# tinyDTLS Experiment Record [Draft]

Yan Yan

March 17, 2015

# Contents

# Chapter 1

# Toys

In this hello-world set-up, there is only one server and one client connected through local-link. The protocol suit we adopted is: [IPv4 or IPv6] + UDP + DTLS.

All fiedls of UDP and DTLS header will be analysed.

**Abbreviations**:

**CLIENT** Client.

**SERVER** Server.

For the attack model, we define our adversary as a passive eavesdropper, i.e.

- The adversary is allowed to use a sniffer to capture all packets transmitted between CLIENT and SERVER.
- Only DTLS payload is encrypted, all other contents of the packets are captured in plaintext by the adversary.
- The adversary has specific pre-knowledge for each application, e.g. the set of contents that could be transmitted between CLIENT and SERVER, or some specific behaviour of CLIENT and SERVER.

## 1.1 Odd or Even

**Odd or Even** is an extremely simple toy application. It is designed to demonstrate the fundamental idea of traffic analysis.

### 1.1.1 Description

CLIENT randomly generates a 32-bit unsigned integer R and sends it to SERVER. SERVER replies with a string “ODD” or “EVEN” according to the integer sent(Figure 1.1).

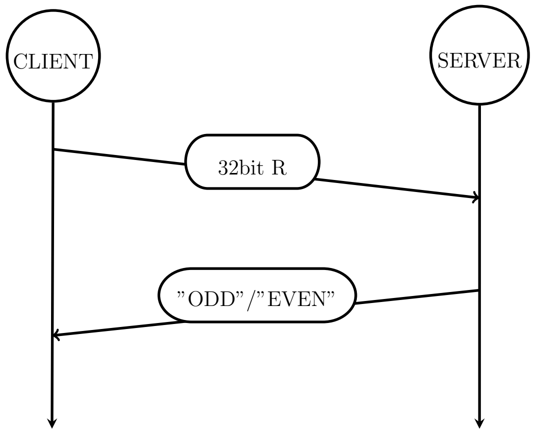


Figure 1.1: Description of an Odd-or-Even session

### 1.1.2 Analysis [To be completed...]

For every Odd-or-Even session,

Packets from CLIENT to SERVER:

All fields for every packet are the same, except: 1. Encrypted Application Data field in DTLS layer. 2. Sequence Number increased by 1 every packet. 3. Checksum in UDP layer.

Packets from SERVER to CLIENT:

All fields are the same for every packet except: 1. Encrypted Application Data field in DTLS layer. 2. Sequence Number increased by 1 every packet. 3. Checksum in UDP layer. 4. Length field in both DTLS layer and UDP layer. The values are always (20,41) respectively when data is "Odd" and (21,42) when data is "Even".

Therefore in this application, given pre-knowledge that server responds with either "Odd" or "Even", the length field in both DTLS layer and UDP layer can directly leak the plaintext.

## 1.2 Leaky Coffee

### 1.2.1 Description

**Leaky Coffee** simulates the scenario that CLIENT initiates a Leaky-Coffee session with a request to SERVER, SERVER replies with a response and CLIENT then reacts according to the response.

#### Syntax

**Definition 1.2.1.** *COFFEE* is a set of strings defined as:
$COFFEE = \{\text{"AMERICANO"}, \text{"CAPPUCCINO"}, \text{"ESPRESSO"}, \text{"MOCHA"}\}$

**Definition 1.2.2.** Let '*' represents SUGAR and '@' represents MILK respectively, we denote $n_*$ and $n_@$ as the number of appearances of '*' and '@' in a string. We also call $n_*$ and $n_@$ the degree of SUGAR and MILK of a string.

**Definition 1.2.3.** We define a set of string *ADDITIVE* as:
$ADDITIVE = \{\{SUGAR, MILK\}^{0-6} | 0 \leq n_* \leq 3, 0 \leq n_@ \leq 3\}$.

In another word, an instance of *ADDITIVE* contains no more than 3 SUGAR and MILK.

#### Leaky-Coffee Session

A Leaky-Coffee session can be described as in Figure 1.2:

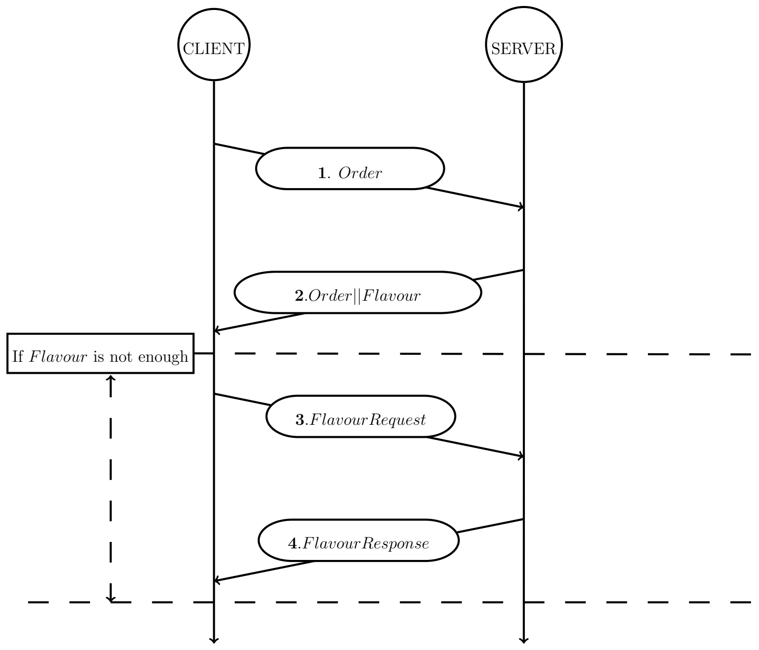


Figure 1.2: Description of a Leaky-Coffee session

**1** As an initiation of a conversation, CLIENT randomly picks a string $Order \in COFFEE$ and sends it to SERVER.

**2** Upon receiving an *Order*, SERVER replies with a string $\{Order||Flavour\}$ where $Flavour \in ADDITIVE$ and $||$ represents concatenation. If $Order$ = "ESPRESSO" then the degrees of both SUGAR and MILK of *Flavour* are set to 0.

**3** CLIENT randomly generates a SUGAR requirement $r_* \in [0,3]$ and a MILK requirement $r_@ \in [0,3]$. Then it scans the reply from **2** and computes its degrees of SUGAR and MILK. If any of the degrees does not met the requirements, i.e. $n_* < r_*$ and/or $n_@ < r_@$, then CLIENT sends a $FlavourRequest = \{"FLAVOUR"||\{SUGAR\}^{\max(r_*-n_*,0)}||MILK^{\max(r_@-n_@,0)}\}$.

**4** If SERVER receives a *FlavourRequest*, it echoes back *FlavourRequest* as its *FlavourResponse*, i.e. $FlvaourResponse = FlavourRequest$.

Note that the *FlavourRequest* and *FlavourResponse* packets are probabilistic in a Leaky-Coffee Session.

**Example 1.2.1.** An example with *FlavourRequest* and *FlavourResponse*(Figure 1.3):

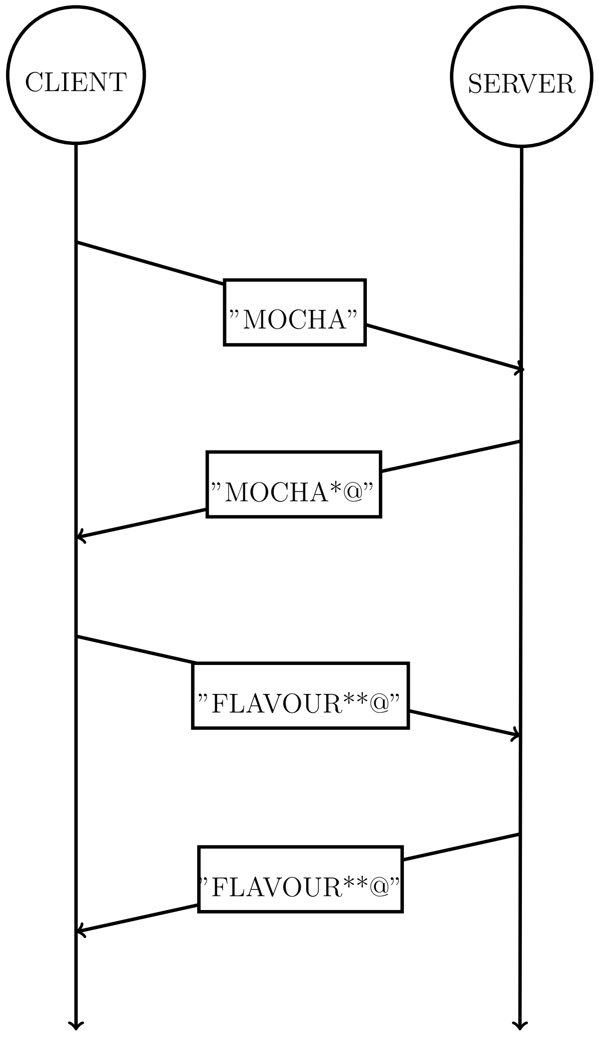


Figure 1.3: Example: A Leaky-Coffee session with *FlavourRequest* and *FlavourResponse*

In this example, CLIENT first sends an *Order* "MOCHA". SERVER then replies with "MOCHA*@" which implies both the SUGAR degree and MILK degree are 1. CLIENT randomly generates a SUGAR requirement 3 and MILK requirement 2 and then sends a *FlavourRequest* to request the shorted SUGAR and MILK. SERVER finally response with the requested

ADDITIVE.

**Example 1.2.2.** Another example without *FlavourRequest* and *FlavourResponse*(Figure 1.4):

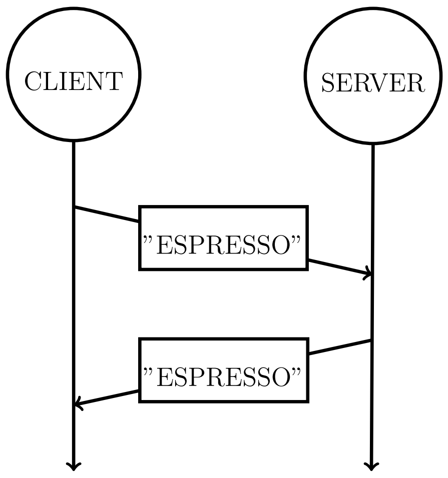


Figure 1.4: Example: A Leaky-Coffee session without *FlavourRequest* and *FlavourResponse*

This example demonstrates a session initiated with "ESPRESSO" where no ADDITIVE will be added in the reply.

### Implementation

SERVER listens to a fixed port (20220) while CLIENT assigns an ephemeral port during each run, i.e. CLIENT's port is selected at the beginning of each run and remains constant during the life time of that instance.

In this experimental implementation, all random values are generated by the Linux kernel random number generator(/dev/urandom); thus assumed to be uniformly distributed.

After each session, CLIENT will be putted into sleep for a random period from 5 to 15 seconds.

We used localloop as our network interface in our experiment; thus packet loss is not considered. DTLS does implement retransmission at some level, but since the sequence number in DTLS header does not change in the retransmitted packet so it is still seemingly possible to reconstruct the equivalent packet stream without any packet loss. Even though the reconstructed stream will preserve all information in each header but the accurate time stamps will be difficult to recover.

## 1.3 Analysis of Leaky Coffee [to be completed]

### 1.3.1 Detect Session

It is obvious that whenever there is a packet transmitted then there

### 1.3.2 Isolate A Session

Given the implementation, we can isolate a session from the packet stream by analysing their time stamp. This is achieved by using a threshold value and then compare it with the interval of two packets. If the interval is greater than the threshold then we can guess these packets belong to different sessions.

```
Algorithm 1: IsSameSession
   Input: threshold $\theta$, time stamps of two continuous packets $t_1, t_2$
   Output: TRUE if the packets are of the same session, otherwise
           FALSE
1  if $\theta > |t_2 - t_1|$ then
2      return TRUE;
3  else
4      return FALSE
5  end
```

In our implementation, a typical guess .By continuously applying Algorithm 1 on the duplex packet stream, we can easily isolate different sessions.

### 1.3.3 Determine Packets in a Session

Once a session has been isolated, it this not difficult to identify the type of each packet in **Leaky Coffee** as there can only be two types of session:

1. Session with 4 packets: This type of session can be identified with 4 packets presented. Further more, those packets can be identified sequentially as: $< Order, Order||Flavour, FlavourRequest, FlavourResponse >$ respectively as well.

2. **Session without *FlavourRequest* and *FlavourResponse*.** 2 packets sessions can be identified as this type of session. Those packets can then be identified as $< Order, Order||Flavour >$ accordingly.

### 1.3.4 Guessing Plaintext by One Packet Length

In this implementation, assume we have the pre-knowledge that each $Order \in COFFEE$ picked by CLIENT has an uniform distribution. Further from, the degree of SUGAR and MILK also have uniform distributions over $0, 1, 2, 3$. Given these distributions, it is some how possible to make guesses of the plaintext in the packets by the length given in DTLS header, or UDP header, without trying to break the encryption primitives.

We denote the value of DTLS Length field as $l_D$ and the actual application data length as $l$. Our experiment shows that:

$$l = l_D - 17 \tag{1.1}$$

under both IPv4 and IPv6.

**Definition 1.3.1.** For a specific packet in a session, let $\mathbb{X}$ be the set of plaintext and $\mathbb{Y}$ be the set of its corresponding content length.

We model the plaintext and their corresponding content length (in bytes) as a channel:

$$W(y|x), x \in \mathbb{X}, y \in \mathbb{Y}.$$

And then the inverse of this channel $W^{-1}(x|y)$ can be viewed as the leakage channel of $\mathbb{Y}$.

The general idea is that with such leakage channel, an adversary can then "decode" the plaintext using this leakage channel.

In this context, $\mathbb{X}$ is the set of packet content and $\mathbb{Y}$ the set of content length $l$.

**Example 1.3.1.** We begin with a simple example: *Order*.

For *Order* packets, we have:

| $W(y\|x)$ | 5 | 8 | 9 | $P$ |
|---|---|---|---|---|
| "AMERICANO" |  |  | 1 | 1/4 |
| "CAPPUCINO" |  |  | 1 | 1/4 |
| "MOCHA" | 1 |  |  | 1/4 |
| "ESPRESSO" |  | 1 |  | 1/4 |

Table 1.1: Content-Length Channel and the probabilities of *Order*

In this implementation, CLIENT randomly picks *Order* from *COFFEE*; therefore the probability for every value is 1/4. Since neither DLTS nor the application induces any randomness to the content length therefore it will always be a deterministic value.

Given $W$ and the probability of *Order*, it can then compute the joint distribution of $(Order, l)$ by:

$$(\widehat{WP})(x,y) = P(x)W(y|x) \tag{1.2}$$

| $\widehat{WP}$ | P |
|---|---|
| ("AMERICANO",9) | 1/4 |
| ("CAPPUCINO",9) | 1/4 |
| ("MOCHA",5) | 1/4 |
| ("ESPRESSO",8) | 1/4 |

Table 1.2: Joint distribution of $(Order, l)$

Then follows the marginal distribution of content length:

$$P(Y = y) = \sum_{x \in \mathbb{X}} \widehat{WP}(x,y) \tag{1.3}$$

| $y$ | P |
|---|---|
| 5 | 1/4 |
| 8 | 1/4 |
| 9 | 1/2 |

Table 1.3: Marginal distribution of $l$

Finally we can construct the leakage channel using Bayes' theorem:

$$P(x|y) = \frac{P(x)P(y|x)}{P(y)} \tag{1.4}$$

| $W^{-1}(x\|y)$ | "AMERICANO" | "CAPPUCINO" | "ESPRESSO" | "MOCHA" |
|---|---|---|---|---|
| 5 | | | | 1 |
| 8 | | | 1 | |
| 9 | 1/2 | 1/2 | | |

Table 1.4: Leakage channel of Length - *Order*

The same strategy can also be applied on the second packet: $Order||Flavour$.

**Example 1.3.2.** The first step is to compute the Content-Length channel. Analysis on $Order||Flavour$ packet is more complicated as it has a larger entropy.

We omit the sequence of SUGAR and MILK to simplify the problem. We also simplify our notation by denoting $D_1$ as the degree of SUGAR and $D_2$ the degree of MILK. Then any *Flavour* can be represented as $(D_1, D_2)$, e.g. $(2, 1)$ represents any *Flavour* that has a degree of SUGAR 2 and degree of MILK 1.

The same strategy can be applied directly on this example as well. However, the space of this channel is much more complicated in this case which are 4

It is sometimes possible to simplify the problem by breaking the Plaintext-Length channel into several sub-channels, namely *Order* channel $W_0(y \in l|x \in COFFEE)$, SUGAR channel $W_1(y \in l|x \in D_1)$ and MILK channel $W_2(y \in l|x \in D_2)$ in this application. These sub-channels requires less computation and we will show how to reconstruct the Plaintext-Length channel using these sub-channels later in this section.

Obviously that $W_0$ is identical to Table 1.1 as the *Order* part in $Order||Flavour$ is simply an echo of the first *Order* packet.

$W_2$ and $W_3$ are actually identical:

| $W_1(x\|y)$ | 0 | 1 | 2 | 3 | P |
|---|---|---|---|---|---|
| 0 | 1 | | | | 1/4 |
| 1 | | 1 | | | 1/4 |
| 2 | | | 1 | | 1/4 |
| 3 | | | | 1 | 1/4 |

| $W_2(x\|y)$ | 0 | 1 | 2 | 3 | P |
|---|---|---|---|---|---|
| 0 | 1 | | | | 1/4 |
| 1 | | 1 | | | 1/4 |
| 2 | | | 1 | | 1/4 |
| 3 | | | | 1 | 1/4 |

Table 1.5: Channels of SUGAR-Length and MILK-Length

Then we merge $W_1$ and $W_2$ to construct the *Flavour* - Length channel $W_1 \otimes W_2((y_1, y_2)|(x_1, x_2))$ where $(y_1, y_2) \in l \otimes l, (x_1, x_2) \in D_1 \otimes D_2$:

| $W_1 \otimes W_2((y_1, y_2)\|(x_1, x_2))$ | (0,0) | (0,1) | ... | $(y_1, y_2)$ | ... | (3,2) | (3,3) | P |
|---|---|---|---|---|---|---|---|---|
| (0,0) | 1 | | | | | | | 1/16 |
| (0,1) | | 1 | | | | | | 1/16 |
| ... | | | | | | | | |
| $(x_1, x_2)$ | | | | $P((y_1, y_2)\|(x_1, x_2))$ | | | | $P(x_1, x_2)$ |
| ... | | | | | | | | |
| (3,2) | | | | | | 1 | | 1/16 |
| (3,3) | | | | | | | 1 | 1/16 |

Table 1.6: *Flavour*-Length channel

The right end column of probability is simply the joint probability of both inputs of $W_1$ and $W_2$.

In this application, degree of SUGAR and degree of MILK are independent variables. This implies their joint probability is simply the product of their marginal probabilities:

$$P(x_1, x_2) = P(x_1)P(x_2) \tag{1.5}$$

Notice that since the output of such channel are actually the length of its input; therefore for a given input, its output is deterministic, i.e.

$$P((y_1, y_2)|(x_1, x_2)) = \begin{cases} 1 & \text{if } y_1 = |x_1| \text{ and } y_2 = |x_2| \\ 0 & \text{otherwise} \end{cases} \tag{1.6}$$

The merged channel $W_1 \otimes W_2$ results in a table with size of $(|X_1||X_2|)$ rows and $(|Y_1||Y_2|)$ columns. This implies that the merge operation of two channels will potentially has an exponential time and space complexity. However, this can be improved by compressing the channel.

The first thing is that the merged output are actually lengths of both inputs; hence $(y_1, y_2)$ can be replaced by their sum: $y = y_1 + y_2$. Therefore Table 1.6 can be compressed by combing columns with a same length, i.e. we can merge columns into one if $(y_1 + y_2) = (y'_1 + y'_2)$. The combination is simply the vector sum of two columns

So we can reconstruct $W_1 \otimes W_2$ as:

| $(W_1 \otimes W_2)'(y = y_1 + y_2 \mid (x_1, x_2))$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | P |
|---|---|---|---|---|---|---|---|---|
| (0,0) | 1 | | | | | | | 1/16 |
| (0,1) | | 1 | | | | | | 1/16 |
| ... | | | | | | | | |
| (1,0) | | 1 | | | | | | |
| ... | | | | | | | | |
| (3,2) | | | | | | 1 | | 1/16 |
| (3,3) | | | | | | | 1 | 1/16 |

Table 1.7: Compressed *Flavour*-Length channel

In Table 1.7, we can see that different inputs can map to the same length, e.g. $(0, 1)$ and $(1, 0)$ all results to $l = 1$.

Practically, we can further compress this channel with the cost of resolution of input. Generally, there are some facts that worth notice:

- As in (1.6), length is deterministic given a content. Therefore it is a reasonable choice to merge contents which will result into same length.
- The intuition of combing two rows with the same length can be interpreted as follow: for two rows with the same output $W(y|x = x_1)$ and $W(y|x = x_2)$, the merged row represents $W(y|x = x_1 \text{ or } x = x_2)$.
- For such a channel, each input are exclusive events; thus the probability of the input of merged rows is simply the sum of the probability of each row: $P(x_{merged}) = P(x_1) + P(x_2)$

So if we compress Table 1.7 by the same $(y_1+y_2)$ which is indeed $|Flavour|$, we will have a further compressed $W_1 \otimes W_2$:

| $(W_1 \otimes W_2)''(y = y_1 + y_2 \vert x = x_1 + x_2)$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | P |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | | 1/16 |
| 1 | | 1 | | | | | | 1/8 |
| 2 | | | 1 | | | | | 3/16 |
| 3 | | | | 1 | | | | 1/4 |
| 4 | | | | | 1 | | | 3/16 |
| 5 | | | | | | 1 | | 1/8 |
| 6 | | | | | | | 1 | 1/16 |

Table 1.8: Further Compressed (with less resolution) *Flavour*-Length channel

The actual degree of SUGAR and MILK are lost in Table 1.8 during this compression, but it also reduced the number of rows from 16[1] to 7.

By applying the same strategy again to merge the *Order* channel $W_0$ with $(W_1 \otimes W_2)$, we will have the *OrderFlavour*-Length channel $W = W_0 \otimes W_1 \otimes W_2$. Then finally as described in Example 1.3.1, we can construct the leakage channel $W^{-1}(x|y)$ (see Chapter A) for the *OrderFlavour* packets .

To generalise, given the distribution of the plaintext, the leakage channel is constructed as following:

**1** If the space of plaintext is large, break the plaintext-length channel into several sub-channels.

**2** Compute the sub-channels and compress them. Resolution may lost during the compression.

**3** Merge the sub-channels to construct the plaintext-length channel.

**4** Use Bayes' Theorem to invert plaintext-length channel.

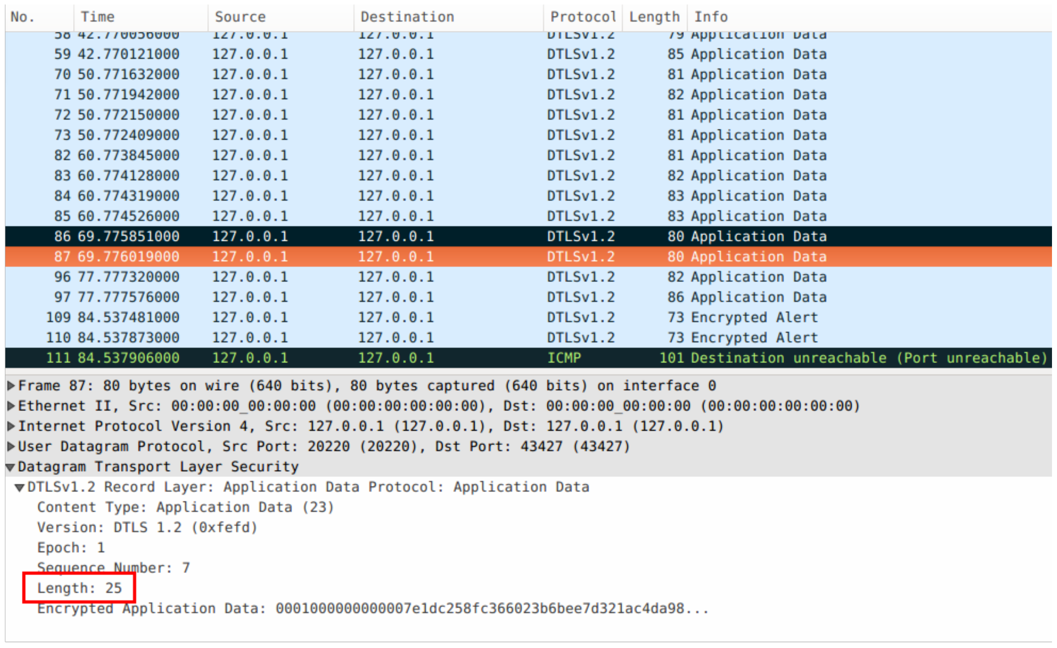
Another aspect to view such leakage channel is to analyse its capacity, i.e. the maximum mutual information of content and length, described in [1].

**[Experiment results...(from Baikal?)]**

[1] $(W_1 \otimes W_2)'$ has $4 \times 4 = 16$ rows.

### 1.3.5 Guessing Plaintext Using Joint Packet Length

In Section 1.3.4 we described a method of packet analysis against a single packet in a session. It is possible to improve the analysis by looking at the packets jointly. As presented in [1], the sequence of packets lengths can be viewed as a vector.



```
No.   Time             Source        Destination        Protocol Length Info
   58 42.770056000     127.0.0.1     127.0.0.1          DTLSv1.2    79 Application Data
   59 42.770121000     127.0.0.1     127.0.0.1          DTLSv1.2    85 Application Data
   70 50.771632000     127.0.0.1     127.0.0.1          DTLSv1.2    81 Application Data
   71 50.771942000     127.0.0.1     127.0.0.1          DTLSv1.2    82 Application Data
   72 50.772150000     127.0.0.1     127.0.0.1          DTLSv1.2    81 Application Data
   73 50.772409000     127.0.0.1     127.0.0.1          DTLSv1.2    81 Application Data
   82 60.773845000     127.0.0.1     127.0.0.1          DTLSv1.2    81 Application Data
   83 60.774128000     127.0.0.1     127.0.0.1          DTLSv1.2    82 Application Data
   84 60.774319000     127.0.0.1     127.0.0.1          DTLSv1.2    83 Application Data
   85 60.774526000     127.0.0.1     127.0.0.1          DTLSv1.2    83 Application Data
   86 69.775851000     127.0.0.1     127.0.0.1          DTLSv1.2    80 Application Data
   87 69.776019000     127.0.0.1     127.0.0.1          DTLSv1.2    80 Application Data
   96 77.777320000     127.0.0.1     127.0.0.1          DTLSv1.2    82 Application Data
   97 77.777576000     127.0.0.1     127.0.0.1          DTLSv1.2    86 Application Data
  109 84.537481000     127.0.0.1     127.0.0.1          DTLSv1.2    73 Encrypted Alert
  110 84.537873000     127.0.0.1     127.0.0.1          DTLSv1.2    73 Encrypted Alert
  111 84.537906000     127.0.0.1     127.0.0.1          ICMP       101 Destination unreachable (Port unreachable)

▶Frame 87: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface 0
▶Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▶User Datagram Protocol, Src Port: 20220 (20220), Dst Port: 43427 (43427)
▼Datagram Transport Layer Security
 ▼DTLSv1.2 Record Layer: Application Data Protocol: Application Data
    Content Type: Application Data (23)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 1
    Sequence Number: 7
    Length: 25
    Encrypted Application Data: 0001000000000007e1dc258fc366023b6bee7d321ac4da98...
```

Figure 1.5: Captured Leaky Coffee packets

**Example 1.3.3.** For example, a 2-packets session (packet No 86 and 87) has been marked out in Figure 1.5. The values of DTLS Length are both 25 as marked in red rectangle. Their actual plaintext length can then be computed as 8 and 8 bytes respectively by Equation (1.1).

As described in [1],

It is possible to do the single packet analysis described in Section 1.3.4 on each of these packets. The result of the first packet tells us that its plaintext is "ESPRESSO"; whilst the second one could be either "ESPRESSO" or "MOCHA" with a *Flavour* of length 3. For instance, The application specifies that the first part of the $Order||Flavour$ is simply the an echo of the first packet; therefore in fact we can immediately tell that the second packet is actually “ESPRESSO”.

There are several ways to utilise such knowledge, such as using some machine learning techniques. However, in this “intentionally crafted” Leaky Coffee application, the first packet seems always enough to reveal (roughly) the rest of plaintext in a session.

### 1.3.6 Other Attacks (Future work???)

Is it feasible to estimate the distribution of plaintext given packet length alongside with other assumptions?

# Appendix A

# $OrderFlavour$-Length leakage channel

In this application, the joint probability of *Order* and *Flavour* are simply the product of their marginal probability. However, since “ESPRESSO” will always followed by *Flavour* of of both degree of SUGAR and MILK being 0(see Section 1.2.1); hence

$$P(x_1, x_2 | \text{“ESPRESSO”}) = \begin{cases} 1 & \text{if } x_1 = x_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore

$$P(\text{“ESPRESSO”}, x_1, x_2) = \begin{cases} 1/4 & \text{if } x_1 = x_2 = 0 \\ 0 & \text{otherwise} \end{cases}$$

# Bibliography

[1] Mather, L., and Oswald, E. Pinpointing side-channel information leaks in web applications. *J. Cryptographic Engineering 2*, 3 (2012), 161–177.