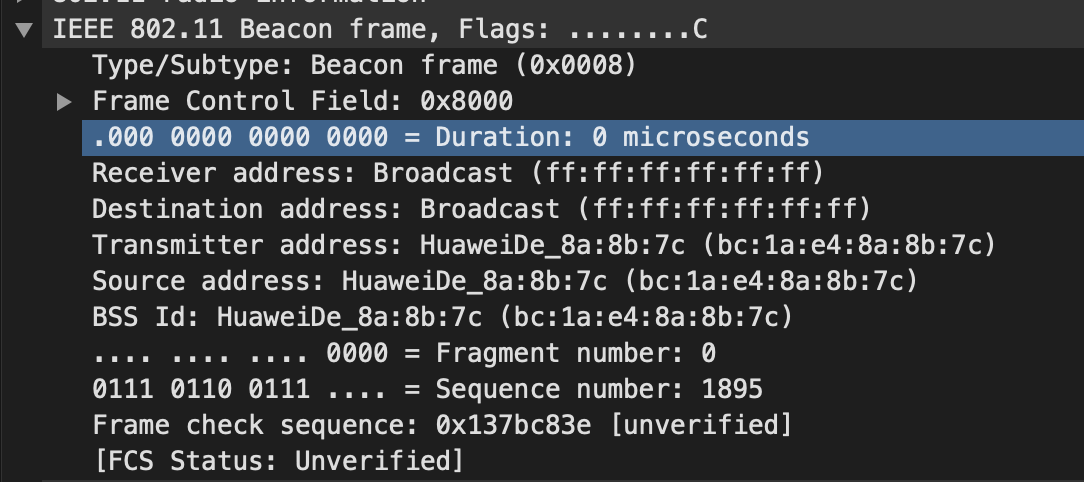
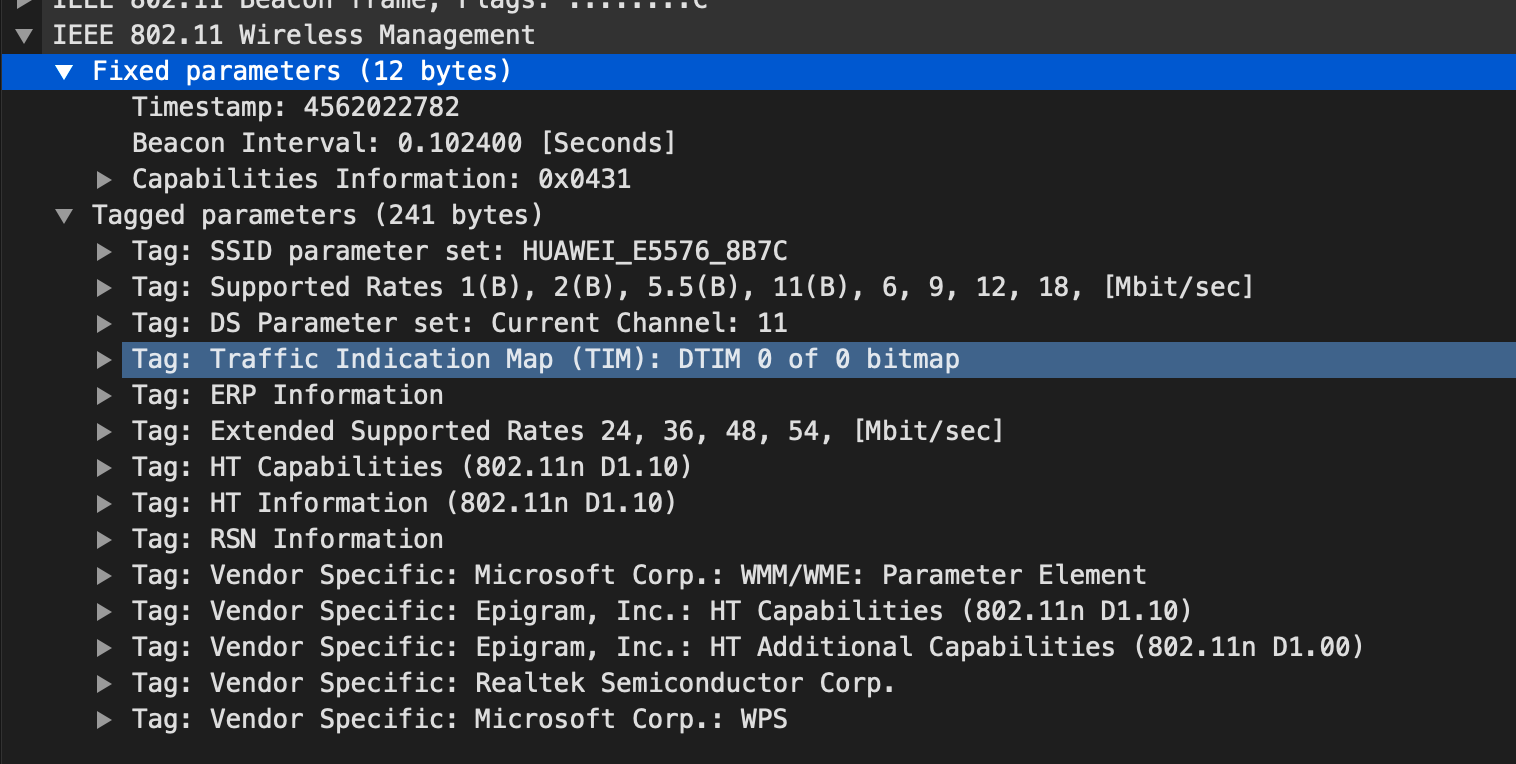
Beacon帧（From AP To STA）（No ACK帧）

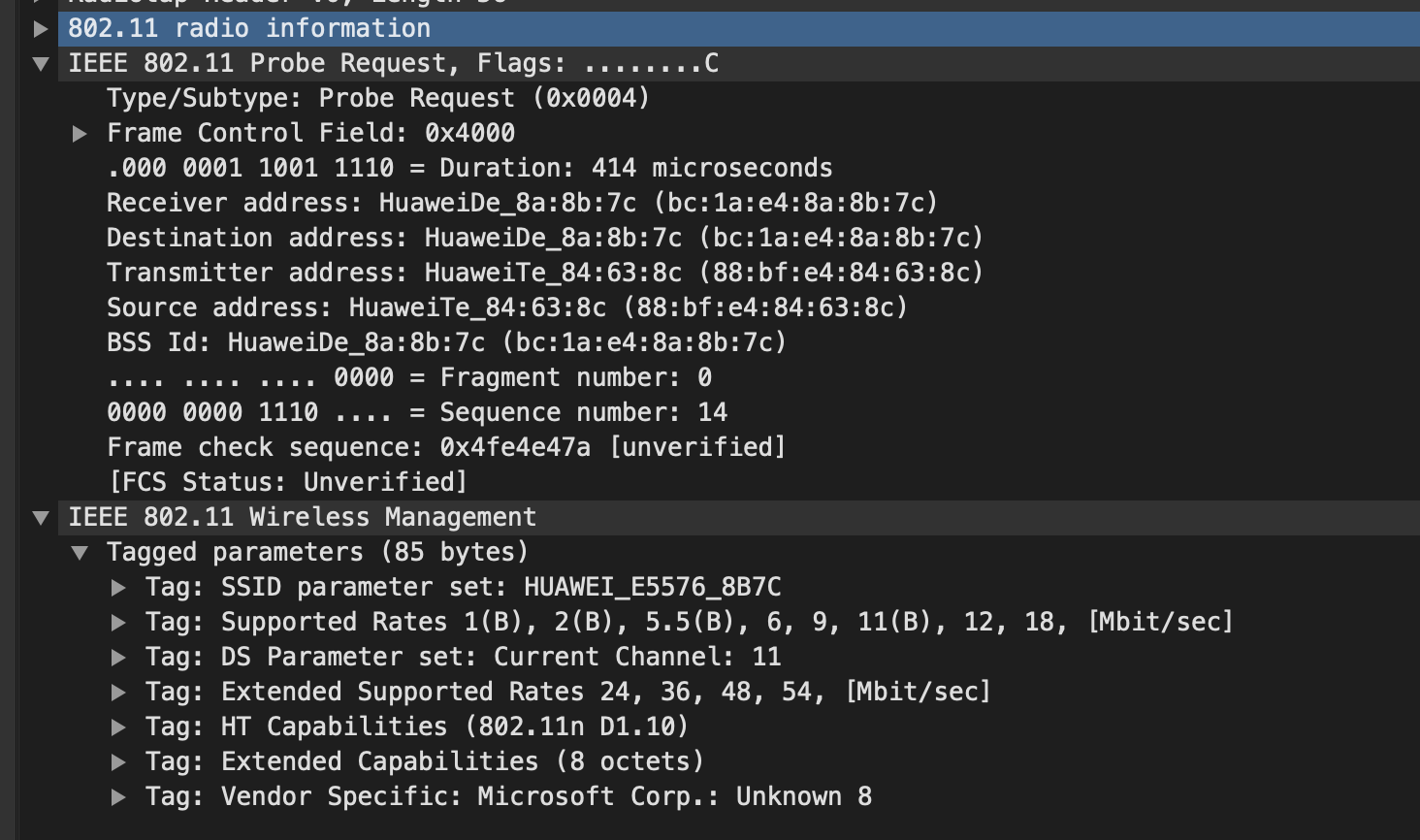
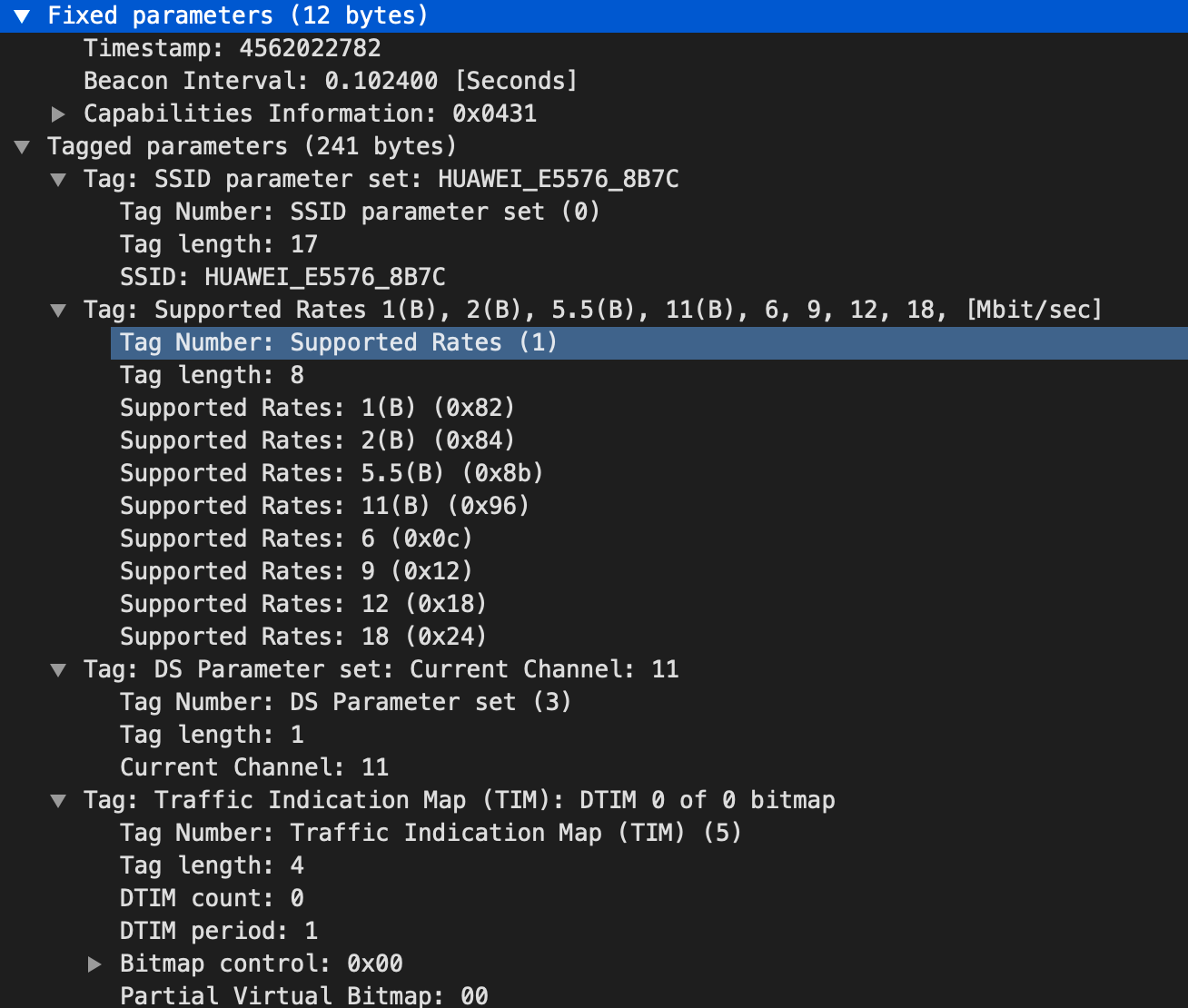
帧头部：

Frame Body

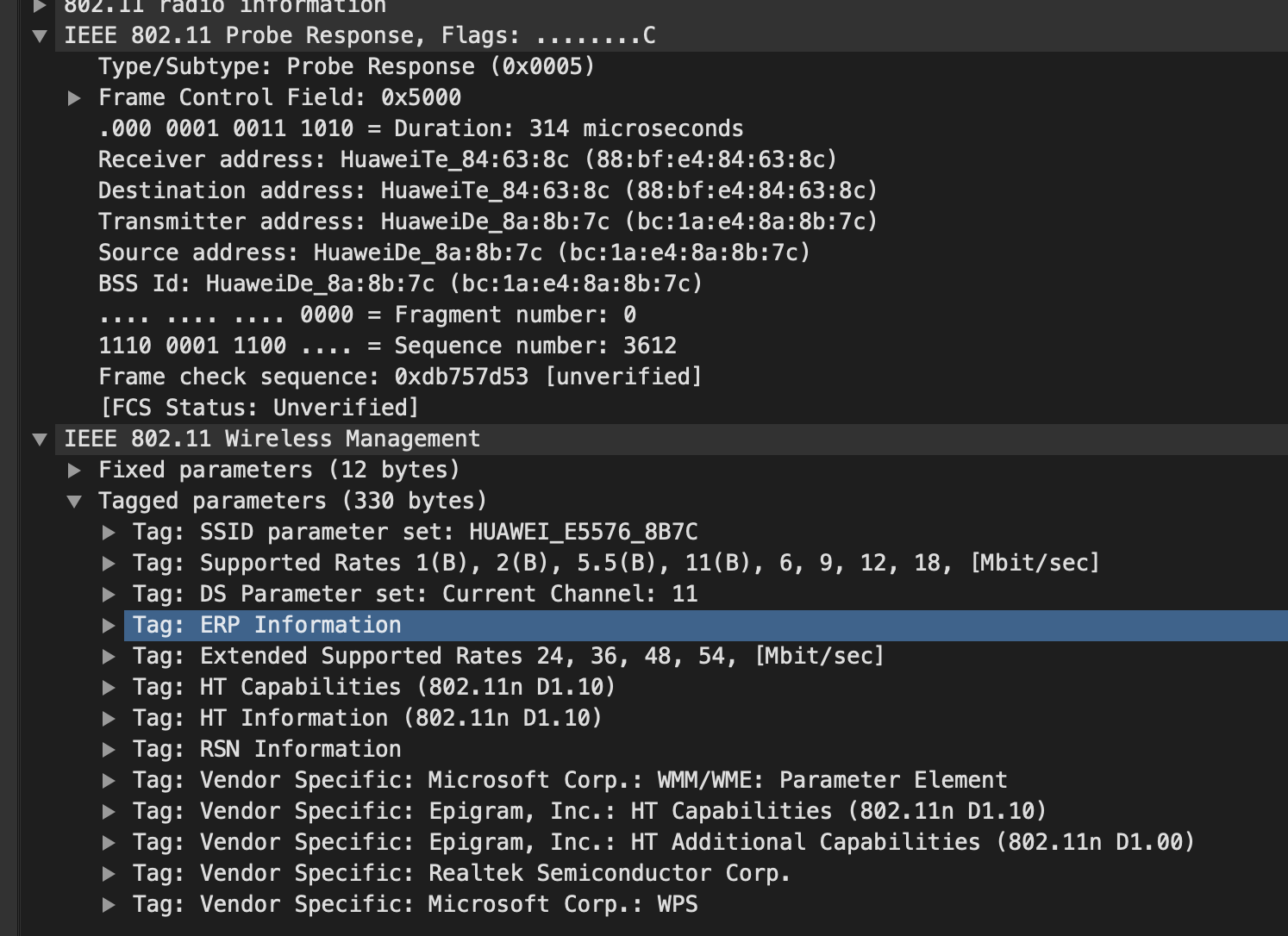
Fuzz 对象

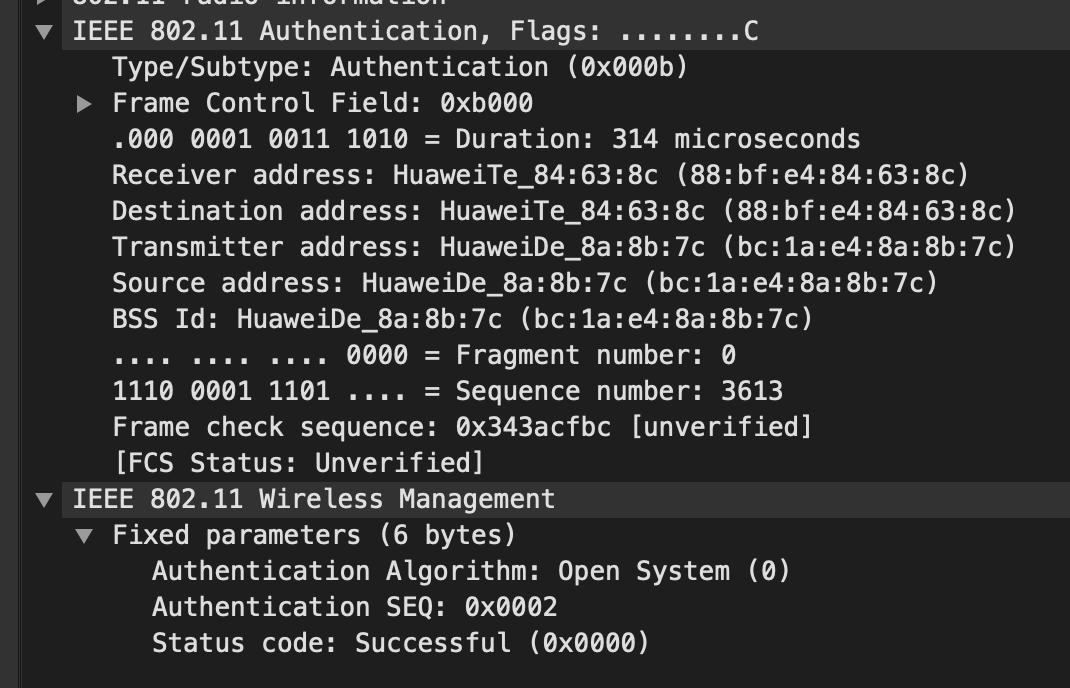
Tagged parameters里面的各个子Tag（早期的帧只有8个information elements，2012版本已经有上百个），结构如下：

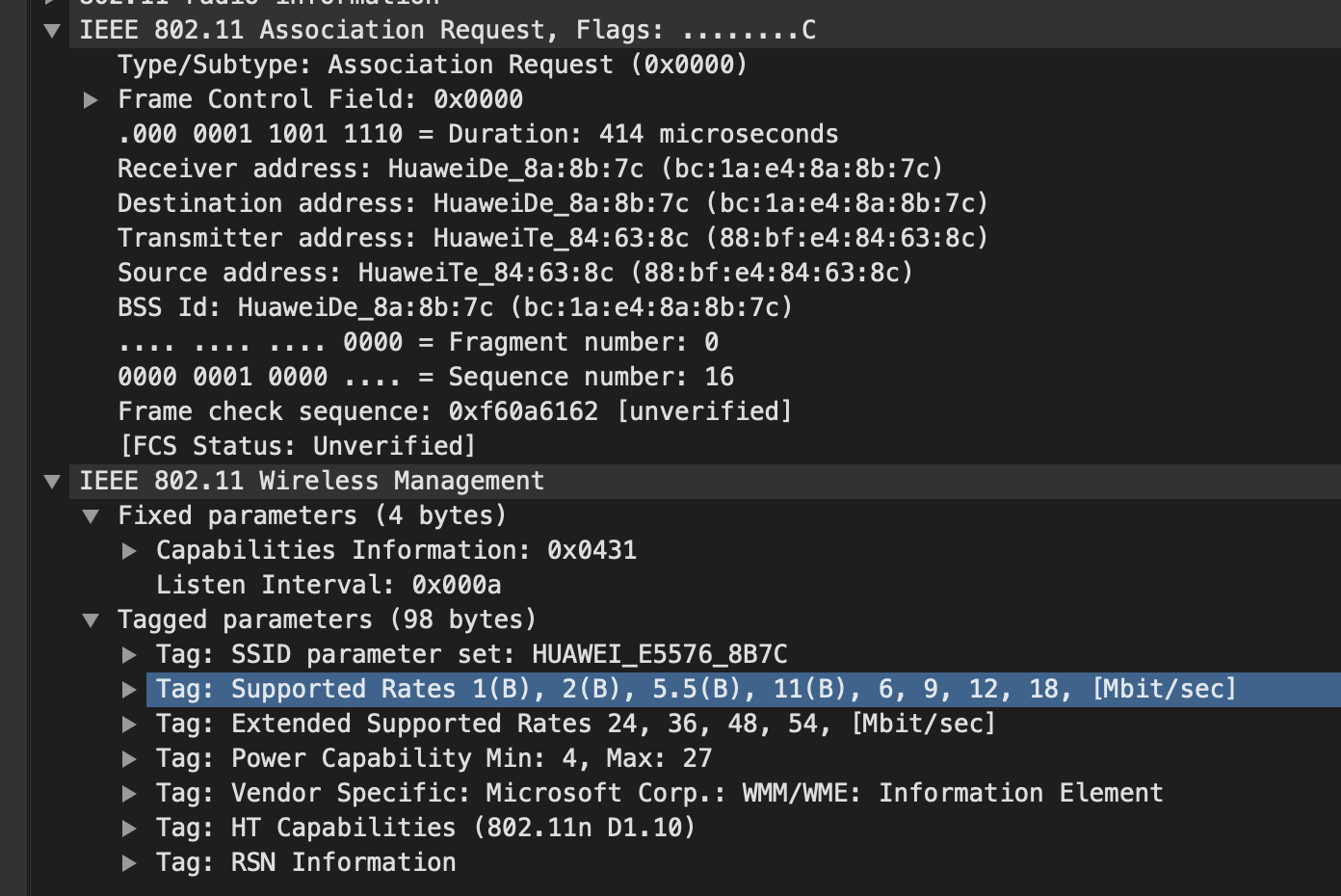
每个Tag类似于一个TLV结构：



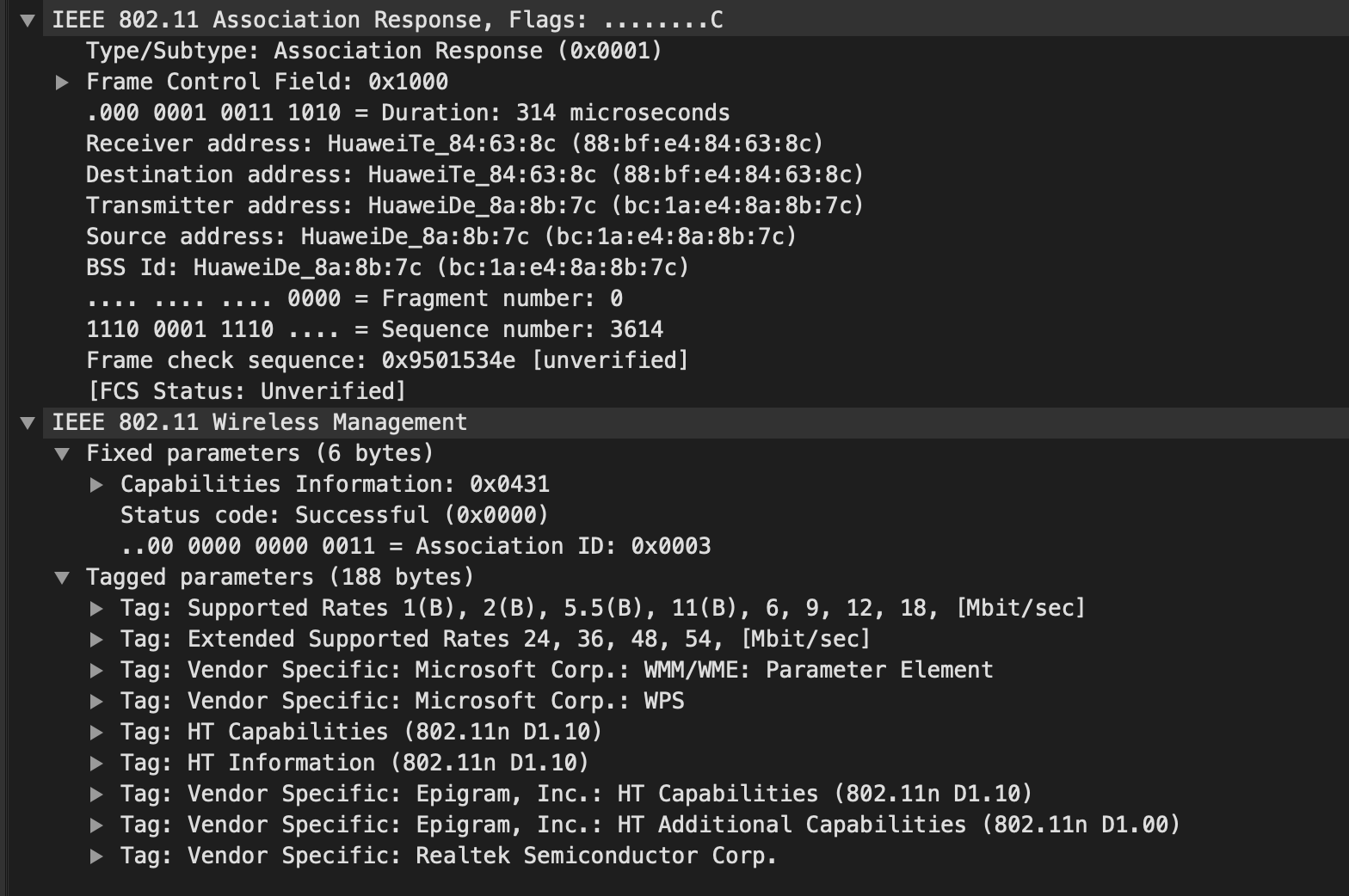
Probe Request（From STA To AP）

Probe Response（AP Response To STA）

Authentication

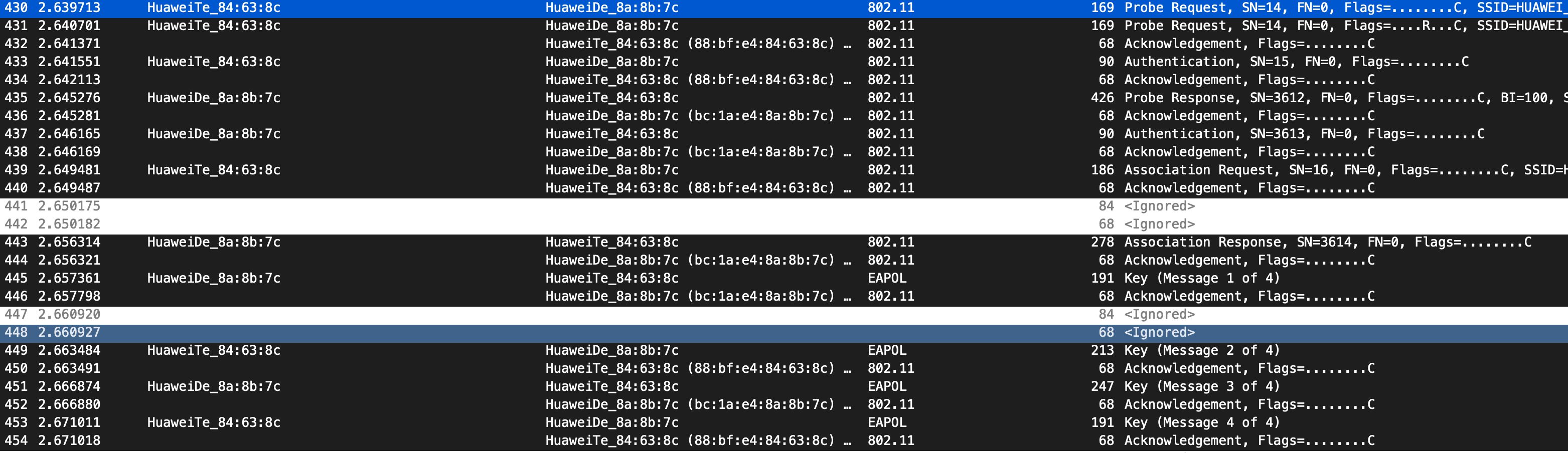
Association Request（From STA To AP）

Association Response（AP Response To STA）



Deauthentication

Disassociation

手机Wi-Fi上线流程

待补充一个交互流程图

Fuzzing Wi-Fi in IoT devices （论文）

五个问题：

1、当我们发送一个数据帧到SUT时，我们怎么知道其是否被正确接收？

2、当我们发送一个数据帧到SUT，该帧被正确接收，我们怎么知道其是否被解析？

3、为了使数据帧更有可能被SUT解析，哪一个时刻发送数据帧最好？

4、针对SUT发出的Probe Request帧，我们回复的Probe Response必须在毫秒级别，以避免SUT转移到另一个信道，那样的话Request帧将失效。

5、当我们收到一个来自于SUT的帧时，我们必须在非常短的时间内回复一个ACK帧，以通知我们正确收到帧。

802.11不同版本使用不同的带宽，主要分为5GHz和2.4Ghz两个，本文主要关注2.4GHz带宽。2。4Ghz分为14个信道，一个设备同一时刻只能监听或发送数据在单一信道上。

普遍意义上wifi存在两种网络架构：infrastructure networks 和ad hoc networks. ad hoc networks只有client设备，相互之间采用直连方式，意味着不能上网，本文并不关注。

infrastructure networks下Wi-Fi芯片四种模式：

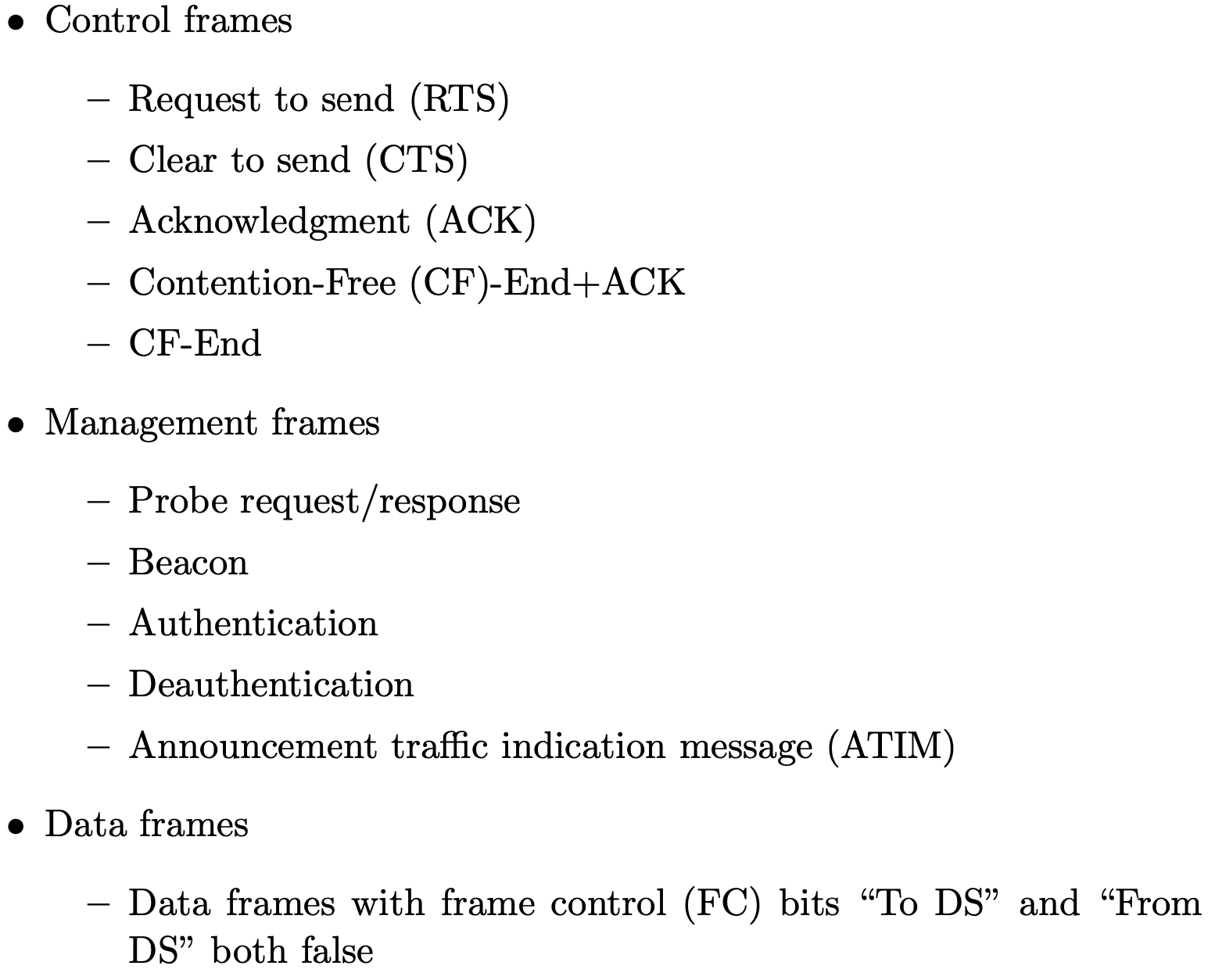
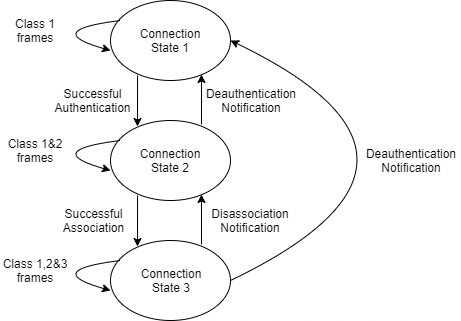
1、client mode

2、AP mode

3、Promiscuous mode ：嗅探网络包（前提是已经接入点网络），链路层

4、Monitor mode ：嗅探网络包（不需要接入网络），物理层和链路层

本文研究对象是处于client mode的设备，也就是fuzzer作为AP mode，至于AP mode将会作为后续的研究。

Wifi接入认证包含了三个状态每个状态下只有特定的帧被允许，因此对所有的wifi帧进行了分类，class1/2/3，每个状态下允许的帧类型如下图：

比如Sate1包含的所有帧类型有这些：

Fuzzing in which state？

以往的研究基本是在state3下进行的，原因是state3会允许所有的帧存在，因此可能发送与接收所有帧类型。但是，这就存在的问题是降低了帧被设备处理的概率，比如只有在设备发送了probe request帧时才会接受处理probe response帧，因此，在其他时刻发送probe response是没有任何作用的（这其实涉及到一个状态的问题）。

Interesting frame to fuzz?

Beacon/Probe帧：

优点：最小化用户交互，不需要前提条件

缺点：攻击窗口小（当然不排除一些设备在state3依然会扫描state1的帧）

(De)Authenticate/(Dis)Association

优点：攻击窗口相对较大

缺点：需要一些前提条件

控制帧：

优点：比较罕见，可能在测试中被忽略

缺点：没有frame body，也没有可变长度

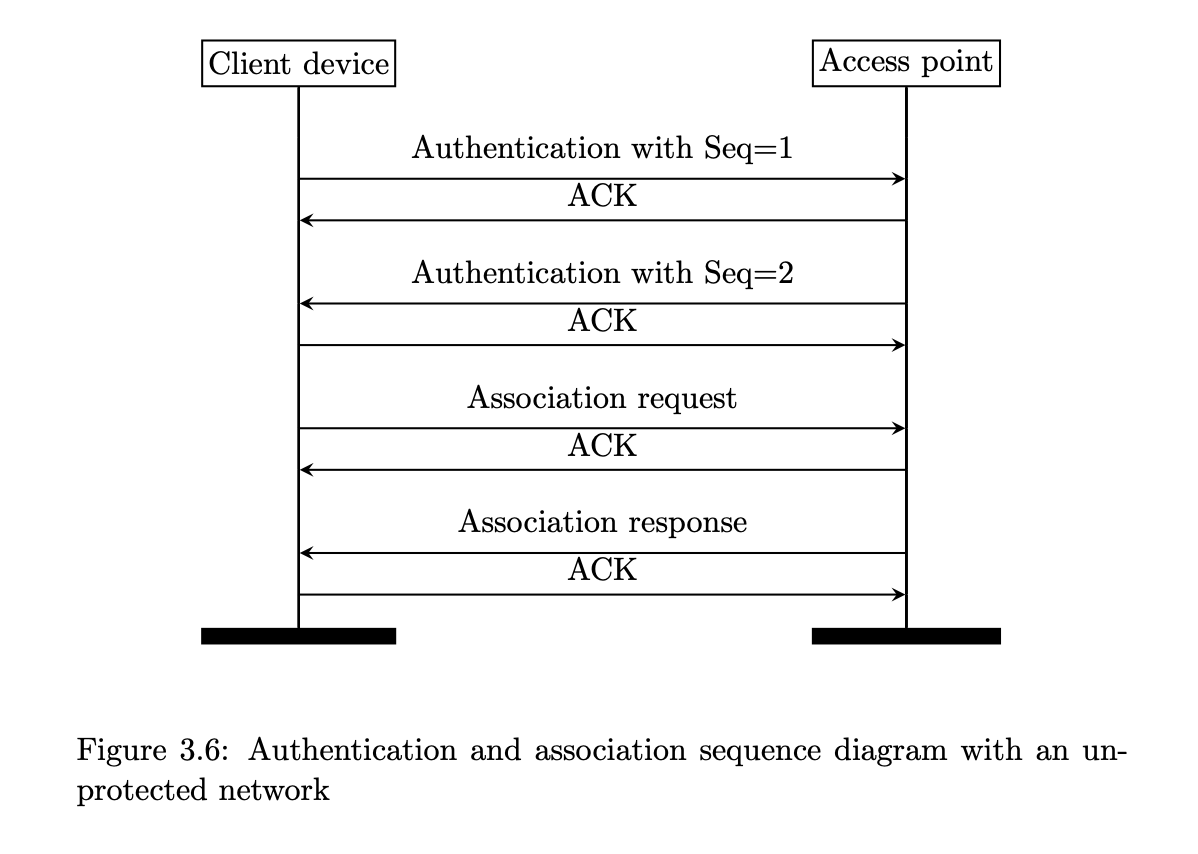
ATIM帧：

仅存在ad-hoc mode，fuzz必要性不大

数据帧：

并不被被驱动或固件解析，没有fuzz的必要

主动扫描模式和被动扫描模式：

注意：在主动扫描模式下，probe帧发送之后只会等待probe response一段时间，然后就跳转到下一个信道，这里的“一段时间”并不少规定死的，不同设备有不同的区别。

Fuzz STA的设备要求：

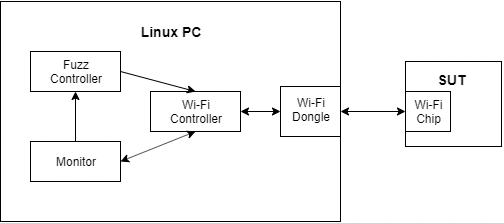
1、iot设备需要支持client mode

2、iot设备要支持扫描网络（不停的扫描），以便与我们的fuzzer进行通信

3、为了对Authentication和Association帧进行fuzz，对象设备必须不停的尝试与我们的fuzzer程序进行认证和关联动作

CFuzz实现

结构示意图：



Wi-Fi Controller：使用Wifi-Dongle来接受和发送数据帧；

Fuzz Controller：用于生成fuzzed frame同时决定fuzz帧的哪一个部分；

Minotor：用于监控SUT来验证数据帧是否被接收，同时验证SUT是否崩溃；

**Wi-Fi Controller**

在Wi-Fi Controller的实现过程中，由于需要与Wifi-Dongle通信，作者对比分析了四种方式：Scapy、Libpcap、RAW socket programming和Lorcon，有效方式位前三种，最终选择Libpcap。

Scapy：python库，使用方便，但是由于是python实现所以效率不够。在测试的设备中并不是所有的设备都能在timeout之前收到回应报文（probe response），也就无法确定数据帧是否被解析，与问题2对应。实验结果显示基于scapy的respond时间（收包到再响应时间）是0.03秒，这个时间在很多设备上是 难以接受的。原因其实很简单，基于python的实现在运行过程中需要使用解释器将python代码转换为机器码，而C是直接被编译成机器码。

Libpcap：C语言的库，使用相对复杂，但是效率高。测试结果显示respond平均时间0.003秒，解决问题4。同时通过监听ACK帧，可以解决问题1。在测试认证和关联阶段时，作者发现ACK帧其实是由Wifi-Dongle的固件模块负责发送的，因此解决了问题5。

RAW socket programming：直接访问网络接口，实现过于复杂，不考虑。

**Fuzz Controller**

当前已有fuzzer框架（基于可定制化以及性能的考虑并没有使用这些框架）：

Sulley/Boofuzz：python库，集成802.11 fuzzer，由于是python实现的所以难以适配。

BeSTORM：商业软件，不开源。

Peach Fuzzer：集成了wifi模块，但是声明表示只适用于确定的Ralink芯片组，似乎可定制性不高。