Some Security Considerations over Contiki-based Sensor Network

Yan Yan

December 18, 2015

Contents

Chapter 1

Introduction

Contiki OS is an embedded system that used to build WSN[1] based on 802.15.4[4] compatible devices and 6LowPAN[5]. This paper discusses two security measurements, namely Link Layer Security (LLSEC) and Datagram TLS (DTLS), within Contiki OS. We also discuss some potential methods of fingerprinting an application running on a sensor node.

In Chapter 2 we describes a LLSEC implementation in Contiki OS called *noncoresec* and argues that it does not met certain cryptographic security notions.

Chapter 3 discusses some implementation issues of DTLS on Contiki OS.

Chapter 4 first argues that under some natures of WSN, an adversary could possibly collect more accurate timing information than usual Internet Web-application attacker. Then we describe a potential side-channel attack that fingerprints an application the target sensor node is running, using interactions between PING protocol and the application running on the target node.

1.1 Related Work

[1] discusses some security concerns in 802.15.4. LLSEC[2] is the implementation of 802.15.4 security in Contiki.

tinydtls[3] is the implementation of DTLS we used in DTLS related experiments.

To our knowledge, this is the first work of application fingerprinting through traffic analysis over wireless sensor network.

[1]Wireless Sensor Network

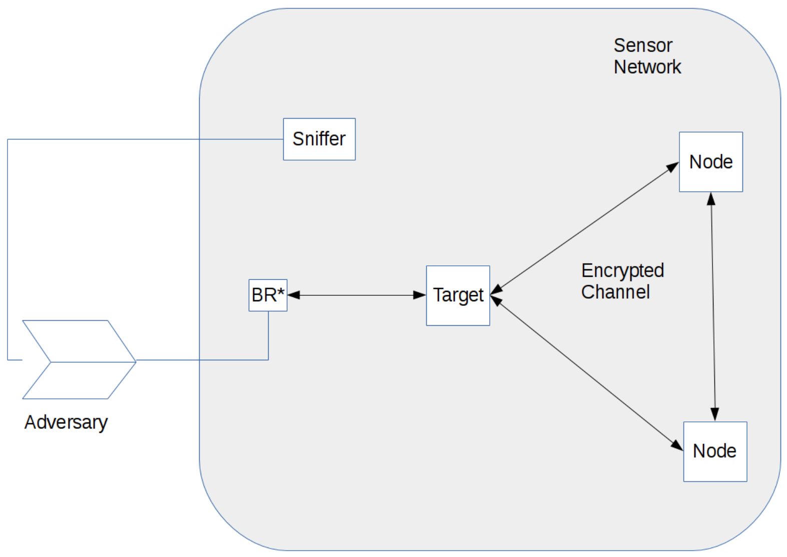


Figure 1.1: Experiment setup

1.2 Experiment Setup

All experiments are done within the Cooja simulator. The environment we simulated is as described in Figure 1.1.

- **Adversary** is a malicious party that tries to illegally reveal information from the encrypted traffic.
- **Border Router**, or BR, is a device that connects adversary to the sensor network. **BR is not allowed when LLSEC is enabled** as the adversary does not have the key and hence cannot connect into the network. We will discuss more about LLSEC in Chapter 2.
- **Sniffer** passively captures all traffic in the network.
- **Target** and **Nodes** are sensors deployed in the sensor network. They communicates to each other through encrypted channels.
- **Sensor Network** discussed in this paper is a 6LowPAN network based on Contiki OS.

Realistically speaking, this scenario could happen say an adversary sitting near a smart house with a laptop attached to a SoC[2] device, or your malicious neighbour walks into your

[2]System on Chip

smart house with her smart phone.

1.3 Adversary Power

The powers assumed in the experiments are considered to be practical in real life.

When LLSEC is enabled, all traffic, including RPL[3] messages, are encrypted; therefore no external nodes can connect to the network as an external node cannot send any valid RPL messages to join the network. The adversary only passively sniffs all traffic.

With LLSEC disabled, the adversary can therefore join the sensor network through a BR and hence is also capable to send messages to the target(s). However, she will not be able to inject any message into an encrypted channel such as a DTLS channel.

1.4 Types of Packets

We simply categorise the packets into two types:

- **Network Management Packets**: These are the packets generated by the protocols to maintain the functionality of network, such as MAC ACKs, RPL messages or ICMP messages.
- **Data Packets**: These are those packets generated by applications running on sensor nodes., such as a CoAP packet.

This is only a subjective rough categorisation and may not be precise. For example an TCP data packet may set its ACK flag, or DTLS handshake packets could ambiguously fall into both categories. However, we ignore this ambiguity as it is not our focus.

[3]Routing Procol for Low-power and Lossy Networks

Chapter 2

Link Layer Security

Link Layer Security, or LLSEC, is a security measure that implements cryptography at Data Link Layer[1] which is only above Physical Layer.

Introducing cryptography at a lower level has several benefits. Firstly, more data being encrypted reduces the observable packet features to an adversary, such as SRC[2] and DST[3] field in the IP header which are very likely to be exploited by an adversary. Secondly, authentication at lower level also prevents an active adversary from joining the network which therefore weakens her power.

On the other hand, imposing cryptography at a lower level also brings more challenge to the design of sensor network architecture. The first problem is its overhead. For example, even for a node that only forwards a packet to its next hop, it must decrypt the whole packet to extract its routing information, and then re-encrypt it before transmission. This is particularly problematic in a mesh wireless sensor network as it could potentially downgrades the performance and causing energy consumption problems. More over, key management is also challenging due to the constrained computational power and power optimised lossy nature of wireless sensor network.

It is noticeable that some packet features are not hidden even with LLSEC enabled, such as packet length, timing information and part of the MAC header in a 802.15.4 packet.

2.1 802.15.4 Security: *noncoresec*

noncoresec[2] is the current implementation of LLSEC in Contiki. It implements AES_CCM_16 ciphersuite in 802.15.4 standard. This section briefly describes how it works.

[1]https://en.wikipedia.org/wiki/OSI_model

[2]Source Address

[3]Destination Address

Flags(1)	Addresses(8)	Frame Counter(4)	Security Level(1)	Block Counter(2)

Figure 2.1: IV of 802.15.4 Frame with Security

- **Key Management**: All nodes share a network wide AES key for both encryption and authentication. The key is hardcoded during the setup stage.
- **AEAD**[4]: *noncoresec* implements AES_CCM_16 [5] as described in 802.15.4[4] which turns AES into a stream cipher. The same key is used for both encryption and authentication.
- **Initial Vector (IV, or nonce)**: The IV for each packet is constructed from certain fields of unencrypted MAC header and therefore is public.

An adversary without the knowledge cannot join the sensor network protected by *noncoresec* as she cannot sent out a valid RPL message.

2.2 Weak IV

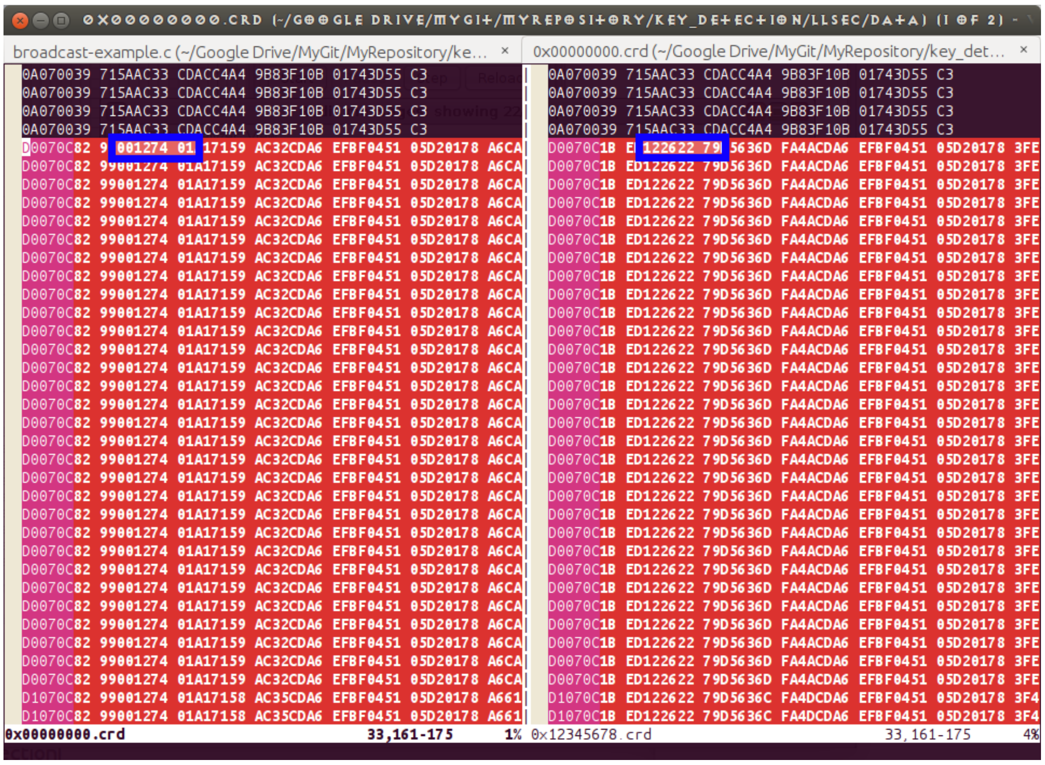
One problem within the *noncoresec* implementation is the low variance of IV. The IV is a 16 byte bit-string constitutes of the following fields(Figure 2.1):

- **Flags (1 byte)**: This field contains part of the MAC header. It is identical to most (basically all) of the data packets.
- **Source Address (8 bytes)**: This is mapped from the source address field of the packet.
- **Frame Counter (4 bytes)**: This field increases by 1 from 0 for each packet sent to prevent replay attack.
- **Security Level (1 byte)**: This field indicates which ciphersuite to be used for this packet. In the case of AES_CCM_16, this is constantly 0x7.
- **Block Counter (2 bytes)**: This field begins from 0x0 and increases by 0x1 for each block in CCM mode. The block length for AES-128 is 16 bytes. The 2 bytes counter is usually sufficient as it supports up to 2^{32} bytes of data whereas the minimum MTU[6] required by 6lowPAN standard[5] is 127 bytes.

[4]Authenticated Encryption with Associated Data

[5]CCM mode of AES-128 with 16 bytes MAC

[6]Maximum Transmit Unit, simply speaking this is the maximum length of a packet.



0x00000000.crd (~/Google Drive/MyGit/MyRepository/key_detection/llsec/data) (1 of 2) -

broadcast-example.c (~/Google Drive/MyGit/MyRepository/ke…

```
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D0070C82 99001274 01A17159 AC32CDA6 EFBF0451 05D20178 A6CA
D1070C82 99001274 01A17158 AC35CDA6 EFBF0451 05D20178 A661
D1070C82 99001274 01A17158 AC35CDA6 EFBF0451 05D20178 A661
0x00000000.crd                          33,161-175       1%
```

0x00000000.crd (~/Google Drive/MyGit/MyRepository/key_det…

```
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
0A070039 715AAC33 CDACC4A4 9B83F10B 01743D55 C3
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D0070C1B ED122622 79D5636D FA4ACDA6 EFBF0451 05D20178 3FE
D1070C1B ED122622 79D5636C FA4DCDA6 EFBF0451 05D20178 3F4
D1070C1B ED122622 79D5636C FA4DCDA6 EFBF0451 05D20178 3F4
0x12345678.crd                          33,161-175       4%
```

Figure 2.2: Captured packets with *noncoresec* enabled

In the current *noncresec* implementation, **Flags** and **Security Level** are constant. **Block Counter** always begins from 0x0 and the **Source Address** is also constant for a specific device. Such design leaves the 4 bytes **Frame Counter** the only field that is variable. This indicates that only 2^{32} messages are allowed without a collision of IV which is cryptographically considered to be inappropriate.

2.2.1 Reset Problem

The low variance of IV leads to a plaintext leakage problem which only requires the adversary to reboot the target node.

The idea is that rebooting the device resets the **Frame Counter** to 0x0; hence once a pair of packets with same **Frame Counter** is found, the difference of their plaintext can be computed by their ciphertext:

$$\Delta p = c_1 \oplus c_2$$

where Δp is the difference of plaintexts. c_1 and c_2 are their ciphertext respectively.

Example 2.2.1. Figure 2.2 demonstrates some packet captured[7] with *noncoresec* enabled. These packets are captured with a sensor broadcasting a 4 byte integer with left side of Figure 2.2 being $[00000000]_{16}$ and right $[12345678]_{16}$. Marked are the corresponding ciphertexts which are $[00127401]_{16}$ and $[12262279]_{16}$ respectively.

As we can see, the difference of ciphertext is exactly the difference of plaintext:

$$\Delta p = [00127401]_{16} \oplus [12262269]_{16} = [12345678]_{16} \tag{2.1}$$

2.3 Distinctive Packet Length for RPL Packets

Some RPL packets are shorter than the minimum length of data packets which can be used to distinguish the packets. Further more, some RPL packets set MAC header flags differently from data packets.

2.4 Performance Issue

The header overhead with LLSEC enabled is 20 bytes which is relatively a large overhead comparing to the 127 bytes MTU requirement of 6LowPAN standard[5].

[7]The duplicated packets are caused by the retransmission of ContikiMAC[6].

Chapter 3

DTLS

DTLS has potentially the best interoperability as it is an variation of the widely used TLS in Internet. However, its design might not fit into the nature of WSN for practical reasons.

3.1 Implementation Issues

The most practical implementation we found on Contiki OS so far is tinydtls.

tinydtls[3] currently supports two ciphersuites, namely TLS_PSK_WITH_AES_128_CCM_8 and TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8.

However, we encountered several difficulties when trying to set up an encrypted network using tinydtls.

Low Computational Power
: Curve computation requires relatively a large amount of computational power. Even using a relatively powerful platform (CC2538), it still takes minutes to complete a DTLS handshake with TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8.

Low Bandwidth
: The 6LowPAN standard specifies that the minimum MTU is 127 bytes whilst 67 (87 with LLSEC) bytes are occupied by protocol headers until UDP, which leaves 60 (40 with LLSEC) bytes available for UDP layer payload. This value has been exceeded by several handshake packets even with pre-shared keys. Doing key exchange or even using longer keys only makes this problem worse. Some attempts have been made to solve this issue, e.g. CoDTLS[7][1]. As a result, DTLS is only available on those devices support extra frame length than 6LowPAN requirements.

[1]This draft has been abandoned for some reason we do not know.

Code Size

The tinydtls fails to fit into some devices, e.g. skymote, as its size of code is too large.

Therefore although TLS_PSK_WITH_AES_128_CCM_8 is less flexible (and probably less secure) as it uses a pre-shared master secret, it is still considered to be a relatively practical security measure as it requires less resources.

3.2 No Multicast Support

Some application protocols, such as CoAP, utilises the multicast feature of 6LowPAN whilst TLS is a protocol designated for securing communications between two parties, so is DTLS. To our knowledge, DTLS does not make any attempt to support multicasting.

3.3 Overloading DTLS with LLSEC

Adopting both security measures at the same time is possible as they are implemented at different layers. However, it is questionable whether this will bring more security, as both *noncoresec* and TLS_PSK_WITH_AES_128_CCM_8 are using 128 bit AES with CCM mode as their cryptographic primitive.

Chapter 4

Application Detection

Similar to website fingerprinting, we try to identify the application running on target note by its traffic. This chapter discusses some general idea without a specific application.

4.1 Network Protocol Headers

Since most contents of MAC[1], IP and UDP headers are related to routing and network maintenance, they are independent to the application data except length fields and CRC[2].

4.2 Packet Length

Packet length is usually the most interested target in traffic analysis. Packet lengths are also highly application dependant; thus we are not pursuing this topic further without a specific application.

4.3 Timing Packet Response Intervals

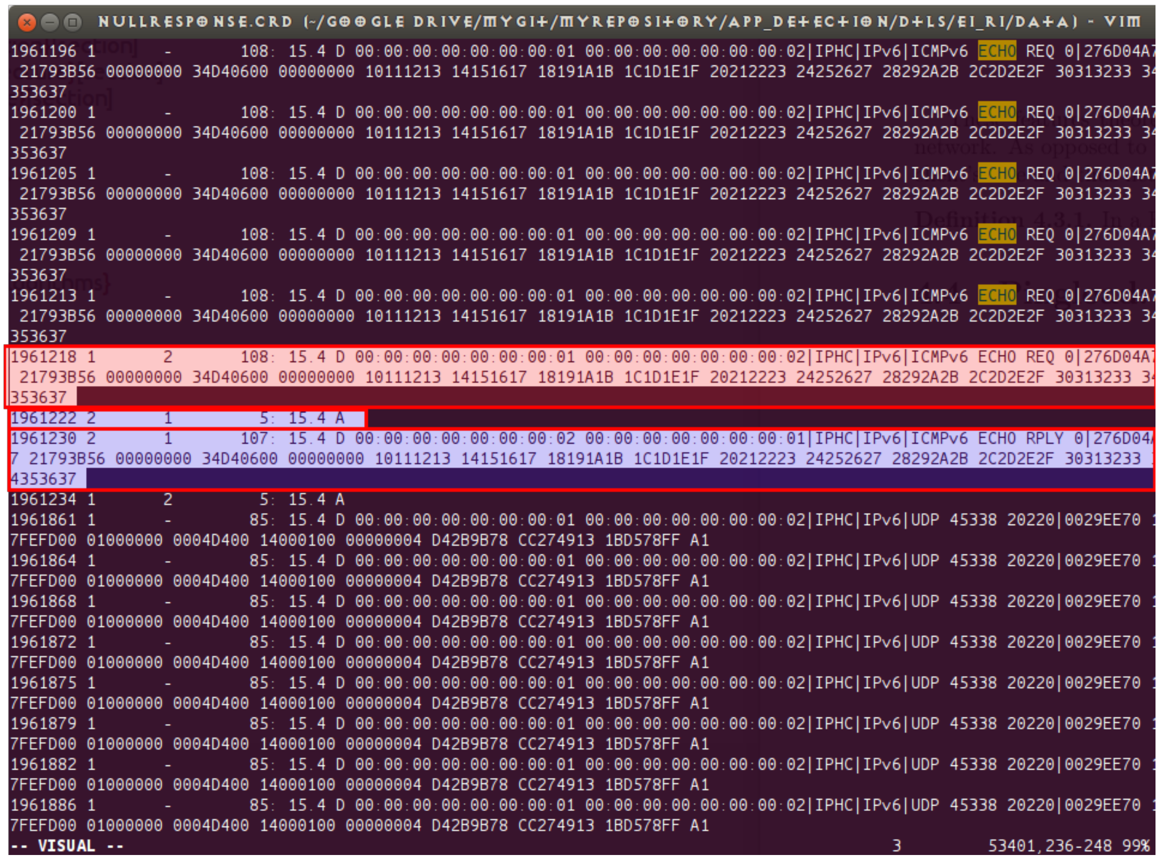
Unlike web applications where the client and server are usually physically distant, sensor networks can sometimes located in a concentrated area, such as a smart house which its radius can be less than 10 meters.

These features theoretically enables one to capture all traffics in such a sensor network. As opposed to the case of Internet where packets are usually timed on the client's side and thus network latency (RTT[3]) must be concerned, being able to capture all traffic in the

[1]Media Access Control, not to be confused with the cryptographic term Message Authenticate Code.

[2]Cyclic Redundancy Check, a code to detect or correct transmission error

[3]Round-Trip Time



```
NULLRESPONSE.CRD (~/GOOGLE DRIVE/MYGIT/MYREPOSITORY/APP_DETECTION/DTLS/EI_RI/DATA) - VIM
1961196 1       -       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961200 1       -       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961205 1       -       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961209 1       -       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961213 1       -       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961218 1       2       108:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|ICMPv6 ECHO REQ 0|276D04A7
 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233 34
353637
1961222 2       1         5:  15.4 A
1961230 2       1       107:  15.4 D 00:00:00:00:00:00:00:02 00:00:00:00:00:00:00:01|IPHC|IPv6|ICMPv6 ECHO RPLY 0|276D04
7 21793B56 00000000 34D40600 00000000 10111213 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233
4353637
1961234 1       2         5:  15.4 A
1961861 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961864 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961868 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961872 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961875 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961879 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961882 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
1961886 1       -        85:  15.4 D 00:00:00:00:00:00:00:01 00:00:00:00:00:00:00:02|IPHC|IPv6|UDP 45338 20220|0029EE70
7FEFD00 01000000 0004D400 14000100 00000004 D42B9B78 CC274913 1BD578FF A1
-- VISUAL --                                                                              3          53401,236-248 99%
```

Figure 4.1: Capture of a ping packet

Time (ms)	From (ID)	To (ID)	Length (bytes)	Type
1961218	1	2	108	ICMP ECHO
1961222	2	1	5	802.15.4 ACK
1961230	2	1	107	ICMP ECHO

Table 4.1: Packet Features of an ICMP ECHO request and response

network provides much more accurate timing information and hence may be exploited to develop more efficient attacks.

Definition 4.3.1. In a Request-Response application model, **RI**, **Response Interval**, is defined as the interval between a request packet being received and its response being sent.

Example 4.3.1. Three packets are marked out in Figure 4.1 which forms an instance of ICMP ECHO[8] (also known as PING) session. The extracted packet features are displayed in Table 4.1.

Explanation of the Packets:

> The first packet is an ICMP ECHO request and the third packet being its response. The second packet is a 802.15.4 ACK[4] and is thus transparent to the upper ICMP protocol.

From this example we can see that the RI for this PING session is:

$$1961230 - 1961218 = 12(\text{ms})$$

Timing information can be exploited by several attacks, such as [9] and [10].

We have experimentally measured a RNG[5] call on Wismote platform in the Cooja simulator that is roughly 0.03 ms.

4.4 PINGLOAD: PING side-channel for Payload

Support of PING is required by [8] and is also enabled in Contiki OS by default. However, our experimental results showed that the RIs of these PING packets could potentially be exploited to reveal the application running on target sensor node.

We call such technique **Application Fingerprinting**.

4.4.1 Hypothesis

A phenomenon we realised is that when a PING packet arrived while the target node is executing some payload, say reading a sensor or processing data, the PING RIs begin to vary comparing to a stable value when no payload is given to the sensor node.

Example 4.4.1. Figure 4.2 shows PING RIs collected in two experiments. In the upper half, the target node is constantly in asleep whilst in the lower half, it occasionally receives a request which triggers the target to call RNG. We can see that the PING RI varies alongside the target is given some payload from this figure.

The data shown in Figure 4.2 suggests that the "plain", that is without any interference, PING RI is 12 to 13 ms. Those variations of shown in the lower half of Figure 4.2 is potentially caused by the payload on target.

This result inspires us that the distributions of PING RIs might vary according to the payload on target and could possibly considered as a fingerprint to the target's application.

[4]This is an acknowledgement from the receiver that notifies the sender that the packet has been received.

[5]Random Number Generator. In Contiki OS, this corresponds to the library function random_rand().

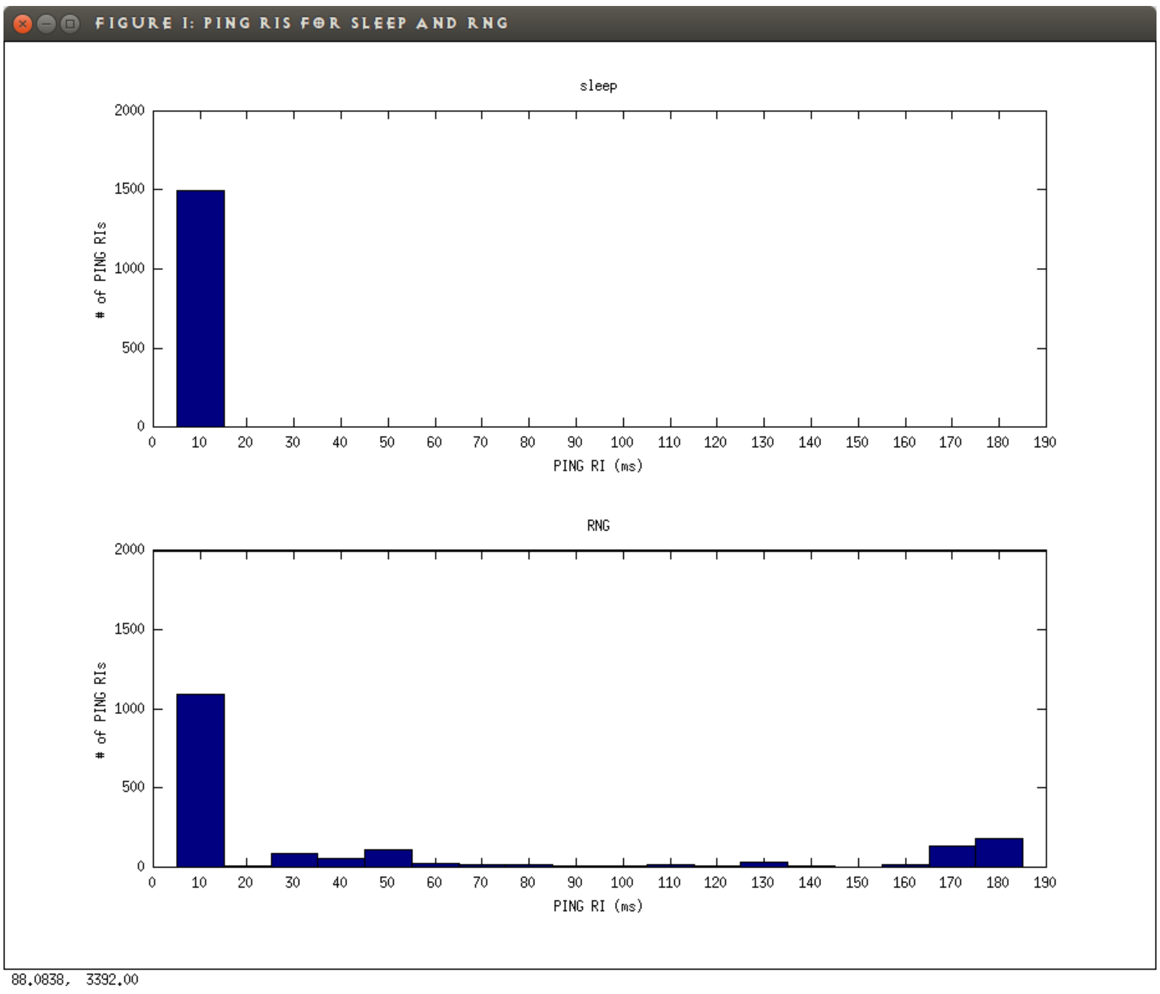


Figure 4.2: An example of PING RIs with different payload

In other word, an adversary could possibly tell whether the target is running a specific application by looking at its PING RIs distribution.

The attack then is strait forward:

Profile sleep RIs :
The PING RIs for a sleeping node of the same platform can be profiled by pinging a sleeping node. We denote the sleeping profile as RI_{sleep}.

Fingerprint application :
The adversary collects PING RIs on a profiling node with known application. The profiling node needs to be of the same platform and executing the same code of the target's. The profiled application fingerprint is a sample set of RIs, denotes as: $F_p = \{a_1, a_2, ..., a_n\}$ where a_i is the RI of ith PING to the profiling node.

Collect fingerprint of target :
The adversary then fingerprints the payload on the target in a similar way by pinging it. We denote the collected target sample set of RIs as: $F_t = \{b_1, b_2, ..., b_m\}$ where b_j is the RI of jth PING to the target node.

Extract Featured RIs : Since a node is usually in a sleep state, most of the PING RIs will hence results into RI_{sleep}. To improve the clarity of our fingerprint, we can remove them from the data sets and keep only the PING RIs those (seemingly) has been interfered by the payload. We denote the extracted RIs as **Featured RIs**:

$$F'_p = F_p - RI_{sleep} = \{x | x \in F_p, x \notin RI_{sleep}\}$$
$$F'_t = F_t - RI_{sleep} = \{x | x \in F_t, x \notin RI_{sleep}\}$$

Practically speaking, the PING protocol are designed to be responded immediately for diagnosis purpose; hence RI_{sleep} usually has an extremely low variance and its mean is also much less than F_p and F_t. Therefore we can ignore the error induced by wrongly removed packets.

Using the Featured RIs not only provides a better vision of the fingerprint but also removes the error caused by different frequency of the target code being executed, as all the Featured RIs are collected when the node is at a non-sleeping state.

Estimate Distribution (Optional) :
We then estimate the distributions of F'_p and F'_t, denote as $\mathbb{D}_p$ and $\mathbb{D}_t$. A naive method is to simply use their histograms. An example of such histograms are shown as Figure 4.3.

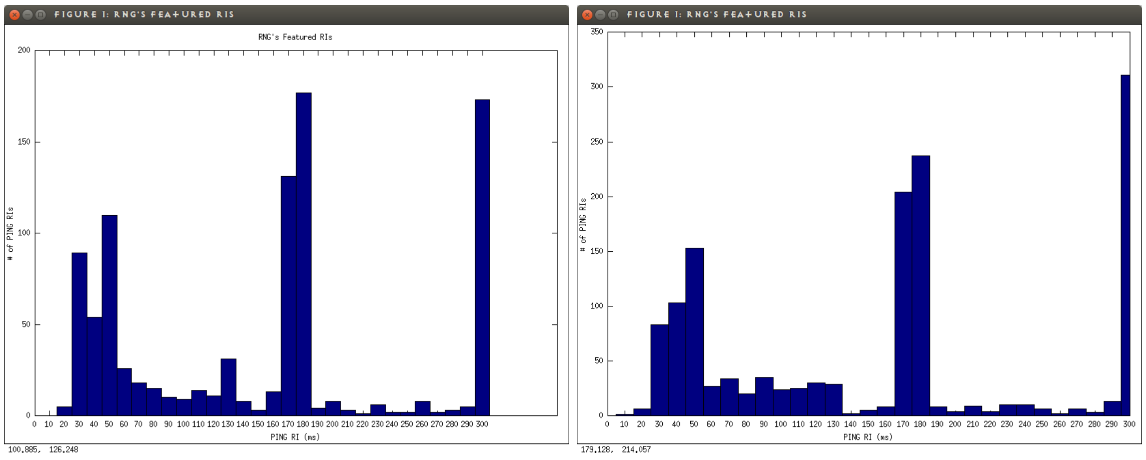


Figure 4.3: Two examples of RNG's Featured RIs histogram

Distinguish Distributions :

Finally we test whether D_t and D_p are the same distribution. A naive way is to compute the correlation of counts of the histograms. We conclude the target node is running the profiled application if D_t and D_p are the same distribution.

Practically speaking, the key point of Application Fingerprinting is to test whether the target's Featured RIs is sampled from same distribution of the profiled one, i.e. whether $F'_t = F'_p$; therefore estimating their distribution might not be necessary for some statistical methods such as t-tests. However as we can see in Figure 4.3, PING RIs' distribution are very unlikely to be normalised. Therefore a future work is to find a better distinguishing method than the current naive one.

4.4.2 Experiment Results

We tried our Application Fingerprinting attack in a Cooja simulated Wismote platform with different codes. **In conclusion, the fingerprint appears to be effective for certain circumstances but will tends to result into false positives as the profiled application and target application gets similar to each other.**

To be more specifically, the target node execute some specific code upon receiving an application layer protocol request, similar to CoAP. Further more, all traffic are protected by DTLS with TLS_PSK_WITH_AES_128_CCM_8 ciphersuite. Both intervals of PINGs and the application request are set to some asynchronised value to avoid overflooding the target and to create a 'more realistic' simulation.

Everything other than the examined code are the same for all experiments. Two samples are collected independently for each code to simulate a fingerprinting scenario. The histograms are clustered by 5ms from 0ms to 500ms.

Correlations	i=50	i=100	i=2500	i=5000
i=50	**0.988**	0.891	-0.014	-0.033
i=100	0.891	**0.973**	-0.025	-0.042
i=2500	-0.014	-0.025	**0.993**	-0.035
i=5000	-0.033	-0.042	-0.035	**0.985**

Table 4.2: Correlations for RNG

Correlations	+	*	%
+	**0.990**	**0.990**	**0.988**
*	**0.990**	**0.989**	**0.985**
%	**0.988**	**0.985**	**0.984**

Table 4.3: Correlations for word arithmetic operations

We examined two classes of codes:

RNG Calls :
The target node repeatedly calls RNG for i times. We examined their Featured PING RI for different values of i. The reason for picking RNG is that on some platforms where a hardware RNG is provided, the call to it is expected to be similar to a call to a sensor reading which is actually an interrupt to the processor. Results are shown in Table 4.2.

Arithmetic Operations :
The target node repeatedly does arithmetic operations, namely addition, multiplication and modular, on two randomly generated word size integers for 10000 times. This class is particularly interested from a cryptographic point of view as the number of arithmetic operations could potentially developed to key recovering attacks. Results are shown in Table 4.3.

We also computed the correlation for $i = 50$ and addition, as shown in Table 4.4.
The results suggests the following conjectures:

1. The results for RNG suggests that the fingerprinting is effective for this class of code, as the same code results into nearly perfect correlations (≥ 0.95).

Correlations	+
i=50	0.877

Table 4.4: Correlation for $i = 50$ and addition

2. Even relatively slight changes can be detected, as we can see the correlation dropped to 0.891 alongside 50 iterations of RNG calls (50 RNG calls take about 1.4ms).

3. The results for arithmetic operations indicates that their fingerprint are unlikely to be distinguishable. There are two potential causes we have considered:

   (a) The differences between these operations are too small to be detected.

   (b) Experiment methodology error. Since the target node we used during the experiments call RNG twice upon each request to generate two operands whilst the word arithmetic operations have much lighter weigh comparing to RNG at magnitude level[6]; thus the fingerprint is dominated by RNG rather than word arithmetic operations. As a result, we can see that a relatively high correlation can be observed between word addition and 50 RNG calls as shown in Table 4.4.

4.4.3 A General Hypothetical Model: Black Box Model

Although the experiments supports the hypothesis that the variation of PING RIs is relevant to the application, it is not yet clear which factor exactly caused the variations.

Therefore we instead suggest that this phenomenon is a result of multi factors, including:

- Code being executed and whether it is preemptive or non preemptive,
- Memory usage, such as packet buffers, or
- Any other hardware/software conditions.

It is difficult to verify these factors as many of them requires strict synchronisation between devices and the PING RIs are only showing statistical features.

Therefore we propose the Black Box Model which we hope could support further research.

Definition 4.4.1. The **Black Box Model** models the target device as a stateful black box. A **state**, denotes as $\vec{S} =< \text{factor}_1, \text{factor}_2 ... >$, is an abstracted vector of multiple factors that could affect PING RIs, such as current code context or memory usage. $\mathbb{D}_{\vec{S}}$ is the distribution of PING RIs when the target node is at state $\vec{S}$.

We can redefine a state by multiple mutually exclusive substates, $\vec{S}_1$, $\vec{S}_2$ etc. Each substate has its own PING RI distribution $\mathbb{D}_{\vec{S}_1}$, $\mathbb{D}_{\vec{S}_2}$ etc. The redefinition process can further more be done recursively.

[6]A RNG call takes about 0.03ms where as an addition takes $\leq 2^{-20}$ms on the Wismote platform.

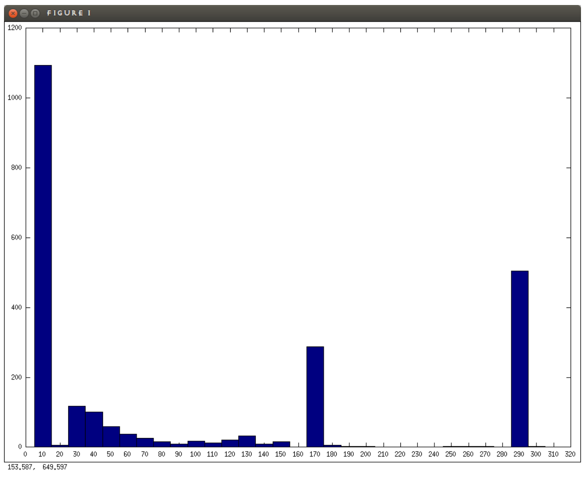


Figure 4.4: PING RIs of $\vec{S}_{root}$

Notice that since each substate are mutually exclusive; thus:

$$\forall x : Prob(RI = x|\vec{s} = \vec{S}) = \sum_{i=0}^{n} \left(p_i * Prob(RI = x|\vec{s} = \vec{S}_i)\right)$$

where $\vec{s}$ is the state at any moment, $p_i = Prob(\vec{s} = \vec{S}_i)$ and n is the number of substates of $\vec{S}$.

Therefore $\mathbb{D}_{\vec{S}}$ is a linear combination of all distributions associated with all the substates of $\vec{S}$. Hence we can write

$$\mathbb{D}_{\vec{S}} = \sum_{i}^{n} \left(p_i * \mathbb{D}_{\vec{S}_i}\right) \tag{4.1}$$

Example 4.4.2. Take one of the RNG applications with $i = 5000$ in Table 4.2 for example.

We define the root state:

$$\vec{S}_{root} =< \text{App} = \text{RNG}, i = 5000 >$$

Its associated PING RI distribution $\mathbb{D}_{\vec{S}_{root}}$ is approximated from the histogram shown in Figure 4.4.

We then redefine $\vec{S}_{root}$ to two exclusive sub-states depends on whether the target is in a sleeping mode.

$$\begin{aligned}
\vec{S}_{root} &= \vec{S}_{sleep} + \vec{S}_{nonsleep} \\
\vec{S}_{sleep} &=< \text{App} = \text{RNG}, i = 5000, \text{Code} = \text{sleep} > \\
\vec{S}_{nonsleep} &=< \text{App} = \text{RNG}, i = 5000, \text{Code} \neq \text{sleep} >
\end{aligned}$$

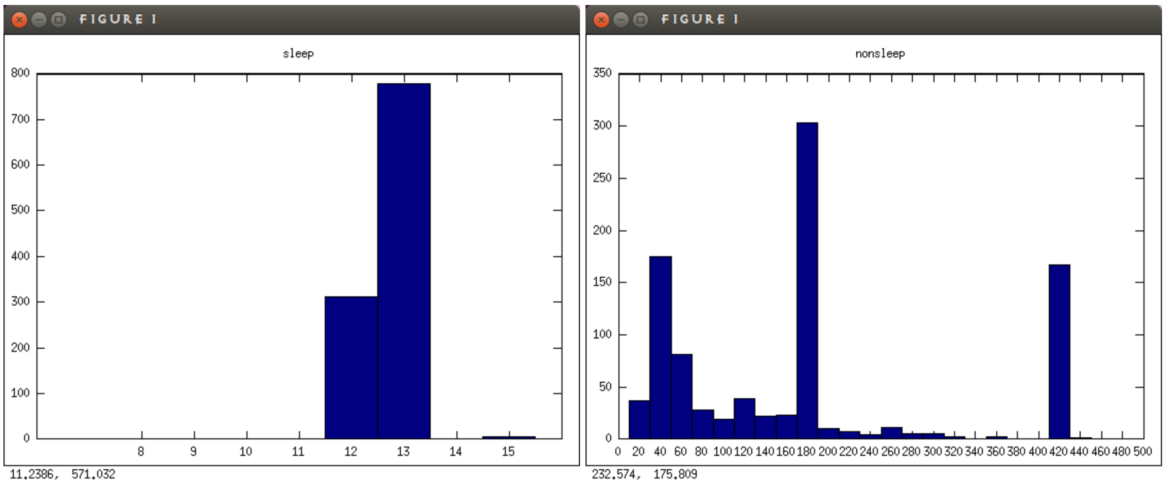


Figure 4.5: PING RIs of $\vec{S}_{sleep}$ and $\vec{S}_{nonsleep}$

Since we know by experiment that the PING RIs are 12 to 13ms when the target is sleep; therefore we can actually approximate $\mathbb{D}_{\vec{S}_{sleep}}$ by filter out those RIs in $[12, 13]$. Vice versa, the distribution of Featured RIs can be viewed as $\mathbb{D}_{\vec{S}_{nonsleep}}$ since they are exactly the RIs filtered by the RIs of sleep. The histograms are shown in Figure 4.5.

So we have

$$\mathbb{D}_{\vec{S}_{main}} = p_0 \mathbb{D}_{\vec{S}_{sleep}} + p_1 \mathbb{D}_{\vec{S}_{nonsleep}} \tag{4.2}$$

where p_0 and p_1 corresponds to the probability of the target being sleep and nonsleep.

Solving Equation (4.2) gives us roughly $p_0 = 0.538$ and $p_1 = 0.462$. Assuming the PING packets are received by the target randomly, we can estimate that the target is in sleep for 53.8% and awake for 46.2% of the time.

Theoretically we can further redefine more sub-states, e.g.:

$$\begin{aligned}
\vec{S}_{nonsleep} &= \vec{S}_{RNG} + \vec{S}_{header} \\
\vec{S}_{RNG} &=< \text{App} = \text{RNG}, i = 5000, \text{Code} = \text{random\_rand()} > \\
\vec{S}_{header} &=< \text{App} = \text{RNG}, i = 5000, \text{Code} = \text{header processing} >
\end{aligned}$$

Practically speaking, p_i might be an interesting information to an adversary as it reveals the internal state of the target which could lead to many information breach. In practice, we might only want to approximate the solutions of p_i, which effectively reduces the problem of approximating Equation (4.1) to a knapsack problem[11]. In most cases obtaining $\mathbb{D}_{\vec{S}_i}$ is difficult as it is hard to determine the exactly the state of target sensor node at an exact moment, as in Example 4.4.2. We leave this problem to future research.

Chapter 5

Conclusion

In Chapter 2 we described a LLSEC implementation in Contiki OS *noncoresec* and argues that its selection of IV is not secure enough and potentially has a reset problem.

Chapter 3 discusses some implementation issues of DTLS on Contiki OS.

Chapter 4 first argues that under some natures of WSN, an adversary could possibly collect more accurate timing information than usual Internet Web-application attacker. Then we described a potential sensor node application fingerprinting attack using the interaction of PING protocol and the application running on the target node.

Chapter 6

Fulture Work

1. All the results are based on Contiki OS with 6LowPAN. However, Zigbee[12] protocol is not supported by Contiki OS. Considering the popularity and market share, other OSes and Zigbee should also be studied.

2. DTLS using TLS_ECHDE_ECDSA_WITH_AES_128_CCM_8 ciphersuite is ignored due to performance issue. However, it should be reconsidered in the future as it provides better security. Further more, implementation other than tinyDTLS, or protocols other than DTLS should also be examined.

3. The security brought by *noncoresec* is not satisfying. A solution is expected, such as better key exchange algorithms, ciphersuites, or choice of IV.

4. Some RPL messages use specific MAC header flag which can be seen even with LLSEC enabled. These flags are yet to be studied in this paper.

5. RNG is being used for convenience for simulating sensor readings in a simulator. It should be replaced by real sensor readings with real devices.

6. Timing informations are all collected using Cooja simulator. The experiments should be re done with real devices.

7. In Section 4.4.1, we used a naive correlation test to test the fingerprints. This method might be improved by using nonparametric tests such as Kolmogorov-Smirnov test.

8. Some other code might be cryptographically interesting to Application Fingerprint, such as the double operations in ECC curve computation or Montgomery Reductions in RSA.

Bibliography

[1] Naveen Sastry and David Wagner. “Security Considerations for IEEE 802.15.4 Networks”. In: *Proceedings of the 3rd ACM Workshop on Wireless Security*. WiSe ’04. Philadelphia, PA, USA: ACM, 2004, pp. 32–42. ISBN: 1-58113-925-X. DOI: 10.1145/1023646.1023654. URL: http://doi.acm.org/10.1145/1023646.1023654.

[2] URL: https://github.com/kkrentz/contiki/wiki.

[3] URL: http://sourceforge.net/projects/tinydtls/.

[4] *IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks- Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*. Tech. rep. 2006, 0_1–305. DOI: 10.1109/ieeestd.2006.232110. URL: http://dx.doi.org/10.1109/ieeestd.2006.232110.

[5] G. Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944 (Proposed Standard). Updated by RFCs 6282, 6775. Internet Engineering Task Force, Sept. 2007. URL: http://www.ietf.org/rfc/rfc4944.txt.

[6] Adam Dunkels. *The ContikiMAC Radio Duty Cycling Protocol*. 2011.

[7] Carsten Bormann Lars Schmertmann Klaus Hartke. *CoDTLS: DTLS handshakes over CoAP*. 2015. URL: https://tools.ietf.org/html/draft-schmertmann-dice-codtls-01.

[8] R. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (INTERNET STANDARD). Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. Internet Engineering Task Force, Oct. 1989. URL: http://www.ietf.org/rfc/rfc1122.txt.

[9] K.P. Dyer et al. “Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. May 2012, pp. 332–346. DOI: 10.1109/SP.2012.28.

[10] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO ’96. London, UK, UK: Springer-Verlag, 1996, pp. 104–113. ISBN: 3-540-61512-1. URL: `http://dl.acm.org/citation.cfm?id=646761.706156`.

[11] URL: `https://en.wikipedia.org/wiki/Knapsack_problem`.

[12] ZigBee Alliance. *ZigBee specification*. Tech. rep. June 2005.