, 7

RPL_CONF_WITH_DAO_ACK
Default: 1

Options: 0, 1

RPL_CONF_MAX_RANKINC
Default: 1024

Options: 0 - 0xffff

RPL_CONF_MIN_HOPRANKINC
Default: 128 if $RPL\_CONF\_OF\_OCP = 1$, else 256

Options: 0 - 0xffff

If $RPL\_CONF\_MAX\_RANKINC$ is not set by the user, it becomes $8 * RPL\_CONF\_MIN\_HOPRANKINC$, which should fit within a uint32. In this case, $RPL\_CONF\_MIN\_HOPRANKINC$ should be between 0 - 0x1fff.

RPL_CONF_DEFAULT_INSTANCE
Default: 0

Options: 0 - 0xff

RPL_CONF_GROUNDED
Default: 0

Options: 0, 1

RPL_CONF_PREFERENCE
Default: 0

Options: 0 - 7

RPL_CONF_DIS_INTERVAL
Default: 30s

Options: 0 - 0xffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe any DIS traffic within the time-frame of our simulations.

RPL_CONF_WITH_PROBING
Default: 1

Options: 0, 1

RPL_CONF_PROBING_INTERVAL
Default: 90s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_DELAY_BEFORE_LEAVING
Default: 300s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_DAO_RETRANSMISSION_TIMEOUT
Default: 5s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_DAO_MAX_RETRANSMISSIONS
Default: 5

Options: 0 - 0xff

RPL_CONF_DAO_DELAY
Default: 4s

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe any DAO traffic within the time-frame of our simulations.

RPL_CONF_TRICKLE_REFRESH_DAO_ROUTES
Default: 0 if $RPL\_CONF\_WITH\_DAO\_ACK = 1$ else 4

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_DAG_LIFETIME
Default: 480

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RPL_CONF_DEFAULT_LEAF_ONLY
Default: 0

Options: 0 / 1

RPL_CONF_SIGNIFICANT_CHANGE_THRESHOLD
Default: 512

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x0fff (31 seconds). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

RIOT

GNRC_RPL_DEFAULT_DIO_INTERVAL_MIN
Default: 3

Options: 0 - 0xff

For feasibility, we limit the value to be between 0 and 0x0f (32 seconds). If higher, we might not be able to observe any DIO traffic within the time-frame of our simulations.

GNRC_RPL_DEFAULT_DIO_INTERVAL_DOUBLINGS
Default: 20

Options: 0 - 0x7FFFFFFF

$GNRC\_RPL\_DEFAULT\_DIO\_INTERVAL\_MIN * 2^{GNRC\_RPL\_DEFAULT\_DIO\_INTERVAL\_DOUBLINGS}$ should be lower than $UINT32\_MAX/2$ (0x7FFFFFFF), so its maximum value depends on the value of the $GNRC\_RPL\_DEFAULT\_DIO\_INTERVAL\_MIN$ parameter.

GNRC_RPL_DEFAULT_DIO_REDUNDANCY_CONSTANT
Default: 10

Options: 0 - 0xff

GNRC_RPL_DEFAULT_LIFETIME
Default: 5

Options: 0 - 0xff

GNRC_RPL_LIFETIME_UNIT
Default: 60

Options: 0 - 0xffff

GNRC_RPL_MOP_NO_DOWNWARD_ROUTES
Default: 0

Options: 0, 1

GNRC_RPL_MOP_NON_STORING_MODE
Default: 0

Options: 0, 1

GNRC_RPL_MOP_STORING_MODE_NO_MC
Default: 1

Options: 0, 1

GNRC_RPL_MOP_STORING_MODE_MC
Default: 0

Options: 0, 1

GNRC_RPL_WITHOUT_PIO
Default: 0

Options: 0, 1

GNRC_RPL_DEFAULT_MIN_HOP_RANK_INCREASE
Default: 256

Options: 0 - 0xffff

GNRC_RPL_DEFAULT_MAX_RANK_INCREASE
Default: 0

Options: 0 - 0xffff

GNRC_RPL_DEFAULT_INSTANCE
Default: 0

Options: 0 - 0xff

GNRC_RPL_DODAG_CONF_OPTIONAL_ON_JOIN
Default: 0

Options: 0, 1

GNRC_RPL_WITHOUT_VALIDATION
Default: 0

Options: 0, 1

GNRC_RPL_DAO_SEND_RETRIES
Default: 4

Options: 0 - 0xff

GNRC_RPL_DAO_ACK_DELAY
Default: 3000ms

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x3fff (16383ms). If higher, we might not be able to observe any DAO-ACK traffic within the time-frame of our simulations.

GNRC_RPL_DAO_DELAY_JITTER
Default: 1000ms

Options: 0 - 0xffffffff

For feasibility, we limit the value to be between 0 and 0x3fff (16383ms). If higher, we might not be able to observe any DAO traffic within the time-frame of our simulations.

GNRC_RPL_DAO_DELAY_DEFAULT
Default: 1000ms

Options: 0 - 0x0fffffff

$GNRC\_RPL\_DAO\_DELAY\_DEFAULT + GNRC\_RPL\_DAO\_DELAY\_JITTER$ should be lower than $UINT32\_MAX$ (0xFFFFFFFF), so its maximum value depends on the value of the $GNRC\_RPL\_DAO\_DELAY\_JITTER$ parameter. For feasibility, we limit the value of $GNRC\_RPL\_DAO\_DELAY\_DEFAULT + GNRC\_RPL\_DAO\_DELAY\_JITTER$ to be between 0 and 0xffff (65535ms). If higher, we might not be able to observe any DAO traffic within the time-frame of our simulations.

GNRC_RPL_DAO_DELAY_LONG
Default: 60000ms

Options: 0 - 0x0fffffff

$GNRC\_RPL\_DAO\_DELAY\_LONG + GNRC\_RPL\_DAO\_DELAY\_JITTER$ should be lower than $UINT32\_MAX$ (0xFFFFFFFF), so its maximum value depends on the value of the $GNRC\_RPL\_DAO\_DELAY\_JITTER$ parameter. For feasibility, we limit the value of $GNRC\_RPL\_DAO\_DELAY\_LONG+GNRC\_RPL\_DAO\_DELAY\_JITTER$ to be between 0 and 0xffff (65535ms). If higher, we might not be able to observe any DAO traffic within the time-frame of our simulations.

GNRC_RPL_CLEANUP_TIME
Default: 5000ms

Options: 0 - 0xffffffff
For feasibility, we limit the value to be between 0 and 0x3fff (16383ms). If higher, we might not be able to observe this behaviour within the time-frame of our simulations.

GNRC_RPL_PARENT_TIMEOUT_DIS_RETRIES
Default: 3

Options: 0 - 0xfd

TinyOS

RPL_STORING_MODE
Default: 1

Options: 0, 1

RPL_OF_0 / RPL_OF_MRHOF
Default: RPL_OF_0

Options: RPL_OF_0, RPL_OF_MRHOF

B

Feature reduction Results

This chapter contains the results of our feature analyses, performed specifically on the DIO feature-set. The original set of features for DIO was shown earlier in table 4.5. From this, we remove all features with 0 variance among all classes. The features removed during this process, can be found in table B.1, with the remaining features listen in table B.2.

Removed feature
Option 1 present
Option 3 present
Option 1 length
Option 2 metric 1 present
Option 2 metric 2 present
Option 2 metric 3 present
Option 2 metric 4 present
Option 2 metric 5 present
Option 2 metric 6 present
Option 2 metric 7 present
Option 2 metric 8 present
Option 2 unknown metric present
Option 2 metric 1 precedence
Option 2 metric 2 precedence
Option 2 metric 3 precedence
Option 2 metric 4 precedence
Option 2 metric 5 precedence
Option 2 metric 6 precedence
Option 2 metric 8 precedence

Table B.1: All features removed from the feature-set of the DIO component after 0-variance reduction.

Final feature-set
RPLInstanceID
Grounded
MOP
DODAGPreference
Option 0 present
Option 2 present
Option 4 present
Option 8 present
Option 2 metric 7 precedence
Option 4 DIOIntervalDoublings
Option 4 DIOIntervalMin
Option 4 DIORedundancyConstant
Option 4 MaxRankInc
Option 4 MinHopRankInc
Option 4 OCP
Option 4 default lifetime
Option 4 lifetime unit
Option 8 valid lifetime
Option 8 preferred lifetime

Table B.2: All features remaining for DIO component after 0-variance reduction.

For the remaining features, we obtain the logistic regression coefficient per class for each feature. This indicates the importance of that feature for the classifier's decision to classify it as that class. Figure B.1 shows the results of this analysis for the DIO feature-set.

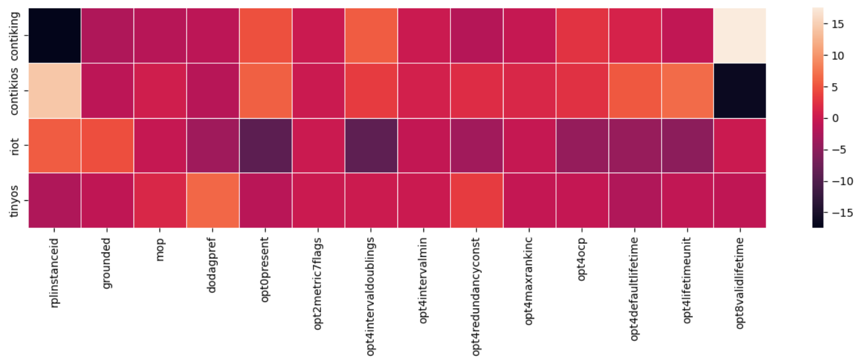


Figure B.1: Logistic regression coefficients of the features of the DIO classifier component per class.

C
6LoWPAN Address Compression

The 6LoWPAN IPHC header defines multiple address compression mechanism, both for the source and destination IPv6 addresses. These are defined by the Source Address Compression (SAC), Source Address Mode (SAM), Multicast compression (M), Destination Address Compression (DAC) and Destination Address Mode (DAM) fields. Tables C.1 and C.2 list the header value combinations and the accompanying source and destination address compression modes.

SAC	**SAM**	**Compression format**
0	0	Full 128-bit IPv6 address is provided.
0	1	Last 64 bits of the IPv6 address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes.
0	2	Last 16 bits of the IPv6 address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes. The remaining 48 bits in between are 0000:00ff:fe00.
0	3	No address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes. The remaining 64 bits should be obtained from the lower layer headers, e.g. 802.15.4.
1	0	The unspecified address 0000:0000:0000:0000:0000:0000:0000:0000.
1	1	Last 64 bits of the IPv6 address is provided. The first 64 bits should be derived from context information. Any remaining bits in between (not covered by the context) are 0.
1	2	Last 16 bits of the IPv6 address is provided. The first 112 bits should be derived from context information. Any remaining bits in between (not covered by the context) are derived from 0000:00ff:fe00.
1	3	No address is provided. The address should be obtained from context information. Any remaining bits (not covered by the context) should be obtained from the lower layer headers, e.g. 802.15.4.

Table C.1: 6LoWPAN IPHC source address compression.

M	**DAC**	**DAM**	**Compression format**
0	0	0	Full 128-bit IPv6 address is provided.
0	0	1	Last 64 bits of the IPv6 address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes.
0	0	2	Last 16 bits of the IPv6 address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes. The remaining 48 bits in between are 0000:00ff:fe00.
0	0	3	No address is provided. The first 64 bits should be obtained from the link-local prefix, padded with zeroes. The remaining 64 bits should be obtained from the lower layer headers, e.g. 802.15.4.
0	1	0	Reserved
0	1	1	Last 64 bits of the IPv6 address is provided. The first 64 bits should be derived from context information. Any remaining bits (not covered by the context) are 0.
0	1	2	Last 16 bits of the IPv6 address is provided. The first 112 bits should be derived from context information. Any remaining bits in between (not covered by the context) are derived from 0000:00ff:fe00.
0	1	3	No address is provided. The address should be obtained from context information. Any remaining bits (not covered by the context) should be obtained from the lower layer headers, e.g. 802.15.4.
1	0	0	Full 128-bit IPv6 address is provided.
1	0	1	8+40 bits of the IPv6 address is provided. The address follows the format ffXX:0000:0000:0000:0000:00XX:XXXX:XXXX.
1	0	2	8+24 bits of the IPv6 address is provided. The address follows the format ffXX:0000:0000:0000:0000:0000:00XX:XXXX.
1	0	3	8 bits of the IPv6 address is provided. The address follows the format ff02:0000:0000:0000:0000:0000:0000:00XX.
1	1	0	16+32 bits of the IPv6 address is provided. The address follows the format ffXX:XXCC:CCCC:CCCC:CCCC:CCCC:XXXX:XXXX, where the values for C are derived from context information.
1	1	1	Reserved
1	1	2	Reserved
1	1	3	Reserved

Table C.2: 6LoWPAN IPHC destination address compression.

Bibliography

[1] Zigbee ip specification, February 2013.

[2] Ieee standard for information technology–telecommunications and information exchange between systems - local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pages 1–4379, 2021. doi: 10.1109/IEEESTD.2021.9363693.

[3] Nicola Accettura, Luigi Alfredo Grieco, Gennaro Boggia, and Pietro Camarda. Performance analysis of the rpl routing protocol. In *2011 IEEE International Conference on Mechatronics*, pages 767–772. IEEE, 2011.

[4] Cedric Adjih, Emmanuel Baccell, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, , and Thomas Watteyne. Fit iot-lab: A large scale open experimental iot testbed. 2015.

[5] Ahmet Aksoy, Sushil Louis, and Mehmet Hadi Gunes. Operating system fingerprinting via automated network traffic analysis. 2017.

[6] Paramasiven Appavoo, Ebram Kamal William, Mun Choon Chan, and Mobashir Mohammad. Indriya2: A heterogeneous wireless sensor network (wsn) testbed. 2019. doi: doi:10.1007/978-3-030-12971-2_1.

[7] Emmanuel Baccelli, Cenk Gundogan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wahlisch. Riot: an open source operating system for low-end embedded devices in the iot. page 12, 2018.

[8] Frédéric Beck, Olivier Festor, and Isabelle Chrisment. Ipv6 neighbor discovery protocol based os fingerprinting. Technical report, INRIA, 2007.

[9] Bruhadeshwar Bezawada, Indrakshi Ray, and Indrajit Ray. Behavioral fingerprinting of internet-of-things devices. 2019.

[10] Yongjun Wang Baokang Zhang Bofeng Zhang, Tiezheng Zou. Remote operation system detection base on machine learning. 2009.

[11] A Brandt, J Buron, and G Porcu. Home automation routing requirements in low-power and lossy networks", rfc 5826. 2010.

[12] A Brandt, Sigma Designs, J Hui, R Kelsey, P Levis, K Pister, R Struik, JP Vasseur, and R Alexander. Internet engineering task force (ietf) t. winter, ed. request for comments: 6550 category: Standards track p. thubert, ed. 2012.

[13] A Brandt, Sigma Designs, J Martocci, and Johnson Controls. Internet engineering task force (ietf) m. goyal, ed. request for comments: 6998 univ. of wisconsin milwaukee category: Experimental e. baccelli. 2013.

[14] AJ Charles and P Kalavathi. Qos measurement of rpl using cooja simulator and wireshark network analyser. *International Journal of Computer Sciences and Engineering*, 6(4):283–291, 2018.

[15] Maxim Chernyshev, Zubair Baig, Oladayo Bello, and Sherali Zeadally. Internet of things (iot): Research, simulators, and testbeds. 2018.

[16] Thang Vu Chien, Hung Nguyen Chan, and Thanh Nguyen Huu. A comparative study on operating system for wireless sensor networks. 2011.

[17] Thomas Clausen, Ulrich Herberg, and Matthias Philipp. A critical evaluation of the ipv6 routing protocol for low power and lossy networks (rpl). In *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 365–372. IEEE, 2011.

[18] Alex Conta, Stephen Deering, and Mukesh Gupta. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification, March 2006.

[19] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6), December 1998.

[20] Mischa Dohler, Thomas Watteyne, Tim Winter, and Dominique Barthel. Routing requirements for urban low-power and lossy networks. In *RFC 5548*, 2009.

[21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. 2004.

[22] S Dwars and T Phinney. Network working group k. pister, ed. request for comments: 5673 dust networks category: Informational p. thubert, ed. cisco systems. *Consultant*, 2009.

[23] Christoph Ellmer. Openthread vs. contiki ipv6: An experimental evaluation, 2017.

[24] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–7, 2009.

[25] Emre Ertin, Anish Arora, Rajiv Ramnath, Mikhail Nesterenko, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, and Hui Cao. Kansei: A testbed for sensing at scale. 2006.

[26] Seong eun Yoo, Poh Kit Chong, Jeonghwan Bae, Tae-Soo Kim, Hiecheol Kim, and Joonhyuk Yoo1. Multichannel packet-analysis system based on ieee 802.15.4 packet-capturing modules. 2014.

[27] Jerome Francois, Radu State, and Thomas Engel. Enforcing security with behavioral fingerprinting. 2011.

[28] Jason Franklin, Damon McCoy, Parisa Tabriz, Vicentiu Neagoe, J Van Randwyk, and Douglas Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In *USENIX Security Symposium*, volume 3, pages 16–89, 2006.

[29] Olfa Gaddour and Anis Koubâa. Rpl in a nutshell: A survey. *Computer Networks*, 56(14):3163–3178, 2012.

[30] Olfa Gaddour, Anis Koubaa, Shafique Chaudhry, Miled Tezeghdanti, Rihab Chaari, and Mohamed Abid. Simulation and performance evaluation of dag construction with rpl. In *Third international conference on communications and networking*, pages 1–8. IEEE, 2012.

[31] Francois Gagnon and Babak Esfandiari. A hybrid approach to operating system discovery based on diagnosis theory. 2012.

[32] Mukul Goyal, Emmanuel Baccelli, Matthias Philipp, Anders Brandt, and Jerry Martocci. Reactive discovery of point-to-point routes in low power and lossy networks. *IETF Request For Comments RFC*, 6997, 2013.

[33] Lixia Guan, Koojana Kuladinithi, Thomas Pötsch, and Carmelita Goerg. A deeper understanding of interoperability between tinyrpl and contikirpl. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.

[34] Ulrich Herberg and Thomas Clausen. Study of multipoint-to-point and broadcast traffic performance in the “ipv6 routing protocol for low power and lossy networks. *Journal of Ambient Intelligence and Humanized Computing*, 2(4):293–305, 2011.

[35] Jonathan Hui, Pascal Thubert, et al. Compression format for ipv6 datagrams over ieee 802.15. 4-based networks. 2011.

[36] Low-Energy Critical Infrastructure and Monitoring LECIM Physical Layer. Ieee standard for low-rate wireless networks.

[37] Oana Iova, Pietro Picco, Timofei Istomin, and Csaba Kiraly. Rpl: The routing standard for the internet of things... or is it? *IEEE Communications Magazine*, 54(12):16–22, 2016.

[38] Jiang Lu Ishaq Unwala, Zafar Taqvi. Thread: An iot protocol. 2018.

[39] Timofei Istomin, Csaba Kiraly, and Gian Pietro Picco. Is rpl ready for actuation? a comparative evaluation in a smart city scenario. In *European Conference on Wireless Sensor Networks*, pages 291–299. Springer, 2015.

[40] Rahul Jadhav, Pascal Thubert, Rabi Narayan Sahoo, and Zhen Cao. Efficient route invalidation. RFC 9009, April 2021. URL `https://www.rfc-editor.org/info/rfc9009`.

[41] Hossein Jafari, Oluwaseyi Omotere, Damilola Adesina, Hsiang-Huang Wu, and Lijun Qian. Iot devices fingerprinting using deep learning.

[42] Farhana Javed, Muhamamd Khalil Afzal, Muhammad Sharif, and Byung-Seo Kim. Internet of things (iots) operating systems support, networking technologies, applications, and challenges: A comparative review. 2018.

[43] Ira Ray Jenkins, Rebecca Shapiro, Sergey Bratus, Ryan Speers, and Travis Goodspeed. Fingerprinting ieee 802.15. 4 devices with commodity radios. 2014.

[44] Patrick Olivier Kamgueu, Emmanuel Nataf, and Thomas Djotio Ndie. Survey on rpl enhancements: a focus on topology, security and mobility. 2018. doi: doi:10.1016/j.comcom.2018.02.011.

[45] Hamidreza Kermajani and Carles Gomez. On the network convergence process in rpl over ieee 802.15. 4 multihop networks: Improvement and trade-offs. *Sensors*, 14(7):11993–12022, 2014.

[46] Hyung-Sin Kim, JeongGil Ko, David E. Culler, and Jeongyeup Paek. Challenging the ipv6 routing protocol for low-power and lossy networks (rpl): A survey. 2017.

[47] JeongGil Ko, Stephen Dawson-Haggerty, Omprakash Gnawali, David Culler, and Andreas Terzis. Evaluating the performance of rpl and 6lowpan in tinyos. In *Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN)*, volume 80, pages 85–90. Citeseer, 2011.

[48] Jeonggil Ko, Joakim Eriksson, Nicolas Tsiftes, Stephen Dawson-Haggerty, Andreas Terzis, Adam Dunkels, and David Culler. Contikirpl and tinyrpl: Happy together. In *Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN)*, volume 570. Citeseer, 2011.

[49] JeongGil Ko, Joakim Eriksson, Nicolas Tsiftes, Stephen Dawson-Haggerty, Jean-Philippe Vasseur, Mathilde Durvy, Andreas Terzis, Adam Dunkels, and David Culler. Industry: Beyond interoperability: Pushing the performance of sensor network ip stacks. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 1–11, 2011.

[50] Sukant Kothari, T.S. Shri Krishnan, Jemimah Ebenezer, and S.A.V. SatyaMurty. Development of network debugging tools for ieee 802.15.4 networks. 2015.

[51] Hanane Lamaazi and Nabil Benamar. A comprehensive survey on enhancements and limitations of the rpl protocol: A focus on the objective function. 2019. doi: doi:10.1016/j.adhoc.2019.102001.

[52] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[53] P. Levis, T. Clausen, J. Hui, O. Gnawali, and J. Ko. The trickle algorithm, March 2011.

[54] Nguyen Thanh Long, Niccolò De Caro, Walter Colitti, Abdellah Touhafi, and Kris Steenhaut. Comparative performance study of rpl in wireless sensor networks. 2012.

[55] Arif Mahmud, Faria Hossain, Tasnim Ara Choity, and Faija Juhin. Simulation and comparison of rpl, 6lowpan, and coap protocols using cooja simulator. *Proceedings of International Joint Conference on Computational Intelligence*, page 317–326, 2020. doi: doi:10.1007/978-981-13-7564-4_28.

[56] S. Mehta, Mst. Najnin Sulatan, H.Kabir, N.Ullah, and K. S. Kwak. Network and system simulation tools for next generation networks: A case study. 2009.

[57] De Montigny-Leboeuf et al. A multi-packet signature approach to passive operating system detection. Technical report, DEFENCE RESEARCH AND DEVELOPMENT CANADAOTTAWA (ONTARIO), 2005.

[58] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor discovery for ip version 6 (ipv6), September 2007.

[59] Adrian Ordorica. Operating system identification by ipv6 communication using machine learning ensembles. 2017.

[60] Georgios Z Papadopoulos, Julien Beaudaux, Antoine Gallais, Thomas Noel, and Guillaume Schreiner. Adding value to wsn simulation using the iot-lab experimental platform. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 485–490. IEEE, 2013.

[61] Georgios Z Papadopoulos, Antoine Gallais, Guillaume Schreiner, Emery Jou, and Thomas Noel. Thorough iot testbed characterization: From proof-of-concept to repeatable experimentations. *Computer Networks*, 119:86–101, 2017.

[62] Aishwarya Parasuram, David Culler, and Randy Katz. An analysis of the rpl routing standard for low power and lossy networks. *Electrical Engineering and Computer Sciences University of California at Berkeley*, 2016.

[63] Pavan Pongle and Gurunath Chavan. A survey : Attacks on rpl and 6lowpan in iot. 2015.

[64] Wolf-Bastian Pottner and Lars Wolf. Ieee 802.15.4 packet analysis with wireshark and off-the-shelf hardware. 2012.

[65] Nelson Prates, Andressa Vergütz, Ricardo T Macedo, Aldri Santos, and Michele Nogueira. A defense mechanism for timing-based side-channel attacks on iot traffic. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[66] N Riou and W Vermeylen. Internet engineering task force (ietf) j. martocci, ed. request for comments: 5867 johnson controls inc. category: Informational p. de mil. 2010.

[67] Kévin Roussel, Ye-Qiong Song, and Olivier Zendra. Using cooja for wsn simulations: Some new uses and limits. In *EWSN 2016—NextMote workshop*, pages 319–324. Junction Publishing, 2016.

[68] Josep Sabater, Jose Maria Gomez, and Manel Lopez. Towards an ieee 802.15.4 sdr transceiver. 2010.

[69] Rajdeep Singh Shaktawat. Comparative analysis of wireless operating system. *International Journal of Science and Research*, 2015.

[70] Sharwari S Solapure, Harish H Kenchannavar, and Ketki P Sarode. Issues faced during rpl protocol analysis in contiki-2.7. In *ICT Systems and Sustainability*, pages 477–485. Springer, 2020.

[71] Roman Trüb, Reto Da Forno, Lukas Sigrist, Lorin Mühlebach, Andreas Biri, Jan Beutel, and Lothar Thiele. Flocklab 2: Multi-modal testing and validation for wireless iot. 2020.

[72] A Selcuk Uluagac, Sakthi V Radhakrishnan, Cherita Corbett, Antony Baca, and Raheem Beyah. A passive technique for fingerprinting wireless devices with wired-side observations. In *2013 IEEE conference on communications and network security (CNS)*, pages 305–313. IEEE, 2013.

[73] Aleksandar Velinov and Aleksandra Mileva. Running and testing applications for contiki os using cooja simulator. 2016.

[74] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. 2005.

[75] Yan Yan, Elisabeth Oswald, and Theo Tryfonas. Exploring potential 6lowpan traffic side channels. *IACR Cryptol. ePrint Arch.*, 2017:316, 2017.

[76] Adeel Yaqoob, Ahmad Hassan Butt, Muhammad Adeel Ashraf, and Yaser Daanial Khan. Wsn operating systems for internet of things(iot): A survey. 2019.

[77] Wei Yuan, Xiangyu Wang, Jean-Paul M. G. Linnartz, and Ignas G. M. M. Niemegeers. Coexistence performance of ieee 802.15.4 wireless sensor networks under ieee 802.11b/g interference. 2013.

[78] Ed. Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann. Neighbor discovery optimization for ipv6 over low-power wireless personal area networks (6lowpans), November 2012.

[79] Tao Zhang and Xianfeng Li. Evaluating and analyzing the performance of rpl in contiki. In *Proceedings of the first international workshop on Mobile sensing, computing and communication*, pages 19–24, 2014.

[80] Zhonghua Zhao, Wei Huangfu, and Linmin Sun. Nssn: A network monitoring and packet sniffing tool for wireless sensor networks. 2012.