James Madison University

Integrated Science & Technology (ISAT)

ISAT 465 - Wireless Networking, Security, & Forensics

TKIP: Temporal Key Integrity Protocol

Author(s):
Troy Gamboa
Isaac Sumner

Submitted to:
Dr. Emil Salib

July 20, 2017

Honor Pledge: I have neither given nor received help on his lab that violates the spirit of the JMU Honor Code.

Troy Gamboa	Isaac Sumner
Signature	*Signature*
Date	*Date*

Contents

List of Figures

List of Tables

1 Introduction

The TKIP algorithm was created to alleviate the weak points that were implemented with WEP encryption. TKIP improved on points of WEP, with the design in mind that TKIP should still be used with hardware that was currently running WEP. As a result, there are some limitations that come with using TKIP, including the fact that TKIP was initially made as a temporary fix to WEP.

TKIP uses the RC4 algorithm. Improving on WEP, TKIP encrypts each data packet with a unique encryption key.

TKIP first implements a key mixing function that combines the secret root key (128 bit temporal key) with the IV (Initialization Vector) and STA MAC before sending it to RC4 initialization. This is an improvement from WEP, as WEP simply concatenated the two. Next, there is a sequence counter to protect against replay attacks. If packets come out of order, the access point will reject the packets. Finally, Michael is used. Michael is the Message Integrity Check, known as MIC.

In this lab report / Semester Project, we will be validating the overall TKIP algorithm. Wireshark has a built in function to decrypt the TKIP algorithm, provided that there is access to the Pre-Shared-Key between the access point and the station. Through use of a program written in C, emulation of the decryption process will be executed. This C program is a driver that makes use of the open-source code provided by `aircrack-ng`, a popular tool commonly used to crack Wi-Fi passwords.

2 Network Diagrams & Tables

2.1 Exercise 1 - Network Diagrams & Tables

Figure 1 shows the network configurations constructed and exercised in Exercise 1.

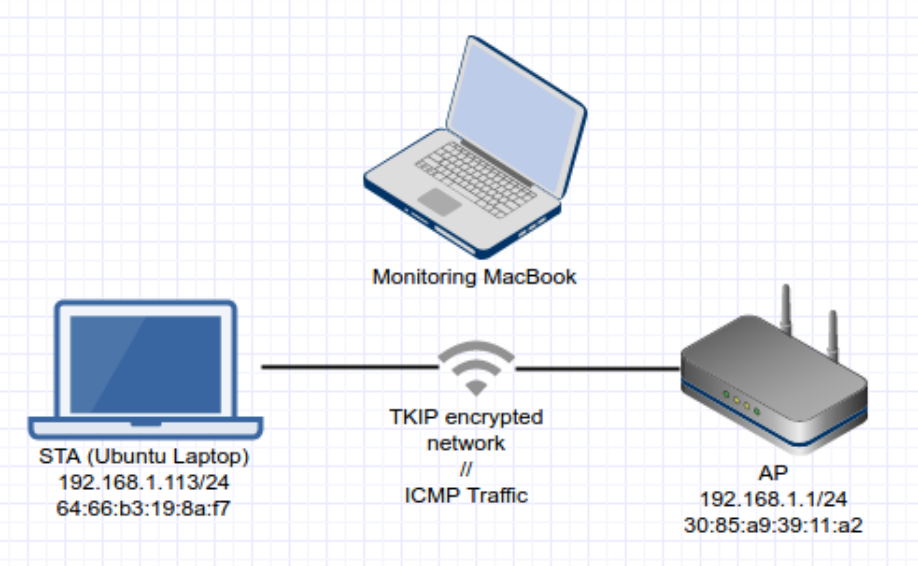

Figure 1: Network Topology For Exercise 1

Table 1 provides the network interfaces information for the hosts, switches shown in Figure 1

Device	Interface	IP Address	MAC Address
Ubuntu Laptop Station	wlan0	192.168.113/24	64:66:b3:19:8a:f7
Access Point	wl0	192.168.1.1/24	30:85:a9:39:11:a2

Table 1: Network Information Table For Exercise 1

3 Lab Exercises : Results & Analysis

3.1 Exercise 1:

3.1.1 Analysis & Evidence

3.1.1.1 Step 0: Network Diagrams and Preparation

See Figure 1 and Table 1 in Section 2

To begin this exercise, we delve into the TKIP algorithm. Moreover, we researched why it was created, and how it works as an algorithm. To do this, we set up a preliminary network topology to receive packets that were related to the TKIP protocol.

First, we configured a wireless interface on a DD-WRT flashed Asus Access point as seen in figure 2 below.

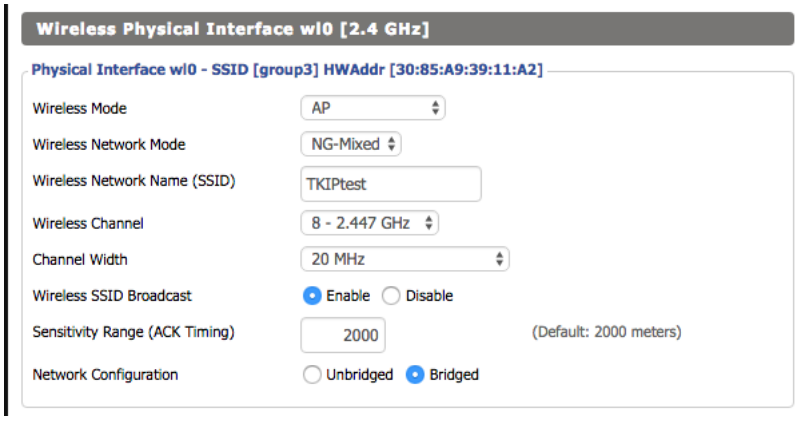

Figure 2: Shows the configuration of the preliminary TKIP network

We also configured the Wireless security settings as seen in figure 3 below.

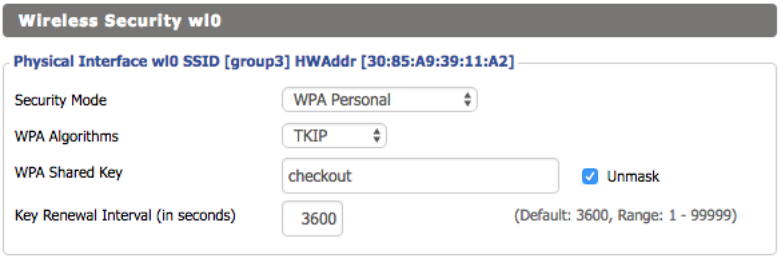

Figure 3: Changing the security settings of the network

3.1.1.2 Step 1:

An example of a TKIP data packet is shown in the figure 4 below.

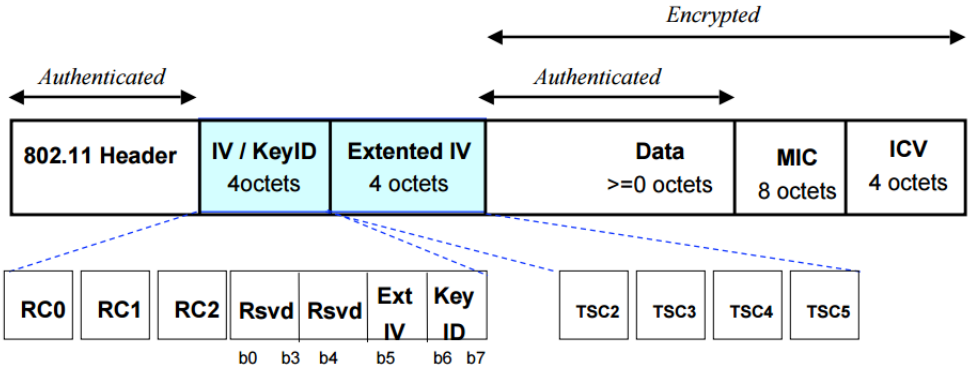

Figure 4: Example of the TKIP packet header

See figure 5 for an example of the flow that TKIP follows.

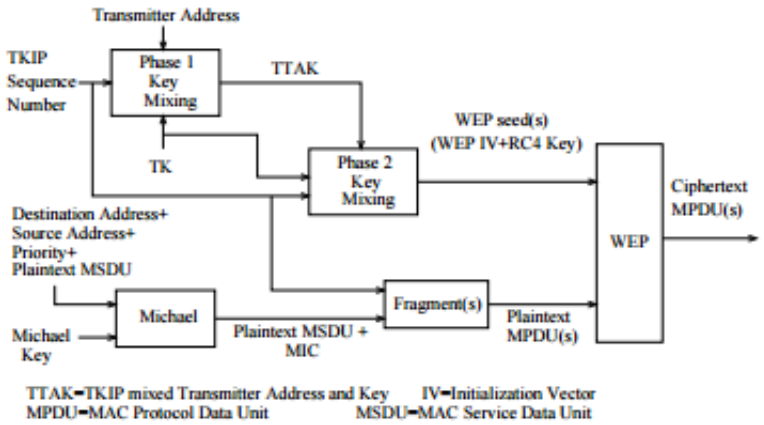

Fig. 1. *TKIP Encryption Diagram*

Figure 5: Diagram of the TKIP encryption algorithm and all of its steps.

After The TKIP algorithm was investigated, we began to capture packets that would be used as the "Target". We began this by training a Monitoring MacBook to the newly created "TKIPtest" network. Once the MacBook was trained, we then connected an Ubuntu Laptop STA to the AP, and pinged the AP 3 times to generate traffic. After the pings were sent through, we then stopped the wireshark capture, and exported the packets into a file called

"tkip_test3.pcapng" attached. In this file, we then took the packets and viewed the TKIP encrypted data as seen in the figure 6 below.

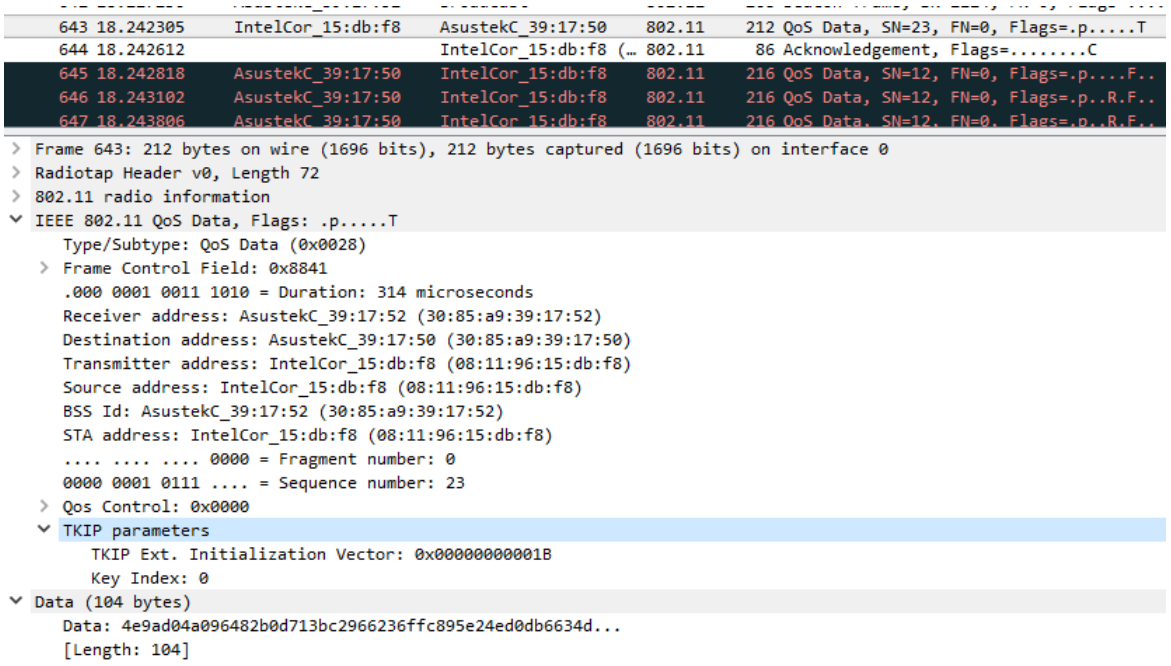

Figure 6: Example of a encrypted packet in wireshark, with the highlighted TKIP parameters

As seen in the figure 6, the TKIP parameters showing the IV for the packet and the key index are shown. Then using wireshark, we decrypted the packets and navigated to the same packet. This is shown in the figures 7 and 8.

Q1: How did TKIP change the way the IV works from WEP?

A1: In WEP, the IV was only 24 bits. In addtion to this, WEP also simply concatenates the IV onto the secret key to create the RC4 Encryption key. In TKIP, The IV length is extended to 48 Bits, fed into 2 phase key mixing algorithms, and outputs the RC4 Encryption key. Due to this, the value of the RC4 key is different for each IV value, and the structure of the RC4 key separated into a "old IV" field and a 104 secret key field. Also, the IV is used as a sequence counter in TKIP.

Q2: What "ingredients" are included in the Phase 1 and Phase 2 key mixing stages?

A2: In phase 1, The secret session key, the high order 32 bits of the IV and the MAC address are included. In Phase 2, The outputs of Phase 1 are then mixed with the lower 16 bits of the IV (The only part that changes each session).

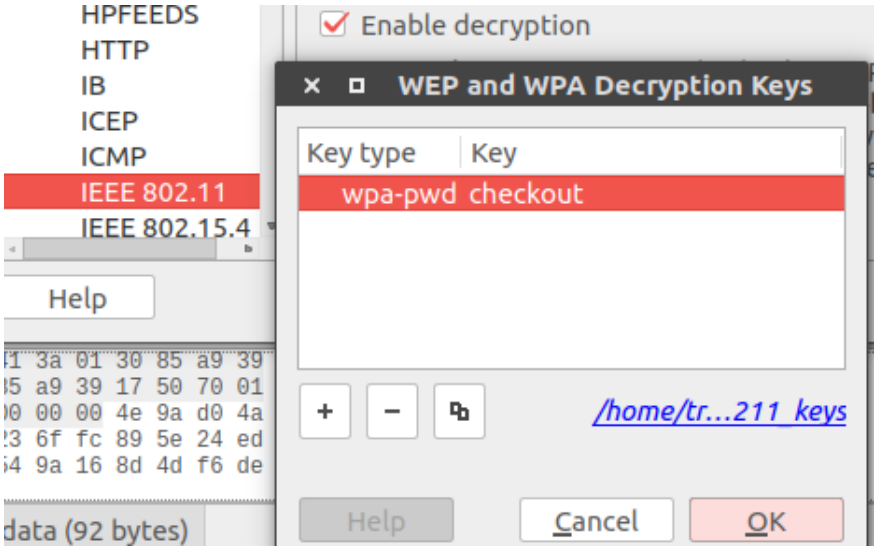

Figure 7: Going to Edit / Preferences, and entering the Key to decrypt the packet shown in 6.

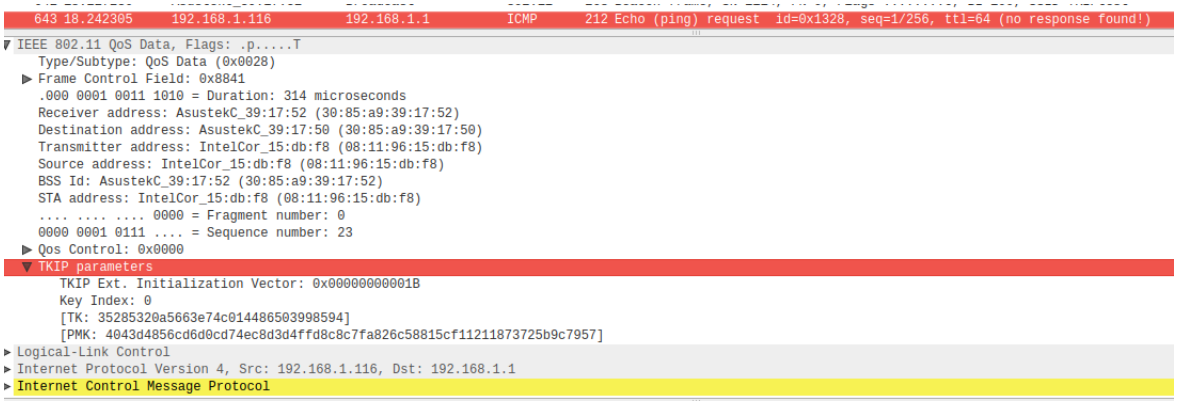

Figure 8: Result of decrypting the TKIP packet, with the relevant parameters highlighted

As seen in the figures‘ above, the decrypted packet contained ICMP packets, relating to our ping, as well as the various TKIP parameters that are revealed when the packet is decrypted. These parameters include the Temporal Key (TK) and the Pairwise Master Key (PMK).

Now that we have a Target for our data flow, we will now attempt to replicate the process that TKIP uses (as previously seen in wireshark) through use of a program written in C.

3.1.1.3 Step 2: Emulating Wireshark Decrypt Process with C

First, we downloaded the relevant wireshark file in the attachments, namely, `tkip_test3.pcap`.

Next, we designated a new directory for the necessary C files. We called this directory `ISAT465_SP` and cloned the `https://github.com/sumnerib/ISAT465_SP` repository as per the instructions.

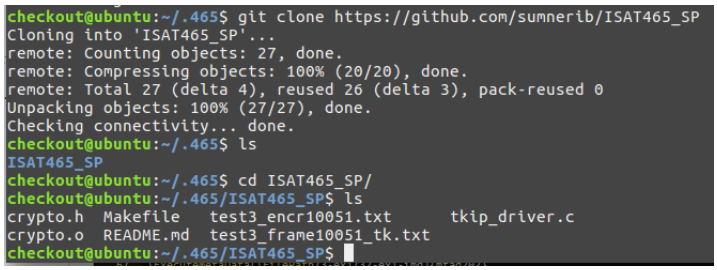

Figure 9: Cloning the Repository with the relevant C Scripts.

Q3: What is the crypto.o file?

A3: A compiled binary file that contains the cryptographic functions used by Aircrack.

Then, we used the `make` command to compile from the `tkip_driver.c` source file and create an executable that we can use to decrypt our frame.

We were then able to run the program with the command `./tkip_driver test3_frame10051_tk.txt test3_encr10051.txt 106 56`. "test3_frame10051_tk.txt" is the file containing the Temporal Key taken from Wireshark and "test3_encr10051.txt" is the frame with the encrypted data we are trying to decrypt. 106 is the length of the frame and 56 is the length of the target. These two txt files are the hex stream copied from Wireshark.

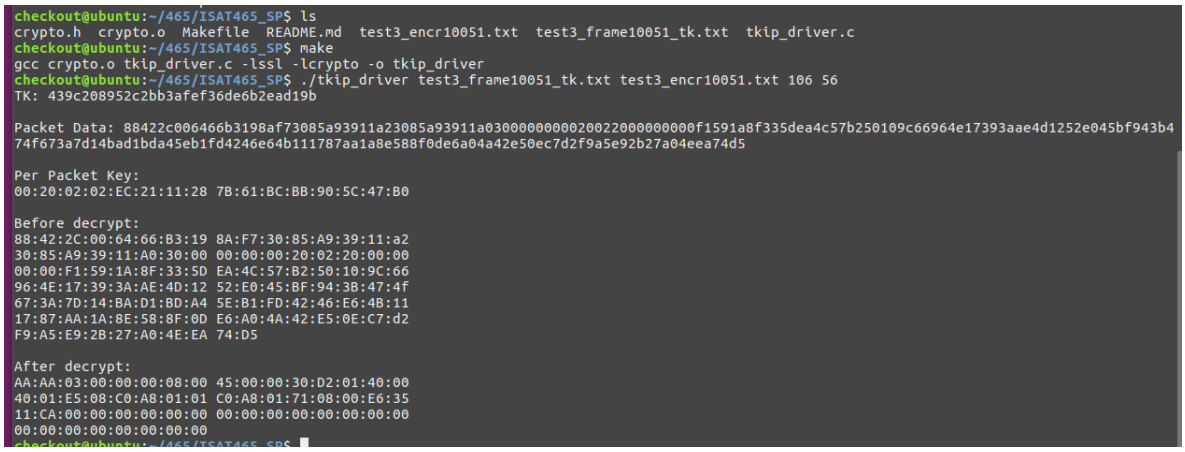

Figure 10

Q4: What are the outputs of the `tkip_driver c` program? Provide evidence. Are these the same as the results provided by Wireshark?

A4: See figures 11 and 12. As seen in the screenshot, these are the same values returned in Wireshark.

Using the `tkip_test3.pcap` file that we had captured earlier in the exercise, we then compared it to the output of the tkip driver c program. See figure 11 and 12. Here, we are comparing the IEEE 802.11 Data layer. Confirming, that the correct hex stream is selected. Also in figure 12, you are able to see the decrypted TKIP parameters that wireshark provides. These parameters seen include the TK and the PMK. The TK is required for the decryption used in the tkip driver C program.

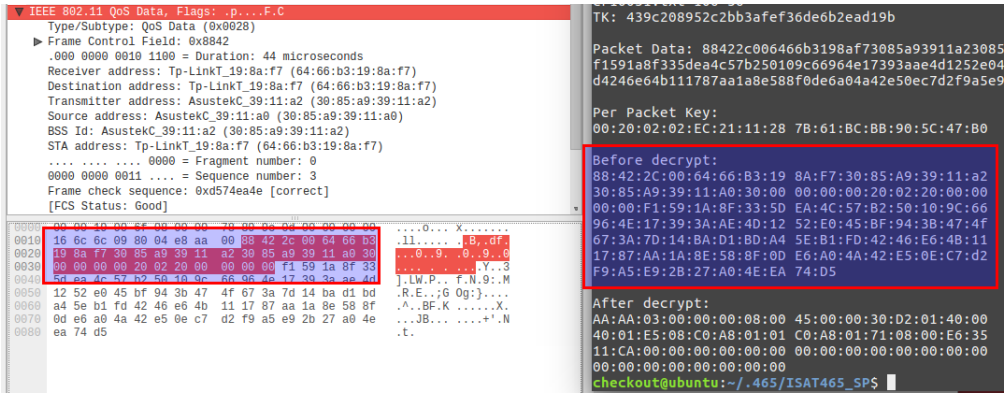

Figure 11: Verifying that the data before encryption is consistent with wireshark.

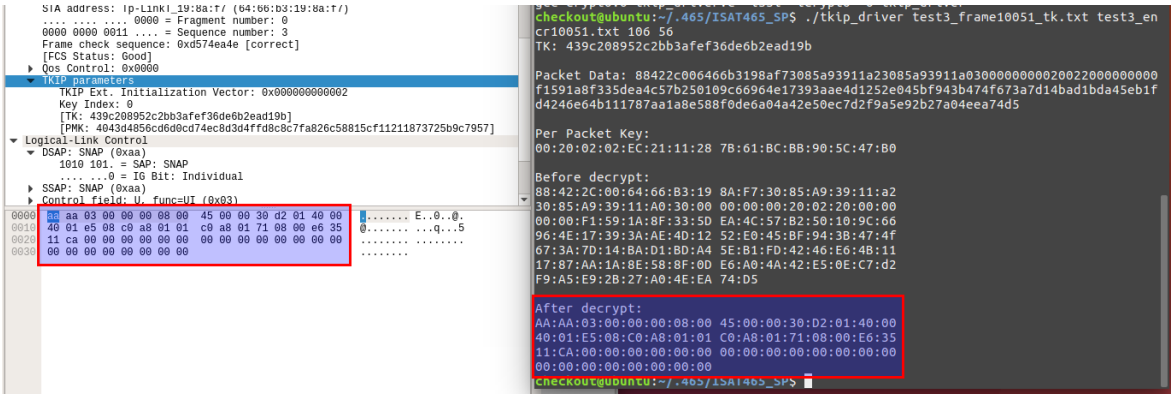

Figure 12: Confirming that the decryption in the C program is consistent with that of wireshark.

Let's take a look at some of the source code in the `tkip_driver.c` file. This can be found in the appendix in section 8.

In Figure: **??** we see the entry point of our program. It first checks and makes sure that the proper number of arguments are provided, and if not it prints out a help dialogue. Then it reads in the length of the provided data. It then reads in the TK file and the frame file. It prints the data out that it read in as is, and then it converts the data from hexadecimal to raw bytes (stored as unsigned characters).

```
98   // Decrypting
99   decrypt_tkip(packet_ascii, len, tk_ascii);
100  printf("After decrypt:\n");
101  dump_formatted_hex(packet_ascii+34, targ_len);
102
103  // clean up
104  free(packet_ascii);
105  free(tk_ascii);
106  free(packet);
107
108  return 0;
```

Figure 13: Calling the decrypt_tkip function

Next, in Figure: 13 we see the actual calling the TKIP decryption function implemented by Aircrack. It takes the frame, its length, and the temporal key as arguments. It modifies the given array with the decrypted data. We then print this out on line 101.

```
1189  int decrypt_tkip( unsigned char *h80211, int caplen, unsigned char TK1[16] )
1190  {
1191      unsigned char K[16];
1192      int z;
1193
1194      z = ( ( h80211[1] & 3 ) != 3 ) ? 24 : 30;
1195      if ( GET_SUBTYPE(h80211[0]) == IEEE80211_FC0_SUBTYPE_QOS ) {
1196          z += 2;
1197      }
1198
1199      calc_tkip_ppk( h80211, caplen, TK1, K );
1200
1201      return( decrypt_wep( h80211 + z + 8, caplen - z - 8, K, 16 ) );
1202  }
```

Figure 14: The decrypt_tkip function

Figure: 14 shows us the decrypt_tkip function implemented by Aircrack. First the function determines how much it needs to offset based on the 802.11 header. Then, it calculates the Per-packet key using the TK. Next, it uses the PPK to decrypt with the WEP decryption function.

```
1126  int calc_tkip_ppk( unsigned char *h80211, int caplen, unsigned char TK1[16], unsigned char key[16] )
1127  {
1128      int i, z;
1129      uint32_t IV32;
1130      uint16_t IV16;
1131      uint16_t PPK[6];
1132
1133      if(caplen) {}
1134
1135      z = ( ( h80211[1] & 3 ) != 3 ) ? 24 : 30;
1136      if ( GET_SUBTYPE(h80211[0]) == IEEE80211_FC0_SUBTYPE_QOS ) {
1137          z += 2;
1138      }
1139      IV16 = MK16( h80211[z], h80211[z + 2] );
1140
1141      IV32 = ( h80211[z + 4]       ) | ( h80211[z + 5] <<  8 ) |
1142             ( h80211[z + 6] << 16 ) | ( h80211[z + 7] << 24 );
1143
1144      PPK[0] = LO16( IV32 );
1145      PPK[1] = HI16( IV32 );
1146      PPK[2] = MK16( h80211[11], h80211[10] );
1147      PPK[3] = MK16( h80211[13], h80211[12] );
1148      PPK[4] = MK16( h80211[15], h80211[14] );
1149
1150      for( i = 0; i < 8; i++ )
1151      {
1152          PPK[0] += _S_( PPK[4] ^ TK16( (i & 1) + 0 ) );
1153          PPK[1] += _S_( PPK[0] ^ TK16( (i & 1) + 2 ) );
1154          PPK[2] += _S_( PPK[1] ^ TK16( (i & 1) + 4 ) );
1155          PPK[3] += _S_( PPK[2] ^ TK16( (i & 1) + 6 ) );
1156          PPK[4] += _S_( PPK[3] ^ TK16( (i & 1) + 0 ) ) + i;
1157      }
```

Figure 15: Phase 1 of the TKIP Key-mixing Process

Next, in Figure: 15 we can see Phase 1 of generating the PPK. This phase only needs to be done once per session, but for the purposes of Aircrack, both Phase 1 and Phase 2 are computed every time a PPK is calculated. In normal circumstances only Phase 2 is computed with each transmission, in order to make the most efficient use of limited clock cycles.

```
1159        PPK[5] = PPK[4] + IV16;
1160
1161        PPK[0] += _S_( PPK[5] ^ TK16(0) );
1162        PPK[1] += _S_( PPK[0] ^ TK16(1) );
1163        PPK[2] += _S_( PPK[1] ^ TK16(2) );
1164        PPK[3] += _S_( PPK[2] ^ TK16(3) );
1165        PPK[4] += _S_( PPK[3] ^ TK16(4) );
1166        PPK[5] += _S_( PPK[4] ^ TK16(5) );
1167
1168        PPK[0] += ROTR1( PPK[5] ^ TK16(6) );
1169        PPK[1] += ROTR1( PPK[0] ^ TK16(7) );
1170        PPK[2] += ROTR1( PPK[1] );
1171        PPK[3] += ROTR1( PPK[2] );
1172        PPK[4] += ROTR1( PPK[3] );
1173        PPK[5] += ROTR1( PPK[4] );
1174
1175        key[0] =   HI8( IV16 );
1176        key[1] = ( HI8( IV16 ) | 0x20 ) & 0x7F;
1177        key[2] =   LO8( IV16 );
1178        key[3] =   LO8( (PPK[5] ^ TK16(0) ) >> 1);
1179
1180        for( i = 0; i < 6; i++ )
1181        {
1182            key[4 + ( 2 * i)] = LO8( PPK[i] );
1183            key[5 + ( 2 * i)] = HI8( PPK[i] );
1184        }
1185
1186        return 0;
1187    }
```

Figure 16: Phase 2 of the TKIP Key-mixing Process

Finally, in Figure: 16, we see Phase 2 of the mixing process. This is performed for every transmission and can be precomputed, since the only thing that changes between transmissions is the IV, which is also a sequence counter. Once "key" is populated with the proper data, this function returns and the PPK can be used as the key in the WEP algorithm.

3.1.2 Key Learning/Takeaways:

In this lab / excerise, we learned that aircrack had provided relevant functions used to decrypt the TKIP algorithm. By altering some of the open-source code that aircrack had provided on their github, we were able to input certain hex streams of data, namely the Temporal Key, and the IEEE 802.11 data layer. Learning how to compile and write C-based programs was a large takeaway. Lastly, the biggest takeaway was learning how the packets encrypted by TKIP were structured, as well as how these packets are delivered in the overall algorithm structure. We learned that this algorithm was very complicated, especially when aspects such as the key mixing and computation of the MIC.

4 Lab List of Attachments & Additional Questions

4.1 List of Attachments

The following is a list of the attachments uploaded to the 01-atts directory:

1. tkip_test3.pcap

4.2 Additional Questions

N/A

5 Lab Observations, Suggestions & Best Practices

5.1 Observations

It was slightly difficult to find the correct source code to adhere to the scope of the project. We had initially decided on the version that aircrack had provided, switched to a python implementation, and then reverted back to the aircrack code.

It took awhile to become comfortable with understanding the TKIP algorithm, as it is a very dense and intensive protocol that is composed of many specific steps in order to provide the 'fix' to WEP. This being said, TKIP is still a viable option for wireless security, despite the effectiveness of the more common CCMP/AES option.

5.2 Best Practices

1. Learning how to read / compile C code
2. Investigating how the TKIP Algorithm operates
3. Configuring a TKIP network and capturing the correct usable packets for testing

6 Lab References

7 Acknowledgments

We would like to acknowledge Dr. Emil Salib for assisting us by providing us resources and references to use to validate the TKIP algorithm.

8 Appendices

```
#include <string.h>
#include <stdio.h>
#include "crypto.h"

/**
 * ## Developed for the JMU ISAT465 Semester Project ##
 * Authors: Isaac Sumner & Troy Gamboa
 * Date: 4/25/17
 */

/*
 * Takes chars representing data in hex
 * and converts it to bytes.
 */
unsigned char *hex2string(char *hex, int len)
{
    unsigned char *string = (char*)malloc(sizeof(char) * (len/2));
    char *substring = (char*)malloc(sizeof(char) * 2);
    int index = 0;

    int i;
    for (i = 0; i < len; i+=2) {

        strncpy(substring, hex+i, 2);
        int x = strtol(substring, NULL, 16);
        string[index] = (char)x;
        index++;
    }

    free(substring);
    return string;
}

void dump_formatted_hex(unsigned char *data, int len)
{
    int i;
    for (i = 1; i < len; i++) {

        if ((i % 16) == 0)
            printf("%02x\n", data[i-1]);
        else if ((i % 8) == 0)
            printf("%02X ", data[i-1]);
        else
            printf("%02X:", data[i-1]);
```

```
    }
    printf("%02X\n", data[len-1]);
}

int main(int argc, char *argv[])
{
    if (argc != 5) {
        printf("Usage: tkip_driver <tk_file> <packet_file>");
        printf(" <length> <target length>\n");
        return 0;
    }

    int len = atoi(argv[3]);
    int targ_len = atoi(argv[4]);
    FILE *fp1 = fopen(argv[1], "r");
    char tk_hex[33];
    fgets(tk_hex, 33, (FILE*)fp1);
    fclose(fp1);

    printf("TK: %s\n\n", tk_hex);

    int str_len = (len * 2) + 1;
    FILE *fp2 = fopen(argv[2], "r");
    char *packet =
        (char*)malloc(sizeof(char) * str_len);
    fgets(packet, str_len, (FILE*)fp2);
    fclose(fp2);

    printf("Packet Data: %s\n\n", packet);

    unsigned char *tk_ascii = hex2string(tk_hex, 33);
    unsigned char *packet_ascii = hex2string(packet, str_len);
    unsigned char rc4key[16];

    // Calculate the ppk
    calc_tkip_ppk(packet_ascii, len, tk_ascii, rc4key);

    // Print the ppk
    printf("Per Packet Key:\n");
    dump_formatted_hex(rc4key, 16);
    printf("\n");

    printf("Before decrypt:\n");
    dump_formatted_hex(packet_ascii, len);
    printf("\n");
```

```
    //encrypt_wep(packet_ascii+z+8, len-z-8, rc4key, 16);

    //printf("After encrypt: \n");
    //for(i = 0; i < len; i++)
       // printf("%02X:", packet_ascii[i]);
    //printf("\n");

    // Decrypting
    decrypt_tkip(packet_ascii, len, tk_ascii);
    printf("After decrypt:\n");
    dump_formatted_hex(packet_ascii+34, targ_len);

    // clean up
    free(packet_ascii);
    free(tk_ascii);
    free(packet);

    return 0;
}
```