Bachelor thesis
Computing Science

Radboud University

Inferring state machines of Tunneled Direct-Link Setup

Author:
Sébastiaan Versteeg
s4459636
`hello@sebastiaan.app`

First supervisor/assessor:
dr. ir. Joeri de Ruiter
`joeri@cs.ru.nl`

Second assessor:
dr. ir. Erik Poll
`e.poll@cs.ru.nl`

January 23, 2019

Abstract

The usage increase of the internet and in specific wireless networks has introduced new difficulties in keeping reliable connections. TDLS creates a direct link between two communicating clients improving reliability for all other clients. This protocol is a part of the IEEE 802.11 specification [1] and includes measures to secure the direct link. The specification is over 3000 pages long. This may introduce problems for software engineers creating a new implementation of TDLS. Writing such an implementation is hard and bugs can be introduced in the process, weakening the security of a direct link. We will introduce a method to automatically infer state machines of TDLS implementations. This allows us to get insight in the conformity of such implementations to the specification. We will utilise the L* and the randomwords algorithms to execute the inferring process on the wpa_supplicant implementation of TDLS by utilising our own mapper. By analysing the results of this process we will show that TDLS state machines can be inferred successfully and that inferring such state machines can improve our knowledge about the inner workings of protocols.

Contents

Chapter 1

Introduction

A lot has changed since the introduction of personal computers. Networks have been set-up everywhere and smartphones have been invented. The past decades most internet networks have become wireless. The most used wireless internet protocols are defined in the IEEE 802.11 specification [1] and are more commonly known under the brand name 'Wi-Fi'. Part of this specification is a way to let clients (stations, STAs) communicate directly without severing the connection to the shared access point (AP), this is called a Tunneled Direct Link Set-up (TDLS). A TDLS connection reduces the load on the access point improving the reliability of the connections made by individual clients. This specific part of the specification introduces a handshake protocol which is used to secure the link. We want to look into this part of the specification, analyse the implementation to find out if this handshake is properly executed. Since the protocol is intended for public use, people should be able to rely on the handshake to be completely secure. Analysing the implementation will make sure the public can use the TDLS functionality without worrying about whether their connection is secure. For the analysis we automatically infer the state machine of the implementation of the TDLS handshake comparing the output to the official specification.

This thesis is structured as follows: firstly we will discuss the 802.11 specification (2.1) and the TDLS protocol (2.2). Next we will take a look at Mealy machines and how to infer that kind of state machines (2.3, 2.4). Consequently we will explain how we implemented our mapper (3.1) and how it works in our setup (3.2). Lastly we will compare our expectations (4.1) based on the specification to the inferred state machine (4.2) and draw our conclusions (6).

Chapter 2

Preliminaries

In this chapter we give an introduction to the protocol and method we are using for our research.

2.1 IEEE 802.11 / Wi-Fi

Before we introduce Tunneled Direct-Link Setup we introduce the IEEE 802.11 specification [1] that it is a part of. The standard defined by the 802.11 specification is more commonly known under the Wi-Fi brand overseen by the Wi-Fi Alliance. It consists of multiple media access control and physical layer protocols for the implementation of wireless local area networks (WLAN) including information about radio frequencies, modulation techniques and packet formatting.

The original specification was published in 1997 and it is continuously updated with amendments that add and modify the protocols to accommodate new developments. The latest publication of the standard with the inclusion of all amendments was in 2016.

2.1.1 Wireless local area networks

The 802.11 specification is mostly used to setup wireless local area networks. A setup of a WLAN consists of one or more stations (STAs). These stations have the appropriate hardware to wirelessly send and receive data.

To provide a WLAN one of the stations will act as access point (AP) to manage all the traffic over the network. Every client station will individually connect to the access point which will provide it with access to all the other clients on the network. An access point is usually connected to a wired network that has access to the internet.

The combination of the AP and the connected STAs is called a basic service set (BSS). The BSS can be identified by the unique basic service set identifier (BSSID). This identifier is usually preset on the access point and unchangeable. The identifier used to create unique wireless networks is the service set identifier (SSID) that is broadcast to announce the presence of a network.

2.1.2 Security

The original specification provided a security solution to wireless networks named Wired Equivalent Privacy (WEP) which was proved insufficiently secure in 2001 due to the use of the RC4 stream cipher [8]. To solve this problem the Wi-Fi Alliance introduced Wi-Fi Protected Access (WPA). This solution was based on a draft of the 802.11i amendment and used the Temporal Key Integrity Protocol (TKIP). This was an intermediate solution that could be implemented on older hardware that supported WEP. The eventual 802.11i amendment was used to create WPA2. Replacing WPA this new security mode uses the Counter Mode CBC-MAC Protocol (CCMP), which is based on AES.

Both WPA and WPA2 use the 4-Way Handshake for the authentication to an access point. Implementations of this handshake were shown to be vulnerable to key re-installations in 2017 [19]. The Wi-Fi Alliance announced WPA3 in January 2018. This new solution uses the Simultaneous Authentication of Equals (SAE) handshake in combination with the 4-Way Handshake for personal networks.

The 4-Way Handshake

802.11 specifies the 4-Way handshake as tool to authenticate stations and provide session keys. Before the handshake takes place the stations will go through two stages first: discovery and 802.11 authentication and association. The discovery stage allows stations to find access points and determine the supported cipher suites (TKIP/CCMP). The 802.11 authentication and association is an exchange that the station can use to make itself known to the AP and choose a cipher suite. Be aware that this phase does not do any actual authentication yet. If the AP accepts the connection the real authentication using the 4-Way Handshake will start. This handshake will result in a Robust Secure Network Association (RSNA) of the STA with the AP.

The 4-Way Handshake provides mutual authentication between the STA and AP by using a Pre-Shared Key (PSK). This PSK is used in combination with two nonces (ANonce and SNonce) and the MAC addresses of the stations to calculate the Pairwise Transient Key (PTK) as session key.

The handshake is started by the AP using the EAPOL-Key1 message which communicates it's nonce (ANonce) to the connecting station. This station can respond with the EAPOL-Key2 message. This message consists of the SNonce with a Message Integrity Code (MIC) and the chosen cipher suite (RSNE). The MIC is calculated using the PTK over the complete message. The AP can use this MIC to verify the PTK that was calculated by the STA by calculating the PTK and checking the validity of the MIC. The third message in the handshake is EAPOL-Key3 that contains the previously generated ANonce, a new MIC and an encrypted combination of the group key (GTK) and the RSNE. This group key can be used to encrypt broadcast data to all clients connected to the same AP. The encrypted RSNE can be used to compare it with the RSNE in the EAPOL-Key2 message. The client can confirm the verification by sending the EAPOL-Key4 message.

A visual representation can be found in figure 2.1.

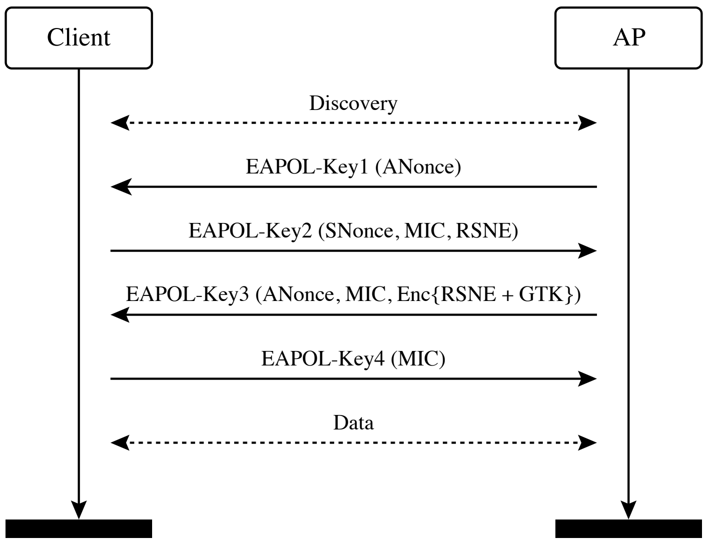
Client
AP
Discovery
EAPOL-Key1 (ANonce)
EAPOL-Key2 (SNonce, MIC, RSNE)
EAPOL-Key3 (ANonce, MIC, Enc{RSNE + GTK})
EAPOL-Key4 (MIC)
Data

Figure 2.1: The 4-Way Handshake

2.2 Tunneled Direct-Link Setup

2.2.1 General

The Tunneled Direct-Link Setup (TDLS) protocol allows two devices to create a direct connection. The stations should be and will stay connected to the same access point. The direct-link will reduce the amount of traffic transferred via the access point and avoids congestion inside that access point.

2.2.2 Establishing a direct-link

To establish a direct-link we need two stations connected to the same AP. Both stations should have to capability to setup a direct-link. The initiating station is called the *initiator*, the other station is the *responder*.

Before a link can be established the initiator will try to discover the capabilities of the responder station. For this a TDLS Discovery Request will be send from the initiator to the responder. If the responder is capable of setting up a TDLS connection it will respond with a TDLS Discovery Response send directly, outside the AP, to the initiator.

The initiator can now send the first message in the exchange to establish a new link. This message is send through the access point by the initiator to the responder, who proposes a direct-link based on the cryptographic capabilities of the two stations. The responder replies through the access point with a status code indicating success or failure in the setup response. The receival of this message is then confirmed by the last message in this part of the exchange. If any of the two stations wants to sever the direct link they can do this by sending a teardown message. This teardown message can both be send through the access point or directly to the receiving station. Figure 2.2 visualises this process.

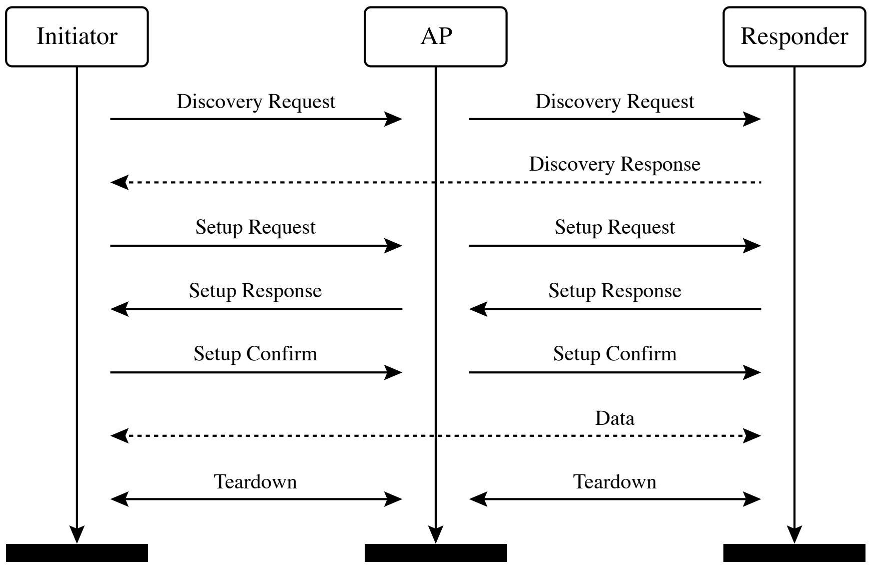

Figure 2.2: The TDLS direct-link establishment

2.2.3 TDLS PeerKey

A direct link connection can be secured via the TDLS PeerKey (TPK). This key is is used to provide a Robust Secure Network Associaton (RSNA) for the direct-link. This provides data origin authenticity of the setup messages and the confidentiality for data that will be send over the direct link.

The TPK is computed using fields from the messages in the TDLS establishment. Next, theses TPK can be used to calculate the Message Integrity Code (MIC) of the messages sent in the establishment.

The fields used in the calculation of the TPK are two nonces (SNonce and ANonce) and the MAC addresses of the initiator, responder and access point. These fields are secured by the existing RSNA of the STAs with the AP preventing eavesdropping by stations other than those already connected to the same AP. The handshake (figure 2.3) is started by the initiator who sends a chosen cipher suite (RSNE) and the SNonce to the responder station. The responder will use its own nonce to generate the TPK. This TPK is then used to calculate the MIC with the whole message as input. The initiator, who is now receiving both the nonces and a calculated MIC will derive the TPK as well and confirm that the MIC is valid. The responder will then send the 3rd message containing a new MIC to validate the direct link.

The TPK is calculated as follows:

$$Input = Hash(min(SNonce, ANonce)\|max(SNonce, ANonce)) \tag{2.1}$$

$$MACS = min(MAC_I, MAC_R)\|max(MAC_I, MAC_R)\|BSSID \tag{2.2}$$

$$TPK = KDF(Input, "TDLSPMK", MACS) \tag{2.3}$$

- The $SNonce$ and $ANonce$ are the values generated by the Initiator and Responder STA respectively.
- The MAC_I and MAC_R are the MAC addresses of the Initiator and Responder STA respectively.
- The $BSSID$ is the BSSID of the current BSS of the Initiator STA
- The KDF function is the 'KDF-Hash-Length' key derivation function defined in section 12.7.1.7.2 of the specification where $Hash$ function is the hash algorithm defined in the RSNE and $Length$ is the specified TK_bits value from the specification + 128. In our case these values are SHA256 and 256 bits respectively.

Only the first 16 bytes of the TPK are used to calculated the MIC. This part of the TPK is called the Key Confirmation Key (TPK-KCK). It provides data origin authenticity in the TDLS Setup Response and TDLS Setup Confirm messages. The second 16 bytes are the Temporal Key (TPK-TK) which will provide confidentiality for the direct-link between the two stations. The MIC in the TDLS handshake is always calculated using the AES-128-CMAC algorithm outputting 128 bits.

The MIC is calculated as follows:

$$TPK\text{-}KCK = TPK[0{:}16] \tag{2.4}$$

$$MIC = AES\text{-}128\text{-}CMAC(TPK\text{-}KCK, message)[0{:}16] \tag{2.5}$$

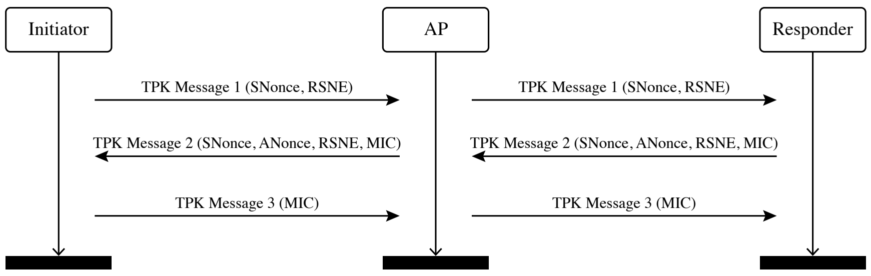
Initiator
AP
Responder
TPK Message 1 (SNonce, RSNE)
TPK Message 1 (SNonce, RSNE)
TPK Message 2 (SNonce, ANonce, RSNE, MIC)
TPK Message 2 (SNonce, ANonce, RSNE, MIC)
TPK Message 3 (MIC)
TPK Message 3 (MIC)

Figure 2.3: The TPK Handshake

Comparing the TPK and PTK

The resulting RSNA of the TDLS handshake is called a TDLS PeerKey Security Association (TPKSA). The RSNA of a station and an access point after the 4-way handshake is a Pairwise Transient Key Security Association (PTKSA).

Both RSNAs are bidirectional and the TPK and PTK are calculated in a similar way. The difference is in the length of the output. The TPK is 32 bytes, while the PTK is 64 bytes when the CCMP-128 cipher suite is specified in the RSNE.

The TPK is smaller since it only exists of the KCK (16 bytes) and TK (16 bytes). The PTK can be split as follows:

- A Key Confirmation Key (KCK, 16 bytes) to calculate a MIC
- A Key Encryption Key (KEK, 16 bytes) to encrypt data in the handshake (the GTK)
- A Temporal Key (TK, 16 bytes) to encrypt direct traffic between a client and the AP
- An Authenticator Transmission Key (8 bytes), only used for the TKIP mode
- An Authenticator Receiver Key (8 bytes), only used for the TKIP mode

A TPK will always use the CCMP-128 cipher suite and thus does not need the Authenticator keys. Neither does the protocol send data during the handshake, which explains why no KEK is required.

Figures 2.4 and 2.5 visualise the breakdown of the TPK and PTK respectively.

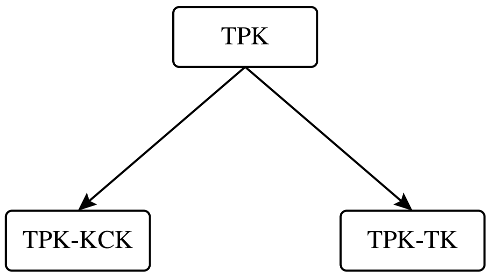

Figure 2.4: Breakdown of the TDLS PeerKey (TPK)

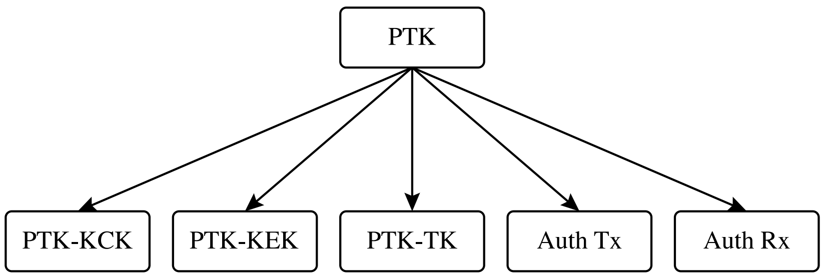

Figure 2.5: Breakdown of the Pairwise Transient Key (PTK)

2.3 Model Learning

2.3.1 Mealy machines

To model the implementation of TDLS we are going to use a specific kind of state machine, *Mealy machines*. This type of state machine is a kind of deterministic finite state machine that has a transition and output for every state and input [13].

A Mealy machine is formed by a set of finite states (Q) and the transitions between those states. One of these states is the initial state q_0, the state where the Mealy machine always starts. Transitions represent an input received while the machine is one of itś states. The transitions are defined as $\delta : Q \times \Sigma \rightarrow Q$. This function uses the input alphabet Σ, which is a finite set of symbols that represent the input. Different from a normal finite state machine is the output function $\lambda = Q \times \Sigma \rightarrow \Lambda$ where Λ is the finite set of output symbols.

Since a Mealy machine is deterministic the state machine has only one transition for each input. This is perfect for our research since TDLS should reply the same every time the same message sequence is executed.

2.3.2 Learning state machines

Our goal is to learn a state machine of a TDLS implementation. We are approaching this by using the L* algorithm [3]. The goal of this algorithm

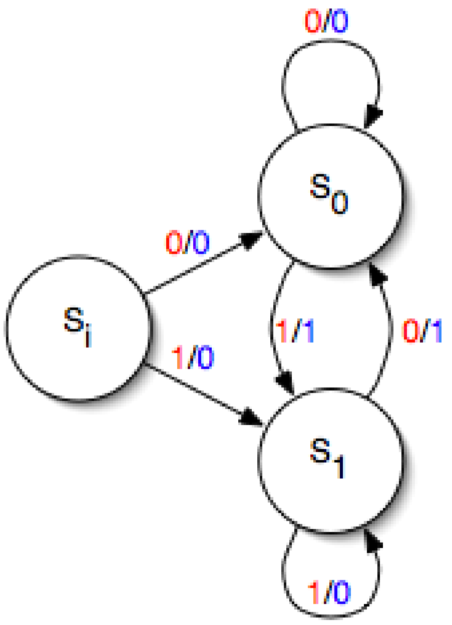

Figure 2.6: Example of a Mealy machine

is to learn a finite state machine from a so-called *System Under Learning* (SUL). This SUL can be any type of computational system. The L* algorithm is run by the learner, who tries to communicate with the SUL to infer the state machine of that SUL. In our learning process the SUL will be an implementation of TDLS.

Learning process

The first step in the learning process is sending arbitrary messages to the SUL utilising the L* algorithm. The SUL will answer based on the message sent by the learner. These answers can differ for each type of message sent by the learner. In between messages the learner will reset the state of the SUL by sending a special reset message. This allows the learner to build a hypothetical state machine based on those messages, both the input and output. Since the state machine of the learner is only hypothetical it needs to check with the SUL if it matches the actual state machine.

Checking if the state machines match is done by using an equivalence algorithm, in our case we choose to use *randomwords*. This algorithm will send a random sequence of inputs of a predefined minimum and maximum length to the SUL. If the output of the SUL based on this input matches the inferred state machine the next sequence of inputs will be sent. After a predefined number of tries the learner will draw the conclusion that the inferred state machine is probably equal to the state machine of the SUL.

The randomwords algorithm never gives you 100% certainty that the state machine is correct. In the case that the learner will find a counter example of the inferred state machine it will go back to the first step to update the hypothesis.

To make this all possible we need a mapper to convert the arbitrary messages from the learner to messages that the SUL can understand and the other way around so that the learner can understand the response by the SUL. This mapper is situated between the learner and the SUL.

2.4 Tooling

2.4.1 LearnLib

The first of the tools used to execute our research is LearnLib[1]. This is, as they call it, "an open framework for automata learning". It is free and open source under the Apache 2.0 License. The development is executed at the Chair for Programming Systems of TU Dortmund University in Germany, where it was introduced by Malte et al. [11]. LearnLib features both multiple learning algorithms and multiple strategies to approximate equivalence, of which L* and random words are used in this thesis. The version used in this thesis, 0.12.0, was released in June 2015. The latest version (0.13.1) was released in May 2018 since development was picked up again. However, since we are not using LearnLib directly but via StateLearner we are not able to use the latest version.

2.4.2 StateLearner

The second tool is called StateLearner[2], developed by Joeri de Ruiter. This tool is based on LearnLib and is tailored to automated model learning. It connects to the SUL either directly or via a mapper. In our case a mapper takes the symbols of the input alphabet that we provide StateLearner with and maps those symbols to the appropriate messages. The responses are then once again mapped to symbols that form the output alphabet. This way we can use StateLearner to form hypotheses using LearnLib.

[1]https://learnlib.de/

[2]https://github.com/jderuiter/statelearner

Chapter 3

Research

This chapter will explain the setup used to infer the state machine of a TDLS implementation. We will take a look at the mapper, which is used to convert input and output symbols used by StateLearner to the right TDLS messages. Next we will focus on the assumptions made for this research, plus the final setup used.

3.1 Mapper

As previously mentioned StateLearner requires a mapper to send and receive messages to the TDLS implementation. First we will explore two tools used in our mapper: Scapy and pycrypto. Then we will look how these tools are used in our mapper.

3.1.1 Scapy

Scapy[1] is a Python library that allows you to capture, manipulate and send network packets. This open source library was used to create the 802.11 TDLS packets and wrap them inside Ethernet frames. Scapy itself has support for basic 802.11 packets that are mentioned in the specification. It does not implement the exact messages that are used for TDLS however. This means that we needed to implement all messages used for TDLS from scratch using the building blocks that Scapy provides. This includes the Setup Request, Setup Response, Setup Confirm and Teardown messages that we previously discussed in section 2.2.2.

3.1.2 Cryptographic utilities

The construction of TDLS messages involves cryptographic operations. These operations, hashing algorithm SHA256 and the AES encryption algorithm

[1]https://github.com/secdev/scapy

are not natively implemented in Python itself, therefore we need an external implementation. The solution we are using for our research is the Python Cryptography Toolkit[2] or pycrypto for short. This toolkit, also open source, implements multiple secure hash functions and various encryption algorithms and it has been used in this type of research before [21, 22].

3.1.3 Implementation

The implementation of the mapper was written in Python to make use of the previously mentioned tools. It exposes a socket connection to the learner to enable communication between the mapper and the learner. The input symbols determined by the learner are sent over the socket and transcribed into the right TDLS messages. These messages are then send on the network interface of the learner side in the network. The side of the SUL gets the time to answer and the mapper will subsequently translate the answer to the right output symbols. The output symbol will then be send back to the learner and the mapper will be made ready for the next input symbol. If the mapper receives a setup response message from the implementation the contents of the message will be saved in the mapper. This message contains the ANonce, SNonce, BSSID and MAC addresses needed to create the right TPK for the connection. That TPK is used if the next message by the learner is a setup confirm message. Any other symbol, except for the connection check, from the learner will reset these values following the specification. The mapper will always attempt to setup a secured direct-link and thus will always include values for the ANonce, SNonce and RSNE fields if possible.

Our input alphabet exists of the following symbols:

- SETUP_REQUEST - Translates to a TDLS setup request message
- SETUP_CONFIRM - Translation to a TDLS setup confirm message
- TEARDOWN - Translates to a TDLS teardown message
- CONNECTED - Translates to the internal mapper connection status

The output alphabet is defined as follows:

- SETUP_RESPONSE - Translation from a successful TDLS setup response message
- NO_RESPONSE - Indicates that no response was received
- CONNECTED - Indicates that the mapper detected a completed setup
- NOT_CONNECTED - Indicates that the mapper has not detect a completed setup

[2]https://github.com/dlitz/pycrypto

3.2 Setup

3.2.1 wpa_supplicant

The implementation we analysed in our research is the wpa_supplicant[3] software written by Jouni Malinen. This Linux user space 802.11 client is used in all major Linux distributions and mobile operating system Android. Along with wpa_supplicant comes the access point software called hostapd. Since the TDLS protocol requires an access point we will utilise this software in our setup to create a virtual AP. However, our research could be conducted with any type of access point implementation.

We will use version 2.7 of hostapd and wpa_supplicant to conduct our research. This version was released in December 2018.

3.2.2 Networking

Before we can run the mapper and learner to infer the state machine we will have to setup a simulated network environment. Since the wpa_supplicant source code already includes automated tests that use such a simulated environment we will be partially reusing this implementation. The environment requires a special build of both wpa_supplicant and hostapd which can be constructed by following the instructions in the repository[4]. Our test will use the mac80211_hwsim Linux kernel driver which is capable of simulating 802.11 hardware. We setup three interfaces: one WPA2 protected access point and two stations. The access point is running hostapd, the stations are both controlled by an instance of wpa_supplicant. However, the instance on the initiator end is only used for the virtual interface and does not respond to messages during our research. Since we know the MAC addresses of the access point and both stations we are able to craft messages and send them via the AP to the receiving wpa_supplicant instance. The response of that instance can then be read by sniffing the interface of the interface we simulated to be the sender. Figure 3.1 shows a simplified visual representation of our setup.

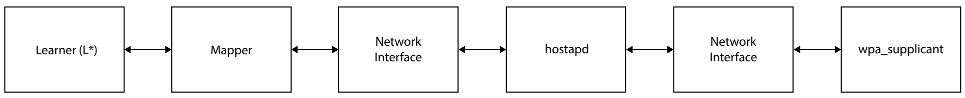

Figure 3.1: A visual representation of our setup

[3]https://w1.fi/

[4]https://w1.fi/cgit/hostap/tree/tests/hwsim/README

3.2.3 Learner settings

As we mentioned previously we will use the L* and randomwords algorithms for the learning process. We will setup the equivalence algorithm to use a minimum and maximum length of 5 and 10 respectively. We will require 5000 queries to prove that no counterexample can be found.

Chapter 4

Results

In this chapter we state our expectations of the inferred model. Next we evaluate the results of the learner by comparing them to our expectations using a manual analysis.

4.1 Expectations

We expect that the TDLS state machine has three states:

1. No TDLS connection
2. Setup in progress
3. Active TLDS connection

If we follow the 802.11 specification we should have at least the following edges:

- State 0 to 1: SETUP_REQUEST / SETUP_RESPONSE_OK - TDLS Setup Request resulting in a success TDLS Setup Response ([1, 11.23.4])
- State 1 to 1: SETUP_REQUEST / SETUP_RESPONSE_OK - TDLS Setup Request resulting in a success TDLS Setup Response ([1, 11.23.4]), since any new Setup Request resets the process and initialises a new handshake
- State 1 to 2: SETUP_CONFIRM / NO_RESPONSE - TDLS Setup Confirm without response
- State 0, 1, 2 to 0: TEARDOWN / NO_RESPONSE: TDLS Teardown without response
- State 2 to 1: SETUP_REQUEST / SETUP_RESPONSE_OK - TDLS Setup Request resulting in a success TDLS Setup Response ([1, 11.23.4 sub e]), this is equivalent to a Teardown followed by a Setup Request

For any situation where the input symbol should not result in a state change we should see a self loop. These situations are not described in the specification and should thus do nothing.

In our mapper we have made assumptions on the establishment of a TDLS connection since we could not find a way to test the established connection. It assumes a successful connection and disconnection after a correct handshake and a sent teardown respectively. The initialisation of a new connection will also assume a disconnect, as per the specification. We have tried to send pings over the TDLS connection to confirm a successful setup, but the virtual interfaces did not seem to support this usage. Another solution we tried was doing a TDLS channel switch, however the interface does not offer this functionality either. This exhausted our options to test the connection, so we settled on keeping an internal state.

Exposing the internal state to the learner means that we should also expect the following state changes:

- State 0 to 0: CONNECTED / NOT_CONNECTED - Indicates that no successful setup was made
- State 1 to 1: CONNECTED / NOT_CONNECTED - Indicates that no successful setup was made
- State 2 to 2: CONNECTED / CONNECTED - Indicates that a successful setup was made

Figure 4.1 gives a visual representation of our expectations.

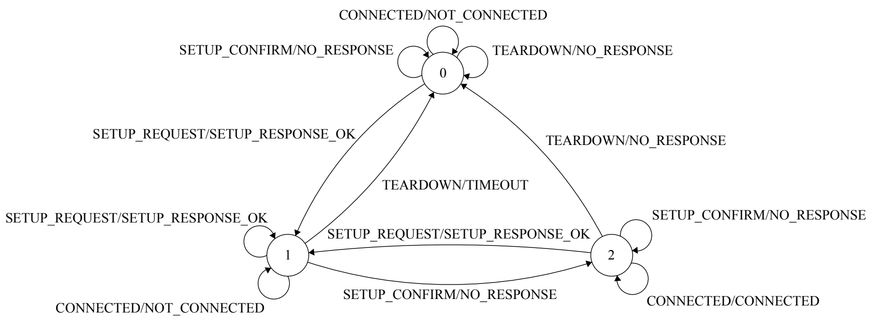
CONNECTED/NOT_CONNECTED
SETUP_CONFIRM/NO_RESPONSE
TEARDOWN/NO_RESPONSE
0
SETUP_REQUEST/SETUP_RESPONSE_OK
TEARDOWN/NO_RESPONSE
TEARDOWN/TIMEOUT
SETUP_REQUEST/SETUP_RESPONSE_OK
SETUP_CONFIRM/NO_RESPONSE
SETUP_REQUEST/SETUP_RESPONSE_OK
1
2
SETUP_CONFIRM/NO_RESPONSE
CONNECTED/NOT_CONNECTED
CONNECTED/CONNECTED

Figure 4.1: Expected TDLS State Machine

4.2 Analysis

The state machine learned by the L* algorithm (4.2) does not deviate from the previously mentioned requirements of the implementation. There are no state changes that we did not expect. However, this behaviour may be caused by the internal state that the mapper holds to determine if a successful connection was made.

The complete process of learning and checking for counter-examples took a total of 8.6 hours. Of which the learning part is only 8 minutes. This seems logical since the second hypothesis by the learner is the final result that the statelearner outputs. Since the process has a total of 5010 equivalence queries we can calculate that the first counter-example was found after 10 queries, since our settings require 5000 equivalence queries before the learner decides that it is done.

The learning process is stable, running the process succeeds when the computer running is prevented from going to sleep or shutting down. During these processes we have kept watch by occasionally checking the network traffic on the computer for problems. Since our mapper does not process errors returned by the implementation and ignores re-transmissions it is possible that messages were missed but this does not seem the case. The mapper does not have to wait for response packets and is able to respond well within any timeouts set by the IEEE 802.11 specification.

We did try to change our setup to include sending and processing malformed messages but eventually decided to leave this out of the mapper since the learning process could not be completed. We suspect that some kind of non-determinism in the implementation could have caused this.

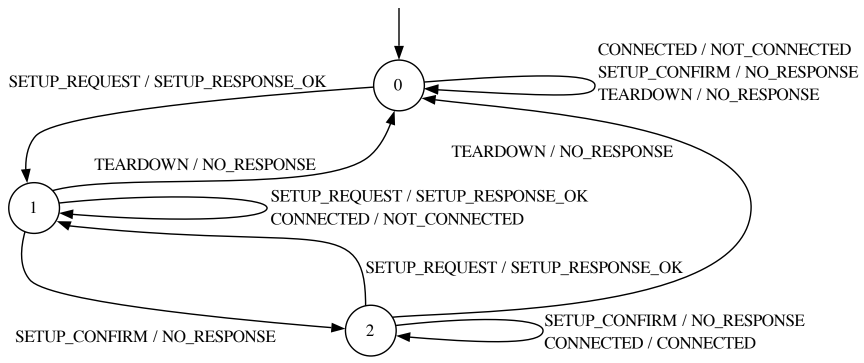
0
CONNECTED / NOT_CONNECTED
SETUP_CONFIRM / NO_RESPONSE
TEARDOWN / NO_RESPONSE
SETUP_REQUEST / SETUP_RESPONSE_OK
TEARDOWN / NO_RESPONSE
TEARDOWN / NO_RESPONSE
1
SETUP_REQUEST / SETUP_RESPONSE_OK
CONNECTED / NOT_CONNECTED
SETUP_REQUEST / SETUP_RESPONSE_OK
SETUP_CONFIRM / NO_RESPONSE
2
SETUP_CONFIRM / NO_RESPONSE
CONNECTED / CONNECTED

Figure 4.2: Learned TDLS State Machine

4.3 Interesting findings

Since our model exactly matched our expectations we investigated the tests that come bundled with wpa_supplicant.

First we found that the tests that are executed to validate the implementation of TDLS do not actually check the connection that is established. The principle used in this research is the same as the principle used in the tests: a happy flow is a successful connection. The tests will detect if errors are encountered and fail, however the connection itself is no more than checking if a setup confirm message is sent.

Our second finding is related to the entire TDLS implementation. There are tests written to verify the workings of the TDLS channel switch, however the drivers used to execute these tests do no support this functionality. This means the test is not useful. After further inspection it seems the test is not enabled to prevent it from failing. This could mean that the implementation is not completely tested and may still include bugs.

Chapter 5

Related Work

In this chapter we give an overview of previous work related to the security of Wi-Fi and/or the use of models to test or infer state machines.

5.1 Wi-Fi

The authentication of clients with access points within the 802.11 specification, Wi-Fi, is facilitated by the so-called 4-way handshake. Closely related to this thesis is the research into both the manual [21] and automated [12] state learning of that handshake.

The 4-Way Handshake has also been subject of numerous formal analyses [9, 6, 23, 10]. Other parts of 802.11, but related to the 4-way handshake, that have been the subject of research are the WEP and WPA2 TKIP security mechanisms [8, 17].

More in-depth research into the handshake discovered a vulnerability in the transmission of the group-key [18]. This research by Vanhoef et al. forced RC4 encryption of the group key, which is insecure [8]. Following this discovery Vanhoef et al. introduced an attack that re-installs the key used by the 4-way handshake making replaying, decryption and forging of packets possible [19].

In October 2018 Vanhoef et al. presented a new paper [20] based on their previous work on the 4-way handshake including an attack on the TPK with a possibility of re-installation of the key as well.

5.2 Learning state machines

Past work relying on state machine learning has already been mentioned in the previous section. We have seen examples of both manual and automated modeling of the 4-way handshake [21, 12]. However, there is other research that analyses protocols by inferring the state machines. Aarts et al. [2]

used model-based learning to infer to state machines of EMV cards. Additionally, hand-held readers use for internet banking were subject of research [4] along with the TLS protocol [5, 16]. These analyses discovered several security flaws in different implementations of TLS. Other implementations of protocols analysed via this technique are SSH [14, 15] and IPSec [22].

Furthermore, the technique was also used for a non-security related subject by learning the TCP network protocol [7]. This revealed ways to fingerprint remote operating systems.

Chapter 6

Conclusions

In this research we have shown that this method of inferring the TDLS state machine works to a certain point. Our approach has shown that the basic TDLS handshake protocol can be inferred as long as assumptions are introduced. We have shown that by sending correct TDLS messages no error responses will be triggered by wpa_supplicant. This could mean that the implementation is working as expected.

Our work can be used as a basis for future investigation. We have successfully implemented the messages used by TDLS to communicate by only using Scapy and cryptographic tools. This means that using a learner and Python tools is a suitable approach to this kind of research for other protocols in the 802.11 specification.

Lastly, this research has shown that automatically inferring state machines can improve our knowledge about the inner workings of protocols, in our case TDLS.

Chapter 7

Future work

Our research can be used a basis for future investigation into this topic. We will point out several possibilities.

In our research we have not taken error messages or timeouts into account. This was to simplify our research. Better results may be found when these factors are introduced. One of the ways to do this would be to use an improved version of the StateLearner[1] software that we used.

Another improvement would be adding a malformed message to the input symbols. We believe that adding this type of message would improve the resulting state machine. Error messages can reveal useful information about the implementation and the way it is implemented. During our testing of the setup request message we noticed that the time to respond from the wpa_supplicant TDLS handshake implementation took significant less time in the case of a malformed message. Thus it is possible that our research missed mistakes in the implementation, even though we always sent correctly formatted messages.

Using fuzzing the messages could also be modified to find states that we did not find in our research. Fuzzing could introduce more kinds of error messages. The learner will thus learn about the different variations of responses.

One more enhancement would be to execute this research on real hardware instead of using a simulated environment. This would introduce more new factors: packet loss and interference. It might however enable us to use the TDLS channel switch to confirm a successful connection. We think this would greatly improve the resulting model.

Lastly we think that adding a ping to confirm a successful connection would improve the results by removing the assumptions we had to make.

[1]https://github.com/ChrisMcMStone/statelearner

Bibliography

[1] 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 802.11-2016 (Revision of IEEE Std 802.11-2012)*, Dec 2016.

[2] Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 461–468. IEEE Computer Society, 2013.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[4] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. USENIX Association, 2014.

[5] Joeri de Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206. USENIX Association, 2015.

[6] Ling Dong, Ke-fei Chen, and Xue-jia Lai. Formal analysis of authentication in 802.11i. *Journal of Shanghai Jiaotong University (Science)*, 14(1):122–128, Feb 2009.

[7] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the TCP Network Protocol. In *Formal Methods for Industrial Critical Systems: 19th International Conference*, pages 78–93. Springer International Publishing, 2014.

[8] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weakness in the Key Scheduling Algorithm of RC4. volume 2259, 08 2001.

[9] Changhua He and John C. Mitchell. Analysis of the 802.11i 4-way Handshake. In *Proceedings of the 3rd ACM Workshop on Wireless Security*, WiSe '04, pages 43–50. ACM, 2004.

[10] Changhua He and John C Mitchell. Security analysis and improvements for IEEE 802.11i. In *In Proceedings of the 12th Annual Network and Distributed System Security Symposium*, pages 90–110, 2005.

[11] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495. Springer International Publishing, 2015.

[12] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *23rd European Symposium on Research in Computer Security, ESORICS 2018*, pages 325–345. Springer, 2018.

[13] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept 1955.

[14] Fiterău-Broştean Paul, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151. ACM, 2017.

[15] Max Tijssen. Automatic modeling of SSH implementations with state machine learning algorithms. Bachelor's thesis, Radboud University, 2015.

[16] Jules van Thoor. Learning state machines of TLS 1.3 implementations. Bachelor's thesis, Radboud University, 2018.

[17] Mathy Vanhoef and Frank Piessens. Practical verification of wpa-tkip vulnerabilities. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 427–436. ACM, 2013.

[18] Mathy Vanhoef and Frank Piessens. Predicting, decrypting, and abusing WPA2/802.11 group keys. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 673–688. USENIX Association, 2016.

[19] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1313–1328. ACM, 2017.

[20] Mathy Vanhoef and Frank Piessens. Release the Kraken: new KRACKs in the 802.11 Standard. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[21] Mathy Vanhoef, Domien Schepers, and Frank Piessen. Discovering Logical Vulnerabilities in the Wi-Fi Handshake Using Model-Based Testing. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 360–371. ACM, 2017.

[22] Bart Veldhuizen. Automated state machine learning of IPsec implementations. Bachelor's thesis, Radboud University, 2017.

[23] X. Xing, E. Shakshuki, D. Benoit, and T. Sheltami. Security analysis and authentication improvement for IEEE 802.11i specification. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*, pages 1–5, Nov 2008.

Appendices

Appendix A

Mapper implementation

Listing A.1: requirements.txt

```
adns-python==1.2.1
anyjson==0.3.3
argcomplete==1.8.1
asn1crypto==0.24.0
attrs==17.4.0
Automat==0.6.0
backdoor-factory==0.0.0
backports-abc==0.5
backports.functools-lru-cache==1.4
backports.shutil-get-terminal-size==1.0.0
backports.ssl-match-hostname==3.5.0.1
BBQSQL==1.0
bcrypt==3.1.4
bdfproxy==0.0.0
BeautifulSoup==3.2.1
beautifulsoup4==4.6.0
BlindElephant==1.0
blinker==1.4
capstone==3.0.4
certifi==2018.1.18
chardet==3.0.4
CherryTree==0.38.4
## FIXME: could not find svn URL in dependency_links for this
    package:
chirp===daily-20170714
click==6.7
colorama==0.3.7
ConfigArgParse==0.11.0
configobj==5.0.6
configparser==3.5.0
constantly==15.1.0
construct==2.8.16
couchdbkit==0.6.5
cryptography==2.1.4
cycler==0.10.0
decorator==4.1.2
```

```
dicttoxml==1.7.4
distorm3==3.3.4
dnslib==0.9.7
dnspython==1.15.0
docutils==0.14
easygui==0.96
EditorConfig==0.12.1
Elixir==0.7.1
enum34==1.1.6
et-xmlfile==1.0.1
feedparser==5.2.1
Flask==0.12.2
funcsigs==1.0.2
functools32==3.2.3.post2
fuse-python==0.3.0
future==0.15.2
futures==3.2.0
GDAL==2.2.4
GeoIP==1.3.2
gevent==1.2.2
greenlet==0.4.12
h2==3.0.1
hpack==3.0.0
html2text==2018.1.9
html5lib==0.999999999
http-parser==0.8.3
httplib2==0.9.2
httpretty==0.8.14
hyperframe==5.1.0
hyperlink==17.3.1
idna==2.6
impacket==0.9.15
incremental==16.10.1
ipaddress==1.0.17
IPy==0.83
ipython==5.5.0
ipython-genutils==0.2.0
itsdangerous==0.24
jdcal==1.0
Jinja2==2.10
jsbeautifier==1.6.4
jsonpickle==0.9.5
jsonrpclib==0.1.7
keepnote==0.7.8
keyring==10.6.0
keyrings.alt==3.0
killerbee==1.0
lxml==4.2.1
M2Crypto==0.27.0
Mako==1.0.7
MarkupSafe==1.0
matplotlib==2.1.1
mechanize==0.2.5
mercurial==4.5.3
```

```
metaconfig==0.1.4a1
mockito==0.5.2
msgpack==0.5.6
mysqlclient==1.3.10
nassl==0.12
netaddr==0.7.19
NfSpy==1.0
numpy==1.13.3
olefile==0.45.1
openpyxl==2.4.9
OWSLib==0.16.0
PAM==0.4.2
paramiko==2.4.0
passlib==1.7.1
pathlib2==2.3.2
pcapy==0.10.8
peepdf==0.4.1
pefile==2017.11.5
pexpect==4.2.1
pickleshare==0.7.4
Pillow==4.3.0
pluggy==0.6.0
ply==3.11
prettytable==0.7.2
prompt-toolkit==1.0.15
psycopg2==2.7.4
py==1.5.3
pyasn1==0.4.2
pyasn1-modules==0.2.1
pycairo==1.16.2
pycrypto==https://github.com/dlitz/pycrypto/archive/v2.7a1.zip
pycryptodomex==3.4.7
pycurl==7.43.0.1
pydns==2.3.6
pyenchant==2.0.0
Pygments==2.2.0
pygobject==3.28.2
pygtkspellcheck==4.0.5
pyinotify==0.9.6
PyJWT==1.5.3
pymongo==3.6.1
pymssql==2.1.3
PyNaCl==1.2.1
pyOpenSSL==17.5.0
pyparsing==2.2.0
PyPDF2==1.26.0
pyperclip==1.6.0
pyproj==1.9.5.1
pyrit==0.5.1
pyscard==1.9.6
pyserial==3.4
pysmi==0.2.2
pysnmp==4.4.3
pysnmp-apps==0.3.2
```

```
pysnmp-mibs==0.1.3
PySocks==1.6.5
pyspatialite==3.0.1
pysqlite==2.7.0
pytest==3.3.2
python-dateutil==2.6.1
python-Levenshtein==0.12.0
python-magic==0.4.16
pythonaes==1.0
pytz==2018.4
pyusb==1.0.0b2
PyX==0.12.1
pyxdg==0.25
PyYAML==3.12
qrcode==5.3
requests==2.18.4
restkit==4.2.2
rfidiot==1.0
roman==2.0.0
scandir==1.7
scapy==2.3.3
SecretStorage==2.3.1
service-identity==16.0.0
Shapely==1.6.4
simplegeneric==0.8.1
simplejson==3.13.2
singledispatch==3.4.0.3
six==1.11.0
slowaes==0.1a1
socketpool==0.5.3
SQLAlchemy==1.2.5
subprocess32==3.2.7
tcpwatch==1.3.1
tornado==5.0.2
traitlets==4.3.2
Twisted==17.9.0
typing==3.6.4
unicodecsv==0.14.1
urllib3==1.22
urwid==2.0.1
uTidylib==0.3
vinetto==0.7b0
volatility==2.6
wapiti==2.3.0
wcwidth==0.1.7
webencodings==0.5
webunit==1.3.10
Werkzeug==0.14.1
wfuzz==2.2.9
Whoosh==2.7.4
wxPython==3.0.2.0
wxPython-common==3.0.2.0
XlsxWriter==0.9.6
xmlbuilder==1.0
```

```
yara-python==3.7.0
zenmap==7.70
zim==0.68
zope.interface==4.3.2
```

Listing A.2: packets.py

```
from scapy.all import *
import binascii
from util import *

class Dot11TDLSAction(Packet):
    name = "802.11 TDLS Action Frame"
    fields_desc=[
        ByteField("payload_type", 2),
        ByteField("category_code", 12),
        ByteEnumField("action", 1, { 0: "setup request", 1: "
            setup response" , 2: "setup confirm", 3: "teardown",
            10: "discovery request" } ),
        ConditionalField(ShortField("status_code", 0), lambda
            pkt:pkt.action>=1 and pkt.action<=3),
        ConditionalField(ByteField("dialog_token", 1), lambda
            pkt:pkt.action!=3),
    ]

class Dot11Cap(Packet):
    """ Our own definition for the supported rates field """
    name = "802.11 Capabilities Information Element"
    fields_desc = [
        BitField("channelAgility", 0, 1),
        BitField("pbcc", 0, 1),
        BitField("short_preamble", 1, 1),
        BitField("privacy", 0, 1),
        BitField("cfPollRequest", 0, 1),
        BitField("cfPollable", 0, 1),
        BitField("ibss", 0, 1),
        BitField("ess", 0, 1),
        BitField("iba", 0, 1),
        BitField("dba", 0, 1),
        BitField("dsss_ofdm", 0, 1),
        BitField("rm", 0, 1),
        BitField("apsd", 0, 1),
        BitField("sst", 1, 1),
        BitField("cfpReserved", 0, 1),
        BitField("spectrum", 0, 1),
    ]

class Dot11EltRates(Packet):
    """ Our own definition for the supported rates field """
    name = "802.11 Rates Information Element"
    # We support all the rates
    supported_rates = [0x02, 0x04, 0x0b, 0x16, 0x0c, 0x12, 0x18]
    fields_desc = [ByteField("ID", 1), ByteField("len", len(
        supported_rates))]
    for index, rate in enumerate(supported_rates):
        fields_desc.append(ByteField("supported_rate{0}".format(
            index + 1),
                                     rate))

class Dot11EltExtRates(Packet):
    """ Our own definition for the supported rates field """
    name = "802.11 Extended Rates Information Element"
    # We support all the rates
    supported_rates = [0x24, 0x30, 0x48, 0x60, 0x6c]
    fields_desc = [ByteField("ID", 50), ByteField("len", len(
        supported_rates))]
    for index, rate in enumerate(supported_rates):
        fields_desc.append(ByteField("extended_supported_rate{0}
            ".format(index + 1),
                                     rate))

class Dot11EltChannels(Packet):
    name = "802.11 Channels Information Element"
    fields_desc = [
        ByteField("ID", 36),
        ByteField("len", 2),
        ByteField("channel", 1),
        ByteField("range", 11),
    ]

class Dot11EltLinkIdentifier(Packet):
    name = "802.11 Link Identifier Element"
    fields_desc = [
        ByteField("ID", 101),
        ByteField("len", 18),
        Dot11AddrMACField("bssid", ETHER_ANY),
        Dot11AddrMACField("initSta", ETHER_ANY),
        Dot11AddrMACField("respSta", ETHER_ANY)
    ]

class Dot11EltCustom(Packet):
    name = "802.11 Information Element"
    fields_desc = [ ByteField("ID", 0),
                    FieldLenField("len", None, "info", "B"),
                    StrLenField("info", "", length_from=lambda x
                        :x.len) ]

bind_layers( Ether, Dot11TDLSAction, {'type':0x890d} )
bind_layers( Dot11TDLSAction, Dot11Cap )
bind_layers( Dot11Cap, Dot11EltCustom )
bind_layers( Dot11EltCustom, Dot11EltCustom )

def create_rsn(gdcsType=4, cipherSuites=[4], akmSuites=[7]):
    info = '\x01\x00' #RSN Version 1
    info += '\x00\x0f\xac' + chr(7) #Group Cipher Suite : 00-0f-
        ac TKIP
    info += chr(len(cipherSuites)) + '\x00' #2 Pairwise Cipher
        Suites (next two lines)
    for suite in cipherSuites:
        info += '\x00\x0f\xac' + chr(suite)
    info += chr(len(akmSuites)) + '\x00' #1 Authentication Key
        Managment Suite (line below)
    for suite in akmSuites:
        info += '\x00\x0f\xac' + chr(suite)
    info += '\x0c\x02'

    return Dot11Elt(ID='RSNinfo', info=info) #RSN Capabilities (
        no extra capabilities)


def create_fte(SNonce=None, ANonce=None, mic=None):
    info = '\x00\x00' # MIC Control and element count
    info += bit128_to_hex(0) if mic is None else mic

    if SNonce is None:
        ANonce = bit256_to_hex(0)
    elif ANonce is None:
        ANonce = bit256_to_hex(random.randint(0, 2**256 - 1))
    info += ANonce

    if SNonce is None:
        SNonce = bit256_to_hex(random.randint(0, 2**256 - 1))
    info += SNonce

    return Dot11EltCustom(ID=55, info=info)

def create_ti(interval):
    info = '\x02'
    info += struct.pack('L', interval)
    return Dot11EltCustom(ID=56, info=info)
```

Listing A.3: tdls.py

```
from scapy.all import *
from packets import *
from util import *
import binascii
import hashlib
import hmac
from Crypto.Cipher import AES
from Crypto.Hash import CMAC

def _read_fte_values(info):
    mic_control = info[0:2]
    ANonce = info[18:50]
    SNonce = info[50:82]

    return ANonce, SNonce

def _read_link_id_values(info):
    BSSID = info[0:6]
    InitiatorMAC = info[6:12]
    ResponderMAC = info[12:18]

    return BSSID, InitiatorMAC, ResponderMAC

def read_tdls_setup_packet(packet):
    elts = packet[Dot11EltCustom]
    index = 0

    ANonce = None
    SNonce = None
    BSSID = None
    InitiatorMAC = None
    ResponderMAC = None

    while index > -1:
        try:
            elt_info = elts[index].info
            if elts[index].ID == 55:
                ANonce, SNonce = _read_fte_values(elt_info)
            if elts[index].ID == 101:
                BSSID, InitiatorMAC, ResponderMAC =
                    _read_link_id_values(elt_info)
            index += 1
        except IndexError:
            index = -1

    return ANonce, SNonce, BSSID, InitiatorMAC, ResponderMAC

def create_tdls_discovery(bssid="02:00:00:00:03:00",
    initSta="02:00:00:00:01:00", respSta="02:00:00:00:00:00"):
    return (Ether(src=initSta, dst=respSta, type=0x890d) /
        Dot11TDLSAction(action=10) /
        Dot11EltLinkIdentifier(bssid=bssid, initSta=initSta,
            respSta=respSta))

def create_tdls_setup_request(bssid="02:00:00:00:03:00",
    initSta="02:00:00:00:01:00", respSta="02:00:00:00:00:00",
    gdcs=None, malformed=False):

    raw = Raw(load='\x7f\x08\x00\x00\x00\x00\x20\x00\x00\x00')
    if malformed:
        raw = Raw(load='\x7f\x02\x10\x00\x00\x70\x20\x01\x01\x00
            ')

    packet = (Ether(src=initSta, dst=respSta, type=0x890d) /
                Dot11TDLSAction(action=0) /
            Dot11Cap() /
            Dot11EltRates() /
            Dot11EltExtRates() /
            raw /
            Dot11EltChannels(channel=2, range=12) /
            Dot11EltLinkIdentifier(bssid=bssid, initSta=initSta,
                respSta=respSta))
    if gdcs:
        packet /= create_rsn(gdcsType=gdcs)
        packet /= create_fte()
        packet /= create_ti(interval=43200)

    return packet

def create_tdls_setup_response(bssid="02:00:00:00:03:00",
    initSta="02:00:00:00:01:00", respSta="02:00:00:00:00:00",
    gdcs=None, success=True, requestPacket=None):
    status = 0 if success else 1
    packet = Ether(src=initSta, dst=respSta, type=0x890d) /
        Dot11TDLSAction(action=1, status_code=status)
    if success:
        packet /= Dot11Cap()
        packet /= Dot11EltRates()
        packet /= Dot11EltExtRates()
        packet /= Dot11EltChannels(channel=2, range=12)

        LinkIdEl = Dot11EltLinkIdentifier(bssid=bssid, initSta=
            initSta, respSta=respSta)
        packet /= LinkIdEl

        if gdcs and requestPacket:
            RSNEEl = create_rsn(gdcsType=gdcs)
            ANonce, SNonce, BSSID, InitiatorMAC, ResponderMAC =
                read_tdls_setup_packet(requestPacket)

            print('ANonce: {}\nSNonce: {}'.format(binascii.
                hexlify(ANonce), binascii.hexlify(SNonce)))

            tpk = calculate_tpk(ANonce, SNonce, BSSID,
                InitiatorMAC, ResponderMAC)
            FTEl = create_fte(SNonce=SNonce, ANonce=ANonce, mic=
                None)
            TimeoutEl = create_ti(interval=43200)
            mic = calculate_mic(tpk, InitiatorMAC, ResponderMAC,
                 3, LinkIdEl, RSNEEl, TimeoutEl, FTEl)
            packet /= RSNEEl
            packet /= create_fte(SNonce=SNonce, ANonce=ANonce,
                mic=mic)
            packet /= TimeoutEl
    return packet

def get_tpk_from_setup_packet(packet):
    ANonce, SNonce, BSSID, InitiatorMAC, ResponderMAC =
        read_tdls_setup_packet(packet)
    return calculate_tpk(ANonce, SNonce, BSSID, InitiatorMAC,
        ResponderMAC)

def calculate_tpk(ANonce, SNonce, BSSID, InitiatorMAC,
    ResponderMAC):
    # TPK-Key-Input = SHA-256(min(SNonce, ANonce) || max(SNonce,
        ANonce))
    tpkKeyInput = hashlib.sha256(min(ANonce , SNonce) + max(
        ANonce, SNonce)).digest()
    print("TDLS TPK-Key-Input: {}".format(binascii.hexlify(
        tpkKeyInput))) #CORRECT!

    # TPK-Key-Data = KDF-N_KEY(TPK-Key-Input, "TDLS PMK", min(
        MAC_I, MAC_R) || max(MAC_I, MAC_R) || BSSID)
    context = min(InitiatorMAC , ResponderMAC) + max(
        InitiatorMAC, ResponderMAC) + BSSID
    print("TDLS KDF Context: {}".format(binascii.hexlify(context
        )))
    # TPK = KDFHashLength(keyInput, "TDLS PMK", context)
    tpk = KDFSHA256(tpkKeyInput, b"TDLS PMK", context)
    tpkkck = tpk[0:16]
    tpktk = tpk[16:]

    print('TDLS TPK-KCK: {}\nTDLS TPK-TK: {}'.format(binascii.
        hexlify(tpkkck), binascii.hexlify(tpktk)))

    return tpk

def calculate_mic(TPK, InitiatorMAC, ResponderMAC, TransSeqNr,
    LinkIdEl, RSNEEl, TimeoutEl, FTEl):
    frame = InitiatorMAC + ResponderMAC + struct.pack('<B',
        TransSeqNr) + str(LinkIdEl) + str(RSNEEl) + str(TimeoutEl
        ) + str(FTEl)
    mic = CMAC.new(TPK[0:16], ciphermod=AES)
    mic.update(frame)
    mic_digest = mic.digest()[0:16]
    print("TDLS MIC: {}".format(binascii.hexlify(mic_digest)))
    return mic_digest

def create_tdls_setup_confirm(bssid="02:00:00:00:03:00",
    initSta="02:00:00:00:01:00", respSta="02:00:00:00:00:00",
    gdcs=None, responsePacket=None):
    status = 0
    packet = Ether(src=initSta, dst=respSta, type=0x890d) /
        Dot11TDLSAction(action=2, status_code=status)
    LinkIdEl = Dot11EltLinkIdentifier(bssid=bssid, initSta=
        initSta, respSta=respSta)
    packet /= LinkIdEl

    if gdcs and responsePacket:
        RSNEEl = create_rsn(gdcsType=gdcs)
        ANonce, SNonce, BSSID, InitiatorMAC, ResponderMAC =
            read_tdls_setup_packet(responsePacket)

        print('ANonce: {}\nSNonce: {}'.format(binascii.hexlify(
            ANonce), binascii.hexlify(SNonce)))

        tpk = calculate_tpk(ANonce, SNonce, BSSID, InitiatorMAC,
            ResponderMAC)
        FTEl = create_fte(SNonce=SNonce, ANonce=ANonce, mic=None
            )
        TimeoutEl = create_ti(interval=43200)
        mic = calculate_mic(tpk, InitiatorMAC, ResponderMAC, 3,
            LinkIdEl, RSNEEl, TimeoutEl, FTEl)
        packet /= RSNEEl
        packet /= create_fte(SNonce=SNonce, ANonce=ANonce, mic=
            mic)
        packet /= TimeoutEl

    return packet

def create_tdls_teardown(bssid="02:00:00:00:03:00",
    initSta="02:00:00:00:01:00", respSta="02:00:00:00:00:00"):
    return Ether(src=initSta, dst=respSta, type=0x890d) /
        Dot11TDLSAction(action=3) / Dot11EltLinkIdentifier(bssid=
        bssid, initSta=initSta, respSta=respSta)

def create_ping_message(response=False):
    bssid, addr0, addr1 = "20:00:00:00:03:00", "
        20:00:00:00:01:00", "f0:18:98:46:e5:9a"
    packet = Ether(src=addr0, dst=addr1, type=0x0006) / Raw(load
        ="HELLO_RESP")
    if response:
        packet = Ether(src=addr1, dst=addr0, type=0x0006) / Raw(
            load="HELLO_INIT")
    return packet
```

Listing A.4: run$_m$*apper.py*

```
#!/usr/bin/env python2.7

from multiprocessing.pool import ThreadPool

import socket

from scapy.all import *
import time
import logging
import time
logger = logging.getLogger()
import subprocess

import sys
sys.path.insert(0, "../hostap/tests/hwsim")
sys.path.insert(0, "../hostap/wpaspy")

from packets import *
from tdls import *

def do_sniff():
    return sniff(iface='wlan1', timeout=1, lfilter = lambda x:
        Dot11TDLSAction in x)

def main():
    verbose = False
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8888))
s.listen(1)

(client, address) = s.accept()

pool = ThreadPool(processes=1)

setup_response_packet = None
tpk = None
connected = False

bssid, addr0, addr1 = "20:00:00:00:03:00", "
    20:00:00:00:01:00", "20:00:00:00:00:00"

while True:
    cmd = client.recv(1024).strip()
    print("Received command: {}".format(cmd))

    async_sniff = pool.apply_async(do_sniff)

    timeout_response = "NO_RESPONSE"

    if cmd == "SETUP_CONFIRM":
        pkt = setup_response_packet
        if setup_response_packet and setup_response_packet[
            Dot11TDLSAction].status_code == 0:
            connected = True
        else:
            pkt = None
        sendp(create_tdls_setup_confirm(gdcs=4,
            responsePacket=pkt), iface='wlan1', verbose=
            verbose)
        setup_response_packet = None
    elif cmd == "SETUP_REQUEST_OPEN_CORRECT":
        setup_response_packet = None
        connected = False
        sendp(create_tdls_setup_request(gdcs=None), iface='
            wlan1', verbose=verbose)
    elif cmd == "SETUP_REQUEST_AES_CORRECT":
        setup_response_packet = None
        connected = False
        sendp(create_tdls_setup_request(gdcs=4), iface='
            wlan1', verbose=verbose)
    elif cmd == "SETUP_REQUEST_OPEN_MALFORMED":
        setup_response_packet = None
        connected = False
        sendp(create_tdls_setup_request(gdcs=None, malformed
            =True), iface='wlan1', verbose=verbose)
    elif cmd == "SETUP_REQUEST_AES_MALFORMED":
        setup_response_packet = None
        connected = False
```

```
            sendp(create_tdls_setup_request(gdcs=4, malformed=
                True), iface='wlan1', verbose=verbose)
        elif cmd == "TEARDOWN":
            setup_response_packet = None
            connected = False
            sendp(create_tdls_teardown(), iface='wlan1', verbose
                =verbose)
        elif cmd == "RESET":
            setup_response_packet = None
            connected = False
            tpk = None
            client.sendall("{}\n".format("NO_RESPONSE"))
            continue
        elif cmd == "CONNECTED":
            timeout_response = "CONNECTED" if connected else "
                NOT_CONNECTED"

        response_msg = timeout_response
        if not cmd == "CONNECTED":
            packets = async_sniff.get()

            if len(packets) > 1:
                packets = packets[1:]
                print('response packet length longer 0')
                if packets[0].action == 2:
                    response_msg = "SETUP_CONFIRM"
                elif packets[0].action == 1:
                    setup_response_packet = packets[0]
                    status = setup_response_packet[
                        Dot11TDLSAction].status_code
                    response_msg = "SETUP_RESPONSE_FAIL_{}".
                        format(status)
                    if status == 0:
                        response_msg = "SETUP_REPONSE_SUCCESS"
                elif packets[0].action == 3:
                    response_msg = "TEARDOWN"
                else:
                    response_msg = "action: {}".format(packets
                        [0].action)

        client.sendall("{}\n".format(response_msg))
        print("Sending to learner: `{}`".format(response_msg))


if __name__ =='__main__':main()
```

Listing A.5: start$_wpa.py$

```
#!/usr/bin/env python2.7

from scapy.all import *
import time
import logging
logger = logging.getLogger()
```

```
import subprocess

import sys
sys.path.insert(0, "../hostap/tests/hwsim")
sys.path.insert(0, "../hostap/wpaspy")

from wpasupplicant import WpaSupplicant
import hwsim_utils
from hostapd import HostapdGlobal
from hostapd import Hostapd
import hostapd
from utils import HwsimSkip, skip_with_fips
from wlantest import Wlantest


def init_stas():
    stdout_handler = logging.StreamHandler()
    stdout_handler.setLevel(logging.DEBUG)
    logger.addHandler(stdout_handler)

    dev0 = WpaSupplicant('wlan0', '/tmp/wpas-wlan0')
    dev1 = WpaSupplicant('wlan1', '/tmp/wpas-wlan1')
    dev2 = WpaSupplicant('wlan2', '/tmp/wpas-wlan2')
    dev = [ dev0, dev1, dev2 ]

    for d in dev:
        if not d.ping():
            logger.info(d.ifname + ":␣No␣response␣from␣
                wpa_supplicant")
            return
        logger.info("DEV:␣" + d.ifname + ":␣" + d.p2p_dev_addr()
            )

    params = hostapd.wpa2_params(ssid="test-wpa2-psk",
        passphrase="12345678")
    hapd = hostapd.add_ap({"ifname": 'wlan3', "bssid": "
        02:00:00:00:03:00"}, params)

    Wlantest.setup(hapd)
    wt = Wlantest()
    wt.wlantest_cli = '../hostap/wlantest/wlantest_cli'
    wt.flush()
    wt.add_passphrase("12345678")
    wt.add_wepkey("68656c6c6f")

    dev[0].connect("test-wpa2-psk", psk="12345678", scan_freq="
        2412")
    dev[1].connect("test-wpa2-psk", psk="12345678", scan_freq="
        2412")

def main():
    subprocess.call('service␣NetworkManager␣stop', shell=True)
    subprocess.call('bash␣start.sh', shell=True, cwd="../hostap/
        tests/hwsim/")
    init_stas()

if  __name__ =='__main__':main()
```

Listing A.6: stop.py

```
#!/usr/bin/env python2.7

from scapy.all import *
import time
import logging
logger = logging.getLogger()
import subprocess

import sys
sys.path.insert(0, "../hostap/tests/hwsim")
sys.path.insert(0, "../hostap/wpaspy")

import hwsim_utils
from wpasupplicant import WpaSupplicant
from hostapd import HostapdGlobal
from hostapd import Hostapd
import hostapd
from utils import HwsimSkip, skip_with_fips
from wlantest import Wlantest

def main():
    subprocess.call('bash stop.sh', shell=True, cwd="../hostap/
        tests/hwsim/")
    subprocess.call('service NetworkManager start', shell=True)

if  __name__ =='__main__':main()
```

Listing A.7: util.py

```
import struct
import hashlib
import hmac
import binascii
from radiotap import radiotap_parse

mask = 0xFFFFFFFFFFFFFFFF

def bit256_to_hex(v):
    return struct.pack('<QQQQ', v&mask, (v>>64)&mask, (v>>128)&
        mask, (v>>192)&mask)

def hex_to_bit256(hex):
    value = struct.unpack('<QQQQ', hex)
    return value[0] + (value[1] << 64) + (value[2] << 128) + (
        value[3] << 192)

def bit128_to_hex(v):
    return struct.pack('<QQ', v&mask, (v>>64)&mask)

def hex_to_bit128(hex, big_endian=False):
    value = struct.unpack('<QQ', hex)
    return value[0] + (value[1] << 64)

def KDFSHA256(key, label, context):
    counter = 1
    buffer = ''

    print("KDF␣Label:␣{}".format(label))
    print("KDF␣Context:␣{}".format(binascii.hexlify(context)))

    pos = 0
    while pos < ((256 + 7) / 8):
        print("HMAC␣Input:␣{}".format(binascii.hexlify(struct.
            pack('<H', counter) + label + context + struct.pack('
            <H', 256))))
        tmp = hmac.new(key, struct.pack('<H', counter) + label +
             context + struct.pack('<H', 256), hashlib.sha256)
        buffer = buffer + tmp.digest()
        pos += 32
        counter += 1
    print("KDF␣Counter:␣{}".format(counter))

    return buffer[:32]

def setBit( value , index ):
    """ Set the index'th bit of value to 1.
    """
    mask = 1 << index
    value &= ~mask
    value |= mask
    return value

def getBit( value , index ):
    """ Get the index'th bit of value.
    """
    return (value >> index) & 1

def hasFCS( packet ):
    """ Check if the Frame Check Sequence (FCS) flag is set in
        the Radiotap header.
    """
    assert( packet.haslayer( RadioTap ) ), \
        'The␣packet␣does␣not␣have␣a␣Radiotap␣header.'
    _ , radiotap    = radiotap_parse( str(packet) )
    radiotapFCSFlag = False
    if getBit( radiotap['flags'] , 4 ) == 1:
        radiotapFCSFlag = True
    return radiotapFCSFlag

def assertDot11FCS( packet , expectedFCS = None ):
    """ Validates the Frame Check Sequence (FCS) over a Dot11
```

```
        layer. It is possible to
        pass an expected FCS; this is necessary when there is no
            padding layer available,
        usually in the case of encrypted packets.
    """
    if expectedFCS is None:
        fcsDot11    = str(packet.getlayer( Padding ))
    else:
        fcsDot11    = '{0:0{1}x}'.format( expectedFCS , 8 ) #
            Padding for leading zero.
        fcsDot11    = fcsDot11.decode('hex')
    dataDot11       = str(packet.getlayer(Dot11))[:-4]
    # Calculate the ICV over the Dot11 data, parse it from
        signed to unsigned, and
    # change the endianness.
    fcsDot11Calculated  = struct.pack( '<L' , crc32( dataDot11 )
        % (1<<32) )

    # Assert that we have received a valid FCS by comparing the
        ICV's.
    assert( fcsDot11 == fcsDot11Calculated ), \
        'The received FCS "0x%s" does not match the calculated
            FCS "0x%s".' \
        % ( fcsDot11.encode('hex') , fcsDot11Calculated.encode('
            hex') )
```

Listing A.8: learner.properties

```
type = socket

hostname = localhost
port = 8888
alphabet = SETUP_REQUEST_AES_CORRECT
    SETUP_REQUEST_AES_MALFORMED SETUP_CONFIRM TEARDOWN CONNECTED

output_dir = output

learning_algorithm = lstar

eqtest = randomwords
min_length = 5
max_length = 10
nr_queries = 10
seed = 1
```