ENSC 427: COMMUNICATION NETWORKS
SPRING 2019

KRACK - The Destruction of WiFi
A simulation of KRACK attack

http://www.sfu.ca/~dgl3/ENSC427_Team05.html

TEAM 5

Derrick Lee	Eric Kwok	Duncan Yee
Dgl3@sfu.ca	Eka34@sfu.ca	Duncan_yee@sfu.ca

Contents

1. Abstract

Modern technology and appliances connect themselves effortlessly to the wireless network. Due to the simplicity of the task, how does the average user maintain the security of the connection? With the given intuitive nature of the setup, people generally assume that frequent changing of passphrases and the use of a firewall provide ample security to the connection. The simplicity ends here as the connection to the wireless network is far more complex.

How do devices secure a link to the wireless network? The key to obtaining a connection to the wireless network has to do with the 4-way handshake. A router contains a passphrase that provides a link between devices and the wireless network. Users are required to input the passphrase into the specified device which will then have to prove to the router that it possesses this passphrase by initiating and completing the 4-way handshake; this handshake is where a vulnerability of the connection lies, exploited by the Key Reinstallation AttaCK (KRACK) which provides attackers access to the user's traffic.

2. Glossary

- AES - Advanced Encryption Standard
- AP - Access Point
- CBC - Cipher Blockchaining
- CCMP - Counter Mode Cipher Block Chaining Message Authentication Code Protocol
- GTK - Group Temporal Key
- HMAC - Hash Message Authentication Code
- KRACK - Key Reinstallation Attack
- MAC address - Media Access Control address
- MIC - Message Integrity Code
- PMK - Pairwise Master Key
- PSK - Pre-Shared Key
- PTK - Pairwise Transient Key
- SHA1 - Secure Hash Algorithm ver. 1
- SSID - Service Set Identifier
- TK - Temporal/Transient Key
- EAP - Extensible Authentication Protocol
- OLSR - Optimized Link State Routing Protocol

3. Introduction

The project will delve into the 4-way handshake that is used to obtain a wireless connection. Steps will be broken down into individual components prior to presenting the KRACK attack which will expose its vulnerabilities. We further demonstrate and learn about the exposure by simulating the attack with NS-3.

Possible prevention methods hypothesized from the data will be discussed. The current prevention given by service providers are in the form of a series of patches. Open-source patches will be analyzed and attempt to be implemented into the KRACK simulation. The analysis of the open-source patches in addition to the hypothesized prevention methods will hope to further impede the success of an attack.

4. Literature review

“A Comprehensive Attack Flow Model and Security Analysis for Wi-Fi and WPA3”:

An overview of existing wireless security protocols, WEP, WPA, WPA2, and the new WPA3 are presented. The WPA2 protocol is examined in detail, providing us with an idea of the measures in place to enable secure wireless transmission. This also allows us to understand the exploits that are possible within the WPA2 protocol itself. A variety of attacks against wireless networks are also presented, and in particular, we focus on the section regarding KRACK attack (3.2.6). A high level description of the attack is explained including specific steps of the protocol. Encryption and decryption details are also included, illustrating how the attack bypasses measures intended for security. However, specific implementation details of the attack are not shown. [1]

“Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2”:

Written by the researchers that discovered the attack, this paper details the vulnerabilities of the WPA2 protocol which enable the exploit. The paper examines a variety of implementations of WPA2 across multiple operating systems including Linux, Android, Windows, etc. A range of behaviours is discussed. In some cases the attack is unsuccessful. In other cases decryption of network traffic is achieved. In the worst cases, such as those operating systems making use of wpa_supplicant version 2.4 and 2.5, the client will reinstall an all-zero key. This is particularly devastating as this makes decryption trivial, and this issue affects many Android and Linux systems. [2]

“Hostapd and WPA_supplicant Security Advisories”

HostAP was one of the 802.11 device drivers used heavily in Linux to perform the functions of a wireless access point. The code was developed by Jouni Malinen which was further improved in January 2017 due to the vulnerability of the 4-way handshake exploited by this attack. The patch update tracks several variables and prevents the degradation of the TKs. [1]

4.1 Related Work

Related work on this topic is rather low because this topic is relatively new. But researchers have created a YouTube video called "KRACK Attacks: Bypassing WPA2 against Android and Linux" [4] showcasing and demonstrating the viability and the strength of the KRACK attack itself. The video was done by one of the researchers who found the flaw within the WiFi protocol, Mathy Vanhoef. This video shows an android device connecting to the WiFi network but the attacker employs the KRACK attack and is able to fully decrypt the packets being passed through the network and is able to obtain the username and password for the site uk.match.com.

5. Problem Description

Nowadays, free public wifi is available everywhere you go. Restaurants, coffee shops, shopping centres, airports, etc. A majority of these routers and access points rely on the WPA2 protocol to secure their networks.

Previously, WPA2 was thought to be secure, as the underlying algorithms had been proven to be mathematically secure. Attacks against AES-CCMP had not been discovered beyond brute force attacks, which were not feasible due to its minimum key length of 128 bits, along with other security features of its counter mode, and cipher-blockchaining. [2]

Although the protocol itself is secure, when developers implement the WPA2 4-way handshake, certain details were not made clear, creating flaws that could be exploited by the KRACK attack. In particular, message 3 of the 4-way handshake can be abused in order to decrypt the network traffic.

For KRACK attack to be successful some previous work must be done for the attack to be viable. For example, a system must establish itself as the **Man in The Middle position** (MiTM) before the supplicants attempt to connect to the network. This is to enable the ability to manipulate the packets between the source the access point (AP) itself.

In normal use, the client will install its PTK after the 3rd message of the handshake. However, a malicious attacker can continue to retransmit message 3 to the client, and the client will reinstall the identical key, while also resetting its counters as well as the nonce.

In order to go into more detail about the inner workings of the attack, we first must talk about the set-up and other related topics like MiTM and the 4 way handshake protocol itself.

5.1 Man in The Middle

The MiTM position is a position where someone is unknowingly passing their traffic from themselves to the MiTM position and then the destination as opposed to from themselves to their destination.

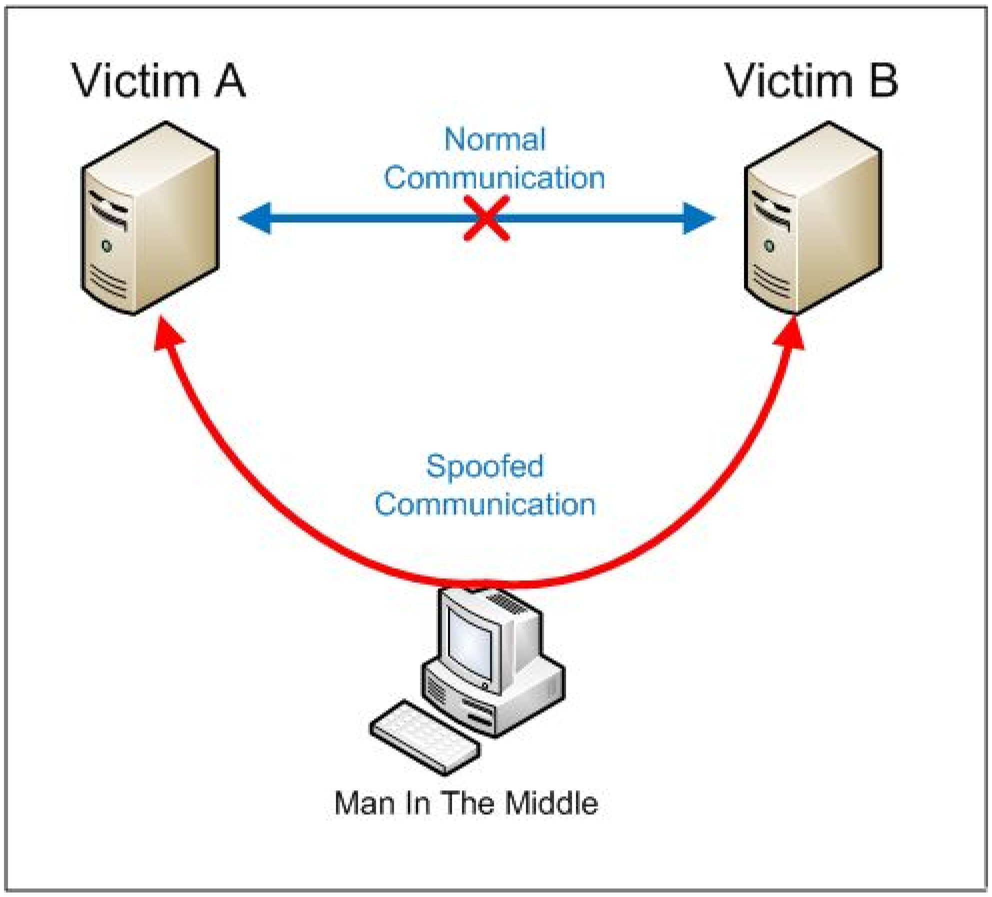


Figure 1 - Architecture of a man in the middle position [4]

As seen in figure 1, once in the MiTM position the user is unknowingly communicating with the attacker instead of the network. This poses a huge security risk to the user being attacked because they can not only passively look at and observe the user's internet activity, but they can also actively forge and manipulate packets to and from your device.

There are many ways to initiate put oneself into the MiTM position: ARP poisoning, cookie injection IP spoofing and wireless MiTM. We will talk about the wireless method of MiTM because it relates to connections of the router. Wireless MiTM can be conducted by creating a dummy AP carrying the same Service Set Identifier (SSID) as the targeted router and a stronger signal. Other methods could be created via hardware modifications like plugging a raspberry pi into the router itself.

<u>5.2 4-Way Handshake</u>

In all modern WiFi devices, it is now standard to have authenticate a connection in order to begin and continue the connection to a router. All devices use mutual authentication based on a shared secret Pairwise Master Key (PMK) which is based on the password to access the router as well as a Pairwise Transient Key (PTK) or session key which is negotiated through the 4 way handshake.

The PMK is generated through the Password Based Key Derivation Function version 2 (PBKDF2). This key is created using the Pre-Shared Key (PSK), Service Set Identifier (SSID), and Hash Message Authentication Code Protocol Secure Hash Algorithm ver. 1 (HMAC-SHA-1) function. The PSK is a key that has been shared between the client and AP, the SSID is the name of the wifi network, and HMAC-SHA-1 is used to verify the message's integrity.

First, the client sends a request to connect. The AP acknowledges, and sends a cryptographic nonce, which is a number used only once. The "ANonce", the nonce from the AP, is a random single use number that allows the client and AP to establish some shared secret information.

Second, when the client receives Anonce, and combines it with its own supplicant nonce, "SNonce", as well as the PMK, and the media access control (MAC) addresses of the client and AP. Message Integrity Code (MIC) key is also derived to ensure that the original message is unaltered in transmission.

Third, after the AP receives Snonce and the MIC from the client, it derives its own PTK and then confirms that the resulting derived MIC matches the received MIC.

Fourth, the client installs the PTK and the secure connection has been created. Because the PTK was generated using the nonces from the client and the AP, the PTK will be different every time a new connection is established. [5]

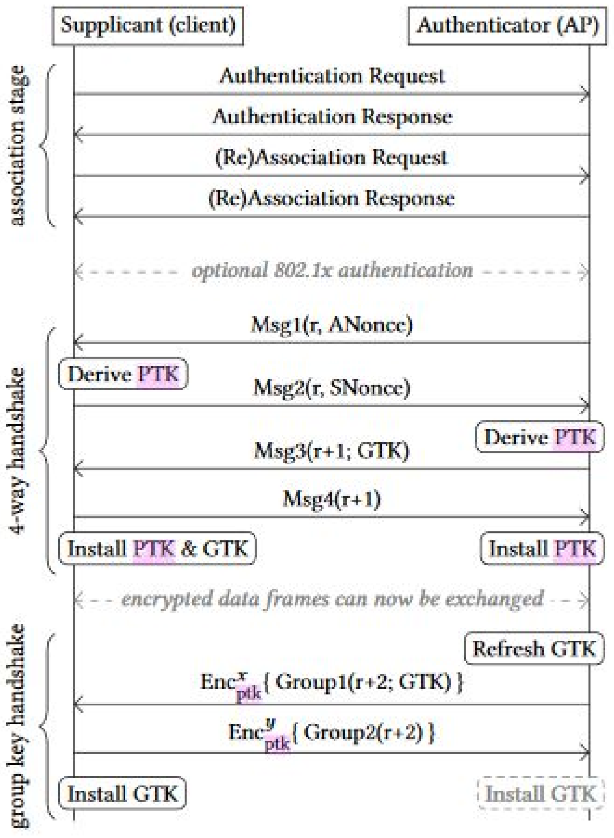


Figure 2 - Overview of the WPA2 4-way handshake [6]

<u>5.3 Encryption Methodology</u>

We will discuss the different methods that is used in encryption and how the KRACK attack is able to beat the encryption algorithms with the simple abuse of the 802.11 protocol.

Counter mode: In addition to using the key to encrypt each packet, a counter is also incremented. This is done so that even if the packet payload is the same, the counter is different so the packets will result in a different encrypted text. KRACK attack exploits counter mode by retransmitting message 3 of the 4-way handshake, resetting the counter. This allows the value to be the same for each captured encrypted packet, making decryption possible. [5]

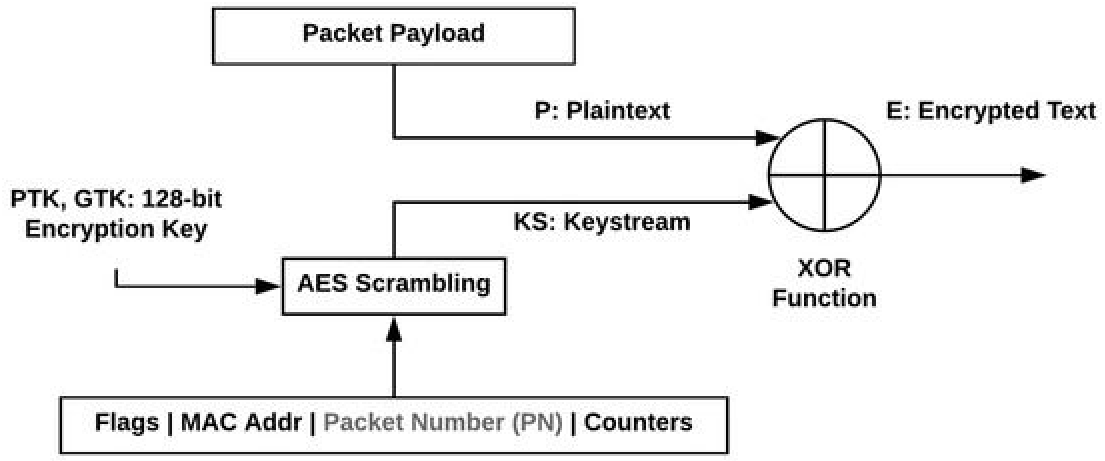


Figure 3 - AES encryption, with counter mode

Cipher block chaining: The ciphertext of the previous message is used to encrypt the next message. Therefore decryption of a specific ciphertext depends on all ciphertexts before it. This means that if an attacker wants to decrypt a specific message, they require every ciphertext before it even if they have the key. [7]

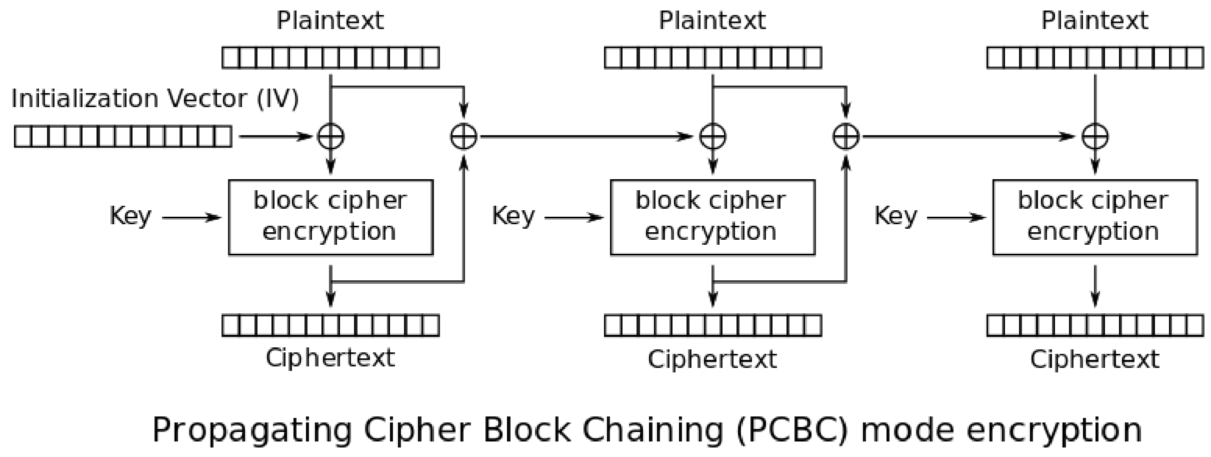


Figure 4 - Cipher Block Chaining encryption of messages [8]

These encryption algorithms above are susceptible to the key reinstallation attacks because during the 4 way handshake protocol within the 802.11 protocol. This is because after receiving message 3 from the AP the supplicant will reinstall the PTK and GTK. The reinstallation of the PTK and GTK will re-initialize the nonce counters to their initial number. This is a serious issue because of how encrypting with the same key can severely weaken encryption. A perfect example found in [9] demonstrates the power of decryption when first one uses the same key to encrypt and second the attacker has the ability to obtain multiple copies of different messages using the same key encryption.
For example:

Person A sends the image figure 5 over WiFi to person B. And this is encrypted by xor-ing figure c1 with a supposedly one time use key figure 6.

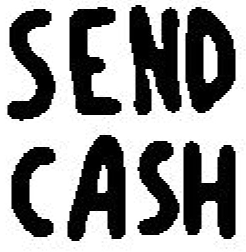


Figure 5 - Plaintext message 1 [10]

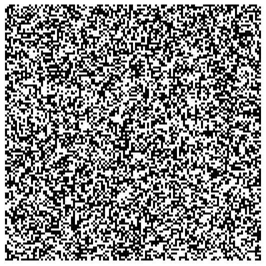


Figure 6 - One time use key 1 [10]

This will create an encrypted image below figure 7.

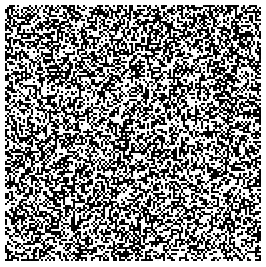


Figure 7 - Encrypted text 1 [10]

If the user then wishes to send another image figure 8 and encrypts it by xor-ing the same key as image the first image. It will generate another securely encrypted image.

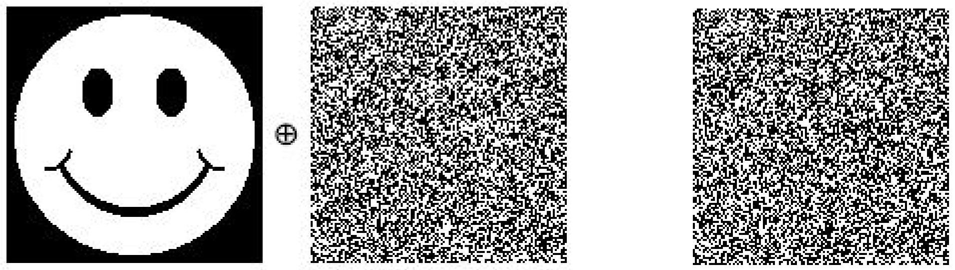


Figure 8 - Encryption of plaintext message 2 using the same key 1 [10]

The real issue is not with the encryption itself but if someone were to get hold of both encryption 1 and encryption 2 (figure 9). If they then xor both the encrypted messages together, it will create figure 9 which gives a pretty clear indication into what the user was trying to do or what kind of information person A was trying to send to person B.

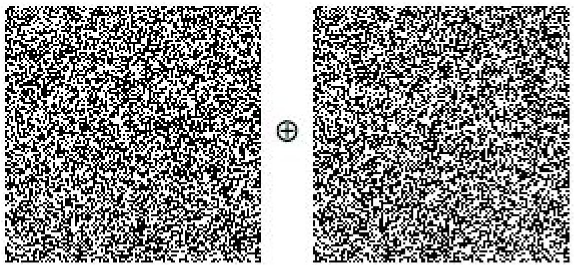


Figure 9 - XOR of encrypted text 1 and 2 [10]

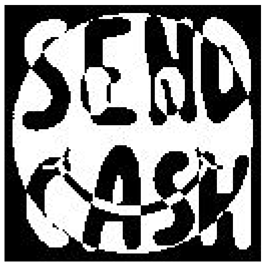


Figure 10 - Result of XOR encrypted texts 1 and 2 [10]

The example above would be similar to what happens in the case of the KRACK attack. The attacker would be able to see the decrypted messages and take sensitive information like the username and passwords.

6. Proposed Simulation and Set Up

Before we began our research, our proposed simulation would have been to create a WiFi network between a stationary node acting as our router, another stationary node acting as our MiTM system and multiple moving nodes acting as a laptop or mobile device. These devices would then go and connect unknowingly to the MiTM set up and then the simulation of the attack would begin. The MiTM node would recognize the Extensible Authentication Protocol (EAP) by placing a sniffer and withhold message 4 from ever reaching the router node. This will initiate the beginning of the KRACK attack.

But after looking into the NS-3 library extensively, the NS-3 program hasn't implemented the EAP protocol. Which left us the option to just simulate a mobile device entering and leaving the WiFi signal while leaving a sniffer at the MiTM which will allow us to see all the traffic passing through the node which would be able to pick up sensitive information being passed to the router.

6.1 Simulation and Results

After the discussion about how to simulate the KRACK attack, it was decided to simulate three nodes which will represent a typical scenario of the KRACK attack in action.

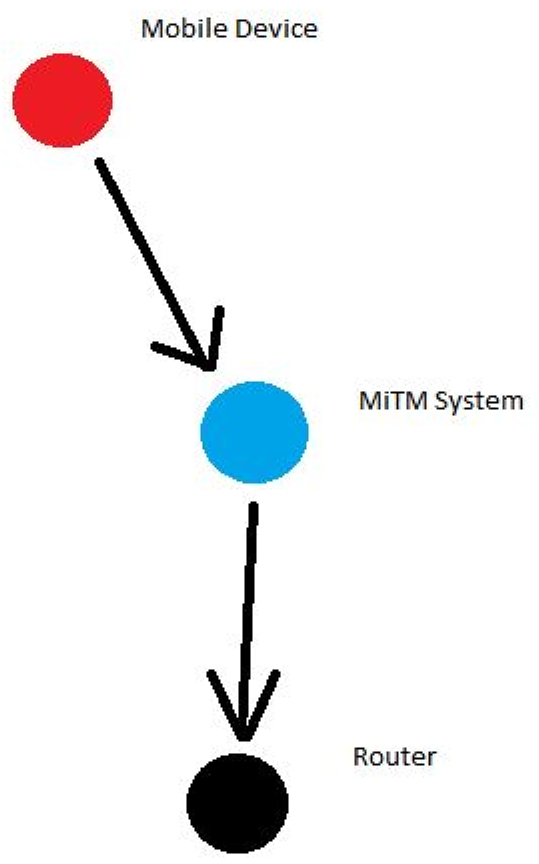


Figure 11 - A mock up of what the simulation layout

A mock image of what our code will simulate is shown in figure 11**.** It shows three nodes: one mobile device, one MiTM system and one router that connects to the internet. The code creates an AP WiFi network and uses ARP protocol

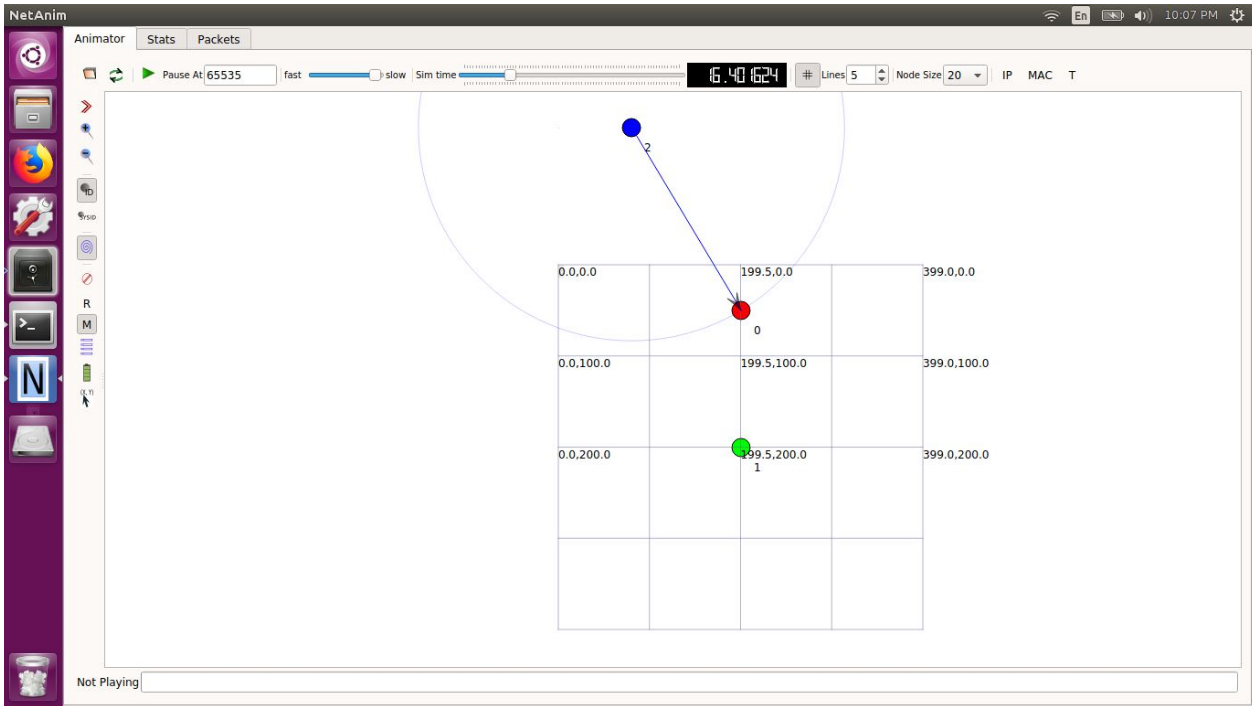
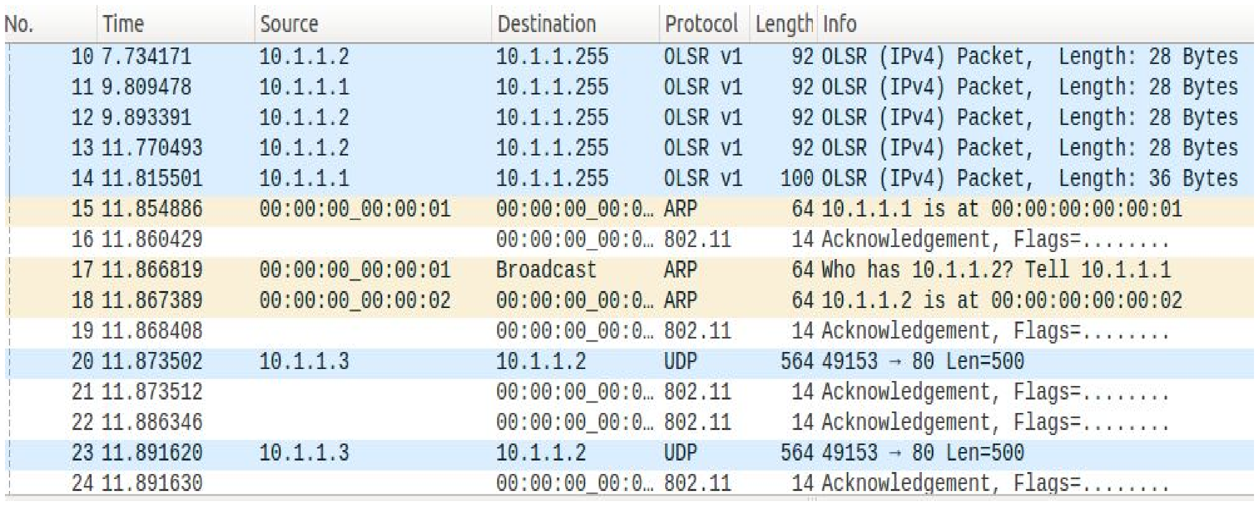


Figure 12 - NetAnim visualization of the code as the mobile node moved into the WiFi radius of the MiTM node

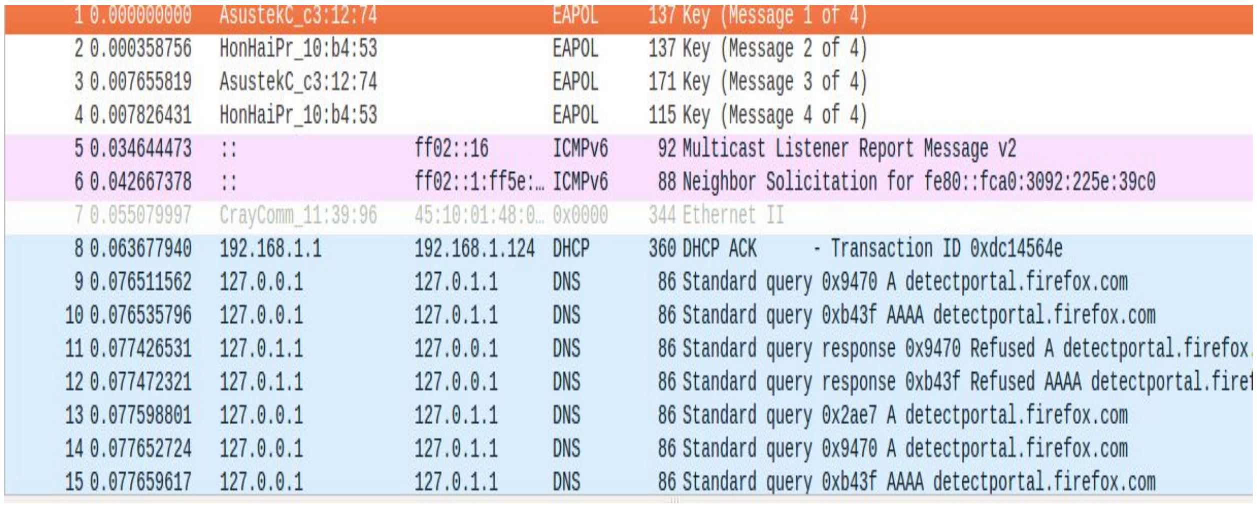
In figure 12, it can be seen that the mobile device (blue dot) has just entered the zone of the WiFi network and has begun to pass messages from the source to the MiTM (red dot) and then the router itself (green dot). The arrows within the figure show the direction the packets are being passed to. Theoretically it would be passing the 4 way handshake message.



No.	Time	Source	Destination	Protocol	Length	Info
10	7.734171	10.1.1.2	10.1.1.255	OLSR v1	92	OLSR (IPv4) Packet, Length: 28 Bytes
11	9.809478	10.1.1.1	10.1.1.255	OLSR v1	92	OLSR (IPv4) Packet, Length: 28 Bytes
12	9.893391	10.1.1.2	10.1.1.255	OLSR v1	92	OLSR (IPv4) Packet, Length: 28 Bytes
13	11.770493	10.1.1.2	10.1.1.255	OLSR v1	92	OLSR (IPv4) Packet, Length: 28 Bytes
14	11.815501	10.1.1.1	10.1.1.255	OLSR v1	100	OLSR (IPv4) Packet, Length: 36 Bytes
15	11.854886	00:00:00_00:00:01	00:00:00_00:0…	ARP	64	10.1.1.1 is at 00:00:00:00:00:01
16	11.860429		00:00:00_00:0…	802.11	14	Acknowledgement, Flags=........
17	11.866819	00:00:00_00:00:01	Broadcast	ARP	64	Who has 10.1.1.2? Tell 10.1.1.1
18	11.867389	00:00:00_00:00:02	00:00:00_00:0…	ARP	64	10.1.1.2 is at 00:00:00:00:00:02
19	11.868408		00:00:00_00:0…	802.11	14	Acknowledgement, Flags=........
20	11.873502	10.1.1.3	10.1.1.2	UDP	564	49153 → 80 Len=500
21	11.873512		00:00:00_00:0…	802.11	14	Acknowledgement, Flags=........
22	11.886346		00:00:00_00:0…	802.11	14	Acknowledgement, Flags=........
23	11.891620	10.1.1.3	10.1.1.2	UDP	564	49153 → 80 Len=500
24	11.891630		00:00:00_00:0…	802.11	14	Acknowledgement, Flags=........

Figure 13 - Looking at the packets that passed through the sniffer node (MiTM)

But from looking at the traffic passing through the MiTM node in figure 13 it can be seen that it is broadcasting itself for other nodes to connect to it. After it generates ACKs after agree to what would have been the 4 way handshake messages being passed between the AP and the source node itself.



```
 1 0.000000000  AsustekC_c3:12:74                  EAPOL   137 Key (Message 1 of 4)
 2 0.000358756  HonHaiPr_10:b4:53                  EAPOL   137 Key (Message 2 of 4)
 3 0.007655819  AsustekC_c3:12:74                  EAPOL   171 Key (Message 3 of 4)
 4 0.007826431  HonHaiPr_10:b4:53                  EAPOL   115 Key (Message 4 of 4)
 5 0.034644473  ::                 ff02::16        ICMPv6   92 Multicast Listener Report Message v2
 6 0.042667378  ::                 ff02::1:ff5e:…  ICMPv6   88 Neighbor Solicitation for fe80::fca0:3092:225e:39c0
 7 0.055079997  CrayComm_11:39:96  45:10:01:48:0…  0x0000  344 Ethernet II
 8 0.063677940  192.168.1.1        192.168.1.124   DHCP    360 DHCP ACK      - Transaction ID 0xdc14564e
 9 0.076511562  127.0.0.1          127.0.1.1       DNS      86 Standard query 0x9470 A detectportal.firefox.com
10 0.076535796  127.0.0.1          127.0.1.1       DNS      86 Standard query 0xb43f AAAA detectportal.firefox.com
11 0.077426531  127.0.1.1          127.0.0.1       DNS      86 Standard query response 0x9470 Refused A detectportal.firefox.
12 0.077472321  127.0.1.1          127.0.0.1       DNS      86 Standard query response 0xb43f Refused AAAA detectportal.firef
13 0.077598801  127.0.0.1          127.0.1.1       DNS      86 Standard query 0x2ae7 A detectportal.firefox.com
14 0.077652724  127.0.0.1          127.0.1.1       DNS      86 Standard query 0x9470 A detectportal.firefox.com
15 0.077659617  127.0.0.1          127.0.1.1       DNS      86 Standard query 0xb43f AAAA detectportal.firefox.com
```

Figure 14 - Packets showing the EAP protocol in a real WiFi network

In figure 14, this was taken from wireshark when my laptop connected to my wireless network at home. As you can see there are four messages at the top of the image in regards to the EAPOL protocol. This would be the 4 way handshake that my mobile device (laptop) would need to complete before being allowed to connect and utilize the network.

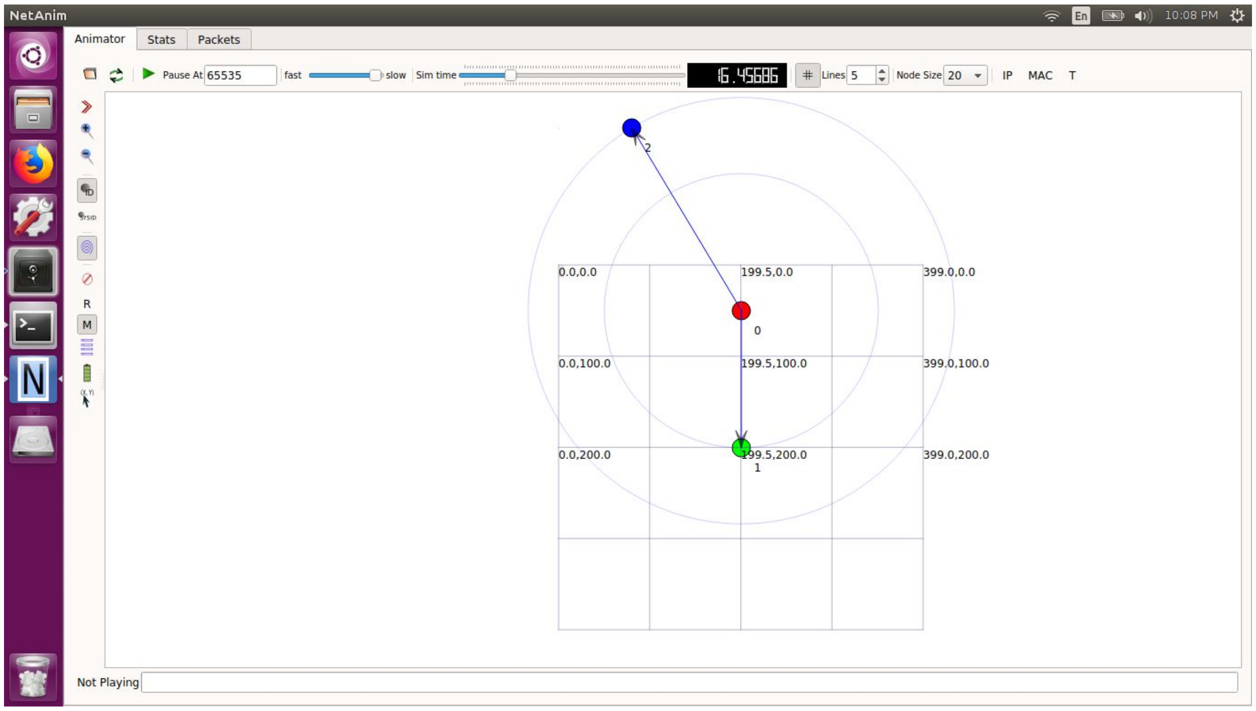


Figure 15 - Image showing the MiTM node forwarding packets to and from the router

In figure 15, we can see that the MiTM position is ACKing the packet from the mobile device as well as forwarding the 4 way handshake message to the router.

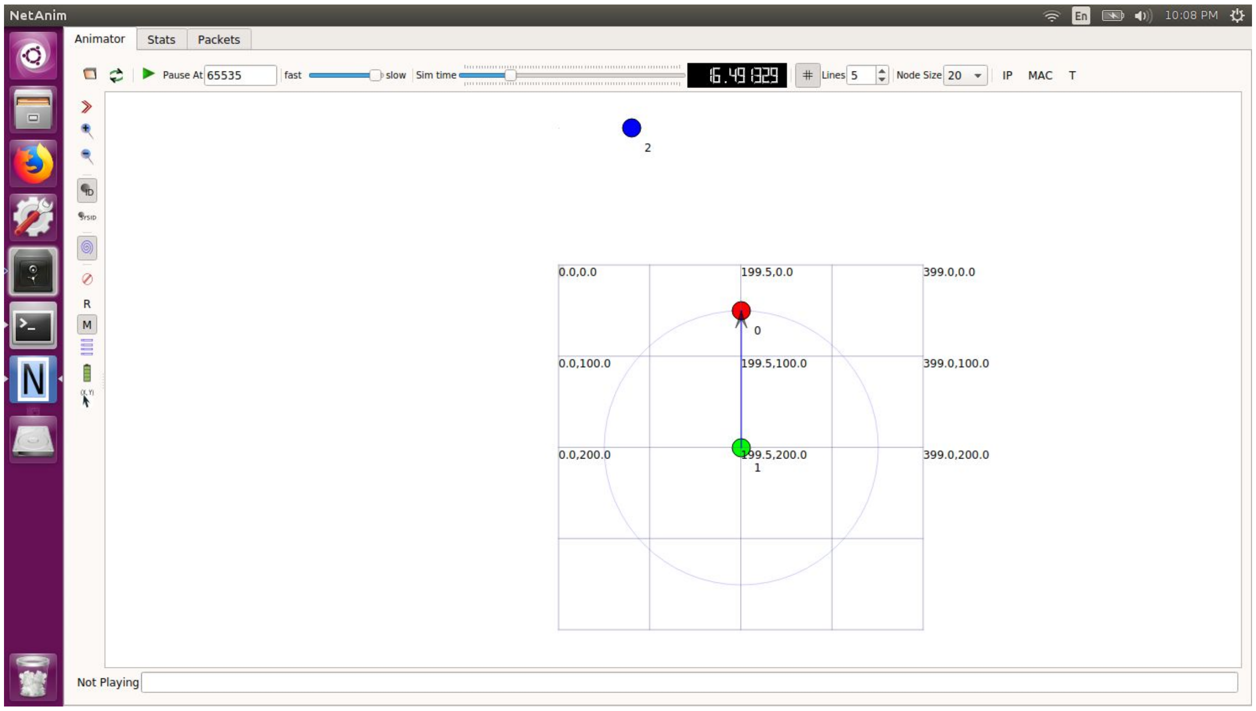
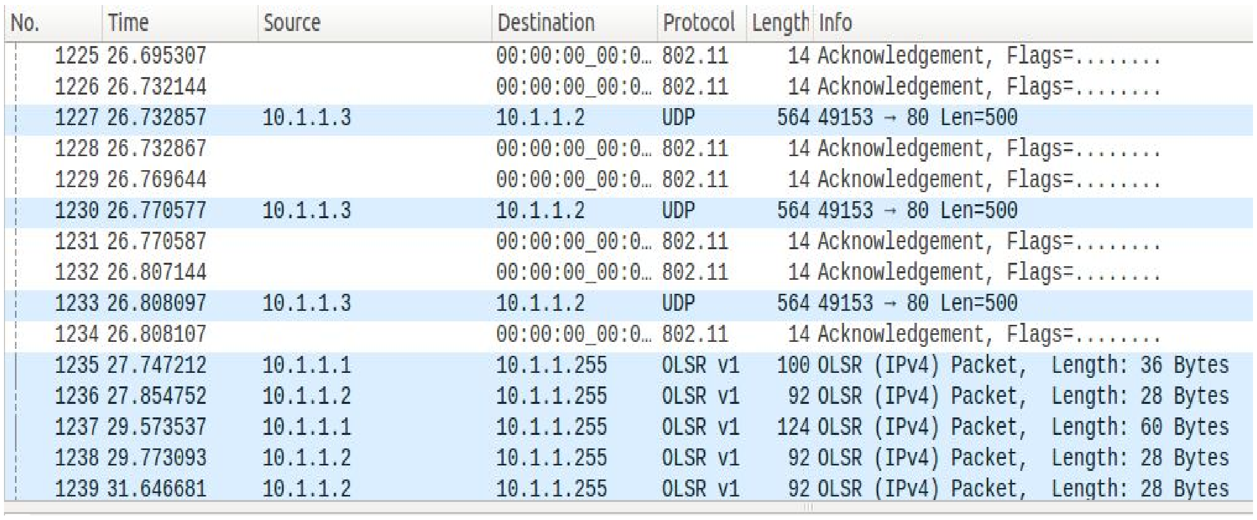


Figure 16 - Image showing the router forwarding packets from the internet to source node

In figure 16, after the router receives the packets sent from the mobile device it returns the information that was queried by the source node and is shown with the arrow.



```
No.   Time       Source     Destination      Protocol Length Info
 1225 26.695307             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1226 26.732144             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1227 26.732857  10.1.1.3   10.1.1.2       UDP        564 49153 → 80 Len=500
 1228 26.732867             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1229 26.769644             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1230 26.770577  10.1.1.3   10.1.1.2       UDP        564 49153 → 80 Len=500
 1231 26.770587             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1232 26.807144             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1233 26.808097  10.1.1.3   10.1.1.2       UDP        564 49153 → 80 Len=500
 1234 26.808107             00:00:00_00:0… 802.11      14 Acknowledgement, Flags=........
 1235 27.747212  10.1.1.1   10.1.1.255     OLSR v1    100 OLSR (IPv4) Packet,  Length: 36 Bytes
 1236 27.854752  10.1.1.2   10.1.1.255     OLSR v1     92 OLSR (IPv4) Packet,  Length: 28 Bytes
 1237 29.573537  10.1.1.1   10.1.1.255     OLSR v1    124 OLSR (IPv4) Packet,  Length: 60 Bytes
 1238 29.773093  10.1.1.2   10.1.1.255     OLSR v1     92 OLSR (IPv4) Packet,  Length: 28 Bytes
 1239 31.646681  10.1.1.2   10.1.1.255     OLSR v1     92 OLSR (IPv4) Packet,  Length: 28 Bytes
```

Figure 17 - Showing what the MiTM node is doing after the mobile device left the WiFi network

By looking at the traffic that is passed through the MiTM node as shown in **figure v7,** it can be seen that after the node has left the WiFi network the MiTM will continue to broadcast its signal with the OLSR protocol to prey on more unsuspecting devices to get more sensitive information.

7. Discussion

7.1 Current Prevention

A prevention that can always be implemented is to use a wired Ethernet connection as opposed to connecting to the wireless network. Since the connection requires the device to be located at a stationary position, this alternative is not always viable. The next best solution is to make sure that the devices are up to date.

As the KRACK attack becomes a widespread problem, service providers are creating updates as patches to their devices. Several solutions that are available to be analyzed are the open-source patch from Linux [11] and the open router firmware design patch from the Linux-based alternative, DD-WRT [12]. Further looking into the design patch provided by DD-WRT, the first modification to the IEEE 802.11c is to track whether the TK has already been configured. Prior configuration will prevent the reinstallation of the key which restarts the sequence number to zero. Another modification is to the wpa authentication process. A new incoming nonce will result in a new TK, diminishing the predictability of the encryption keys. In the analysis of the open-source patch from Linux, it implements all of the components in the previous patch as well as tracking of several other variables that prevent secondary reports of an association event and sleep mode responses. These modifications help to improve the security of the encryption using the network.

7.2 Further Improvements

From the results of the analysis and the simulation, it’s clear that attempts in tracking of the keys as well as several other variables help mitigate the effects of the attack. Currently, the methods implemented by the patch still allow a connection to the rogue access point. A further improvement to the patch could be to implement a detection of the attack which could be placed in the main 802.11 protocol. From the demonstration of the KRACK attack [13], we can see that the rogue access point sends out a stream of queries to targeted devices. Prior to the connection after a requested query, there could be a short timeout placed to see if another query is asked soon after the first one. If a secondary query is detected, it can be seen as a malicious attack and completely deny the connection.

In the future further improvements to the simulation would be to implement the EAP protocol so that it would include the 4 way handshake between the AP and the source node. More mobile devices could also be generated to see if the MiTM node can handle the amount of traffic flowing and be able to maintain the all the different packets from different sources and be able to decrypt the traffic while under such loads.

8. References

[1] L. Epia Realpe, O. Parra and J. Velandia, "Use of KRACK Attack to Obtain Sensitive Information", Mobile, Secure, and Programmable Networking, pp. 270-276, 2019. Available: https://link-springer-com.proxy.lib.sfu.ca/chapter/10.1007/978-3-030-03101-5_22. [Accessed 10 April 2019].

[2] R. Fontes and C. Rothenberg, "On the Krack Attack: Reproducing Vulnerability and a Software-Defined Mitigation Approach", Pdfs.semanticscholar.org, 2017. [Online]. Available: https://pdfs.semanticscholar.org/75a9/0c06511671f0d3e9175c8cfb3ce5ed3ebd84.pdf. [Accessed: 5 April 2019].

[3]M. Vanhoef, "KRACK Attacks: Bypassing WPA2 against Android and Linux", *YouTube*, 2019. [Online]. Available: https://www.youtube.com/watch?time_continue=3&v=Oh4WURZoR98. [Accessed: 10 April 2019].

[4] Valencynetworks.com, “Cyber Attacks Explained Man In The Middle Attack”, valencynetworks.com. [Online]. Available: http://www.valencynetworks.com/articles/cyber-attacks-explained-man-in-the-middle-attack.html . [Accessed: 10 April 2019].

[5] C. Kohlios and T. Hayajneh, "A Comprehensive Attack Flow Model and Security Analysis for Wi-Fi and WPA3", Electronics, vol. 7, no. 11, p. 284, 2018. Available: https://www.mdpi.com/2079-9292/7/11/284/htm. [Accessed 13 April 2019].

[6] M. Vanhoef and F. Piessens, "Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2", Papers.mathyvanhoef.com, 2017. [Online]. Available: https://papers.mathyvanhoef.com/ccs2017.pdf. [Accessed: 10 April 2019].

[7] Search Security, “cipher block chaining (CBC)”, searchsecurity.techtarget.com. [Online]. Available: https://searchsecurity.techtarget.com/definition/cipher-block-chaining. [Accessed: 13 April 2019].

[8] Cipher Block Chaining, Available: https://upload.wikimedia.org/wikipedia/commons/thumb/4/47/PCBC_encryption.svg/800px-PCBC_encryption.svg.png. [Accessed: 13 April 2019].

[9] Cryptosmith, “Cybersecurity education and service”, Cryptosmith. [Online]. Available: https://cryptosmith.com. [Accessed: 12 April 2019].

[10] Cryptography, “Taking advantage of one-time pad key reuse?”, Crypto Stackexchange. [Online]. Available: https://crypto.stackexchange.com/questions/59/taking-advantage-of-one-time-pad-key-reuse. [Accessed: 12 April 2019].

[11] Brainslayer, “Changeset 33525”, trac, 16 Oct 2017. Available: https://svn.dd-wrt.com//changeset/33525. [Accessed 9 April 2019].

[12] J. Malinen, “Index of /security/2017-1”, w1.fi, Jan 2017. Available: https://w1.fi/security/2017-1/. [Accessed 13 April 2019].

[13] HackerSploit, “KRACK Attack - Proof Of Concept”, Youtube, 2 October 2018. Available: https://www.youtube.com/watch?v=Gb8h6M22a6o. [Accessed: 10 April 2019].

[14] S. Naitik, P. Vernekar and V. Shetty, "Mitigation of Key Reinstallation Attack in WPA2 Wi-Fi networks by detection of Nonce Reuse", International Research Journal of Engineering and Technology(IRJET), vol. 05, no. 05, p. 1531, 2018. Available: https://www.irjet.net/archives/V5/i5/IRJET-V5I5290.pdf. [Accessed 10 April 2019].

[15] T. Chin and K. Xiong, "KrackCover: A Wireless Security Framework for Covering KRACK Attacks", Wireless Algorithms, Systems, and Applications, vol. 10874, pp. 733-739, 2018. Available: https://link-springer-com.proxy.lib.sfu.ca/chapter/10.1007/978-3-319-94268-1_60. [Accessed 4 April 2019].

9. Appendix

The code that simulates the WiFi network as well as the simulation aspect of it.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/aodv-module.h"
#include "ns3/olsr-module.h"
#include "ns3/dsdv-module.h"
#include "ns3/ipv4-static-routing-helper.h"
#include "ns3/ipv4-list-routing-helper.h"
#include "ns3/applications-module.h"
#include "ns3/netanim-module.h"
#include "ns3/constant-velocity-mobility-model.h"
#include "ns3/object-factory.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>


NS_LOG_COMPONENT_DEFINE ("wireless_modified");

using namespace ns3;

Ptr<ConstantVelocityMobilityModel> cvmm;
double position_interval = 1.0;
std::string tracebase = "scratch/wireless_modified";

void printPosition()
{
  Vector thePos = cvmm->GetPosition();
  Simulator::Schedule(Seconds(position_interval), &printPosition);
  std::cout << "position: " << thePos << std::endl;
}

void stopMover()
{
  cvmm -> SetVelocity(Vector(0,0,0));
}

int main (int argc, char *argv[])
{
  std::string phyMode = "DsssRate1Mbps";

  int bottomrow = 2;
  int spacing = 150;
  int mheight = 150;
  int brheight = 50;

  int X = (bottomrow-1)*spacing+250;

  int packetsize = 500;
  double factor = 0.75;
  int endtime = (int)100*factor;
  double speed = (X-1.0)/endtime;
  double bitrate = 80*1000.0/factor;
  uint32_t interval = 1000*packetsize*8/bitrate*1000;
  uint32_t packetcount = 200000*endtime/ interval;
  std::cout << "interval = " << interval <<", rate=" << bitrate << ", packetcount=" << packetcount
<< std::endl;

  CommandLine cmd;
  cmd.Parse (argc, argv);

  Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue
("2200"));
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue
("2200"));
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", StringValue
(phyMode));

  NodeContainer fixedpos;
  fixedpos.Create(bottomrow);
  Ptr<Node> lowerleft = fixedpos.Get(1);
  Ptr<Node> mover = CreateObject<Node>();

  WifiHelper wifi;
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
  wifi.SetRemoteStationManager ("ns3::AarfWifiManager");
  YansWifiPhyHelper wifiPhyHelper =  YansWifiPhyHelper::Default ();


  YansWifiChannelHelper wifiChannelHelper;
  wifiChannelHelper.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannelHelper.AddPropagationLoss ("ns3::RangePropagationLossModel", "MaxRange",
DoubleValue(250));
  Ptr<YansWifiChannel> pchan = wifiChannelHelper.Create ();
  wifiPhyHelper.SetChannel (pchan);

  WifiMacHelper mac;
  mac.SetType ("ns3::ApWifiMac");
  NetDeviceContainer devices = wifi.Install (wifiPhyHelper, mac, fixedpos);
  devices.Add (wifi.Install (wifiPhyHelper, mac, mover));

  MobilityHelper sessile;
  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
  int Xpos = 200;
  for (int i=0; i<bottomrow; i++) {
        positionAlloc->Add(Vector(Xpos, brheight, 0.0));
        brheight += spacing;
  }
  sessile.SetPositionAllocator (positionAlloc);
  sessile.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
  sessile.Install (fixedpos);

  Vector pos (0,mheight*-1,0);
  Vector vel (speed, 0, 0);
  MobilityHelper mobile;
  mobile.SetMobilityModel("ns3::ConstantVelocityMobilityModel");
  mobile.Install(mover);
  cvmm = mover->GetObject<ConstantVelocityMobilityModel> ();
  cvmm->SetPosition(pos);
  cvmm->SetVelocity(vel);
  std::cout << "position: " << cvmm->GetPosition() << " velocity: " << cvmm->GetVelocity() <<
std::endl;
  std::cout << "mover mobility model: " << mobile.GetMobilityModelType() << std::endl;
  std::cout << "speed " << speed << std::endl;

  AodvHelper aodv;
  OlsrHelper olsr;
  DsdvHelper dsdv;
  Ipv4ListRoutingHelper listrouting;
```

```
listrouting.Add(olsr, 10);

InternetStackHelper internet;
internet.SetRoutingHelper(listrouting);
internet.Install (fixedpos);
internet.Install (mover);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (devices);
uint16_t port = 80;
Address sinkaddr(InetSocketAddress (Ipv4Address::GetAny (), port));
Config::SetDefault("ns3::UdpServer::Port", UintegerValue(port));

Ptr<UdpServer> UdpRecvApp = CreateObject<UdpServer>();
UdpRecvApp->SetStartTime(Seconds(0.0));
UdpRecvApp->SetStopTime(Seconds(endtime+60));
lowerleft->AddApplication(UdpRecvApp);

Ptr<Ipv4> m4 = lowerleft->GetObject<Ipv4>();
Ipv4Address Maddr = m4->GetAddress(1,0).GetLocal();
std::cout << "IPv4 address of mover: " << Maddr << std::endl;
Address moverAddress (InetSocketAddress (Maddr, port));

Config::SetDefault("ns3::UdpClient::MaxPackets", UintegerValue(packetcount));
Config::SetDefault("ns3::UdpClient::PacketSize", UintegerValue(packetsize));
Config::SetDefault("ns3::UdpClient::Interval",   TimeValue (MicroSeconds (interval)));

Ptr<UdpClient> UdpSendApp = CreateObject<UdpClient>();
UdpSendApp -> SetRemote(Maddr, port);
UdpSendApp -> SetStartTime(Seconds(0.0));
UdpSendApp -> SetStopTime(Seconds(endtime));
mover->AddApplication(UdpSendApp);

wifiPhyHelper.EnablePcap (tracebase, devices);

AsciiTraceHelper ascii;
wifiPhyHelper.EnableAsciiAll (ascii.CreateFileStream (tracebase + ".tr"));

AnimationInterface anim (tracebase + ".xml");
anim.SetMobilityPollInterval(Seconds(0.1));
```

```
  Simulator::Schedule(Seconds(position_interval), &printPosition);

  Simulator::Schedule(Seconds(endtime), &stopMover);

  Simulator::Stop(Seconds (endtime+60));
  Simulator::Run ();
  Simulator::Destroy ();

  int pktsRecd = UdpRecvApp->GetReceived();
  std::cout << "packets received: " << pktsRecd << std::endl;
  std::cout << "packets recorded as lost: " << (UdpRecvApp->GetLost()) << std::endl;
  std::cout << "packets actually lost: " << (packetcount - pktsRecd) << std::endl;

  return 0;
}
```