Enacting Authenticated-OCI in WPA2, can you KRACK?

Anand Agrawal, Ashlin Chirakkal, Urbi Chatterjee, and Rajib Ranjan Maiti

***Abstract*—WPA2 security protocol suite suffers an attack, called Key Reinstallation Attack (KRACK), since 2017 and the severity of this attack has led to the proposal of a new security protocol suite, called WPA3. However, WPA3 may also suffer KRACK due to a vulnerability in its DragonFly authentication mechanism. KRACK exploits a vulnerability in which a repetition of *M3* in the 4-way handshake by a channel-based MitM attacker causes the supplicant to install a known pairwise transient key (PTK). In this paper, we address the problem of KRACK by proposing to include authenticated operating channel information (A-OCI)[1] within *M2* in a 4-way handshake. In particular, A-OCI includes three pieces of information, a signature, operating channel frequency, and operating channel number, totaling 23 bytes within the RSN field in *M2* that has a provision to include upto 256 bytes. We implement our proposed solution in the existing code base of WPA2 and empirically validate the operation of the updated 4-way handshake. Further, we modify an existing formal verification framework of WPA2 in Tamarin prover and show that, unlike state-of-the-art, our proposed solution satisfies the integrity of operating channel, nonce, and authentication in the handshake. We analytically show that our proposed solution can satisfy the recently proposed amendments in WPA3, i.e., protected management frame (PMF) and operating channel validation (OCV), yet does not require to adopt WPA3, which is a critical process.**

***Index Terms*—Wi-Fi, KRACK, WPA2/WPA3**

I. Introduction

Globally, the market share of Wi-Fi (IEEE 802.11) technology has increased significantly in recent times, possibly because of the fast adoption of IoT [2]–[4]. Consequently, the security of Wi-Fi networks has become increasingly important to security researchers to resolve recently discovered attacks, like KRACK [5] or similar attacks [6]–[8]. The security of Wi-Fi has already gone through a series of updates and upgrades, ranging from 802.11n/i/ac/ax versions [9], [10]. WPA2 is the latest security protocol suite available and enabled in vast majority of commercial Wi-Fi devices. It has been operating for over 20 years and is expected to continue for many more. It has been operating for over 20 years and is expected to continue for many more. In 2017, the demonstration of KRACK [5] has caused significant turbulence to the security of WPA2, where the attack is due to a vulnerability in the 4-way handshake in WPA2 protocol. In this paper, we address the problem of this vulnerability without requiring to upgrade the security protocol suite to WPA3 or its related amendments.

In particular, the 4-way handshake in WPA2 involves four messages, called *M1*, *M2*, *M3* and *M4* that are exchanged in order, where *M1* and *M3* are transmitted by the authenticator, i.e., the Wi-Fi access point (AP), and *M2*, and *M4* are transmitted by the supplicant, i.e., the Wi-Fi client (client). The 4-way handshake suffers a vulnerability that a duplicate *M3* causes the client to install a previously known key, called PTK, that is derived from the pairwise master key (PMK), MAC addresses of the supplicant and authenticator, and their corresponding nonce i.e., ANonce and SNonce shared in *M1* and *M2* by the authenticator and the supplicant, respectively. The PTK is then used to ensure the security of the actual data transfer over Wi-Fi. Because the problem is found to be in the protocol standard itself, rather than in any specific implementation, the Wi-Fi alliance has proposed a new protocol suite, called WPA3, that attempted to resolve the vulnerability. The exploitation of the vulnerability that causes KRACK depends on another attack called channel-based-MitM [6], [11], [12], where the client is forced to operate on a channel different from the one in which the AP is active. The channel-based-MitM is possible due to unsolicited channel switching announcements (CSA). In other words, the AP or the client has no mechanism to validate their operating channel. This has led to an amendment in the newly proposed WPA3 [13]. Nevertheless, certain solution strategies have been proposed to counter KRACK, Channel-based-MitM, or similar attacks. For example, these attacks can be detected by a third-party device [14], [15], or the attacks can be verified using a formal verification framework [16], [17].

Further, a method to verify dynamic channel switches has been proposed in [18], where any forged CSA elements in beacon frames can be detected by a supplicant if the mechanism of protected management frame (PMF) is enabled by the authenticator. Basically, enabling PMF ensures secure transfer of operating channel information (OCI). This work also proposes to include OCI in *M2* and *M3* of the 4-way handshake to avoid channel-based MitM. Our work in this paper is closely related to this proposal, where we demonstrate an exact implementation of including authenticated channel information into *M2* in the existing code base of WPA2 protocol suite. Our proposed method is compliant with the amendment of operating channel validation clause in WPA3 [13], [19].

A. Motivation

We motivate this work with the fact that there is a high demand for an effective mitigation strategy in WPA2 to counter

Anand Agrawal, Ashlin Chirakaal, and Rajib Ranjan Maiti are with BITS Pilani, Hyderabad Campus, India. U. Chatterjee is with IIT Kanpur, India.

[1]The idea of guarding the Wi-Fi 4-Way Handshake against the channel-based MiTM has been presented by us as a poster paper [1]. We consider the work in this paper as a new contribution. We have proposed a new scheme to include extra bytes in RSNIE. An update to the supplicant and the authenticator code base. We have empirically and formally validated our proposed scheme for the first time in this work.

the vulnerabilities related to KRACK and similar attacks. This is possibly because it does not call for a certification for newly proposed WPA3 and the adoption of WPA3 may not be a reality anytime soon. Certain reports [20] anticipate that both WPA2 and WPA3 will coexist for a long time to come. Further, some recent studies [21] have shown that WPA3 can still suffer from certain vulnerabilities that lead to attacks similar to KRACK in WPA3, and hence Wi-Fi Alliance has come up with the amendment [13] of operating channel validation into WPA3. Therefore, it is one of the tough challenges for the vendors to easily adapt to WPA3. Further, to date, there are no proper guidelines for migrating to WPA3, except that the specification of WPA3 [19] is publicly available. In fact, the amendment [19] mandates the use of PMF after the first time completing the establishment of a secure connection. This motivates us to explore a prevention strategy for attacks like KRACK without requiring us to migrate to WPA3 yet enable the PMF functionalities.

Another challenging task for updating the code base of WPA2 protocol suite is to deal with its enormous size, i.e., in the number of lines of codes. It may not be very encouraging for a programmer to debug and update the code base of WPA2 in general and 4-way handshake in particular. We motivate ourselves to dig into the code base with the fact that we have already developed a deep understanding based on the works in [14], [15] of the secure connection establishment in WPA2 protocol suite and the root cause of the attacks like KRACK. Consequently, we deep dive into the code base to identify the precise functions that can facilitate the 4-way handshake.

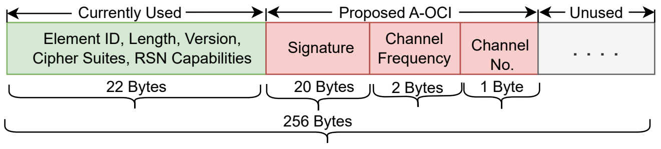


Fig. 1: RSN Information Element Frame Format.

B. Our Contribution

We have carefully analyzed the state machines of both the supplicant (a supplicant is a client in Wi-Fi) and the authenticator (an authenticator is an access point (AP) in Wi-Fi); the state machines are used to establish a secure connection. The state machine at supplicant is executed every time it initiates a connection establishment by sending an authentication, i.e., *Auth*, frame. The authenticator executes its state machine independently for every supplicant. We have considered that successful execution of any of the state machines is achieved when the supplicant transmits *M4* in the 4-way handshake. By deeply analyzing the protocol standard, we have identified the derivations of key elements and attached them with the respective states in both state machines. We have observed the exact parameters being transferred in each of *M1*, *M2*, *M3* and *M4*, and the key elements generated as a result of receiving these messages, i.e., as a result of state transition. The pair-wise transient key (PTK) is derived at *FT-PTK-START* state at the supplicant as a result of receiving *M1*, and at the authenticator as a result of receiving *M2*. Robust Security Network Information Element (RSNIE) field in *M2* currently has a provision of including 256 bytes (according to [9], Section $\oint$ 7.3.2.25) of data, however only 22 bytes are currently used by most of the vendors. We propose to append an authenticated operating channel information (A-OCI) of 23 bytes (20 bytes of signature + 2 bytes of channel frequency + 1 byte of channel number) making the RSNIE of 22 + 23 = 45 bytes (as shown in Figure 1). In other words, we propose for the supplicant to include A-OCI having three data elements, namely the signature ($Sign_S = HMAC\text{-}SHA1(PTK_S||ANonce||SNone||CF_S||CN_S)$), the operating channel frequency (CF_S), and the operating channel number (CN_S) as part of RSNIE in *M2*. Note that $HMAC\text{-}SHA1()$ is already used to compute MIC in *M2*, *M3*, and *M4* and we propose to execute it for exactly one more time. On the other hand, the authenticator recalculates the signature ($Sign_A = HMAC\text{-}SHA1(PTK_A||ANonce||SNonce$ $||CF_A||CN_A)$) using the channel frequency CF_A and the channel number CN_A in addition to the other parameters, and checks with the received signature $Sign_S$. Only if the verification succeeds, the authenticator proceeds to transmit *M3*, otherwise, it transmits an authenticated deauthentication, i.e., *deauth*, frame to the supplicant.

Next, we carefully investigate the WPA2 code base of both supplicant and authenticator to identify the exact files and the functions that carry out the 4-way handshake. Our investigation reveals that `wpa.c` contains those functions for supplicant and `wpa_auth.c` for authenticator. A large number of *events* is involved in `wpa_auth.c`, such as `"WPA PTK_INITIALIZE"`, `"PTK_AUTHENTICATION"`, and `"PTK_START"`, and these are used inside "wpa_auth_sm_event()" function. Similarly, a large number of events are present in "wpa_supplicant_event()" function in "wpa_supplicant.c". We have analyzed the purpose of these events and it turns out that they mostly causes the state changes within the state machine. Reflecting on the standard, none of the codes corresponding to the state machines includes any function for including or verifying the operating channel information; this is the root cause of unsolicited channel switching at the supplicant resulting in attacks like Channel-based MiTM, evil twin, and KRACK. Following this, we have newly introduced the functions for authenticated operating channel information (A-OCI) in both "wpa.c" and "wpa_auth.c". Next, We empirically validate the working of the updated code base in our experimental setup, where we capture and analyze the Wi-Fi traffic using Wireshark for cross validation and comparison with state-of-the-art.

Finally, to formally validate our proposed solution, we have updated the formal verification framework proposed in [22]. In particular, the existing framework has a set of rules that is used to define a set of properties of WPA2 protocol as a whole and a set of properties of 4-way handshake in particular. The major emphasis of this framework has been to formally demonstrate the properties that get violated as a result of KRACK in 4-way handshake. Basically, a set of lemmas corresponding to the rules is shown to be violated as a result of KRACK. We have deeply analyzed this framework and newly added a number of rules and lemmas corresponding to our proposed scheme of including A-OCI in *M2*. Finally, we have experimented using the updated framework to formally demonstrate that attacks like channel-based MitM and KRACK are not possible because the new rules that ensure the authenticity of the operating channel in *M2* at the supplicant and the authenticator. Our contributions are summarized as follows:

1) We have proposed and implemented an in-standard mitigation strategy to counter KRACK and related attacks on 4-way handshake of the WPA2 protocol suite. In particular, our proposed solution includes an extra 23 bytes of A-OCI as part of RSNIE in *M2*, where A-OCI is first computed by the supplicant and transferred through *M2* and then the authenticator sends *M3* only if A-OCI is verified successfully.
2) We have newly introduced extra *9* and *8* lines of codes in "wpa.c" and "wpa_auth.c" respectively. We have analytically and empirically shown that the execution of the new code base of WPA2 has a negligible increase in the overall execution time of 4-way handshake, which is mainly because of executing $HMAC\text{-}SHA1()$ for exactly one more time each at supplicant and at authenticator.
3) We have empirically validated our proposed solution by capturing the Wi-Fi traffic as a third party during the secure connection establishment using the updated WPA2 codes of supplicant and authenticator. Our analysis of the captured traffic shows that the A-OCI is present as part of RSNIE in *M2*. Alternately, the traffic contains *deauth* frames from the authenticator when a supplicant either sends A-OCI containing a different channel than the authenticator or it omits A-OCI completely.
4) Finally, we have updated *9* rules and *8* lemmas, in the existing formal analysis framework of WPA2 in the tamarin prover. The results of our formal analysis show that the updated rules corresponding to our proposed solution protect PTK and PMK (pair-wise master key) together with the integrity of operating channel, Nonces, and authenticity of *M2*. Comparing with state-of-the-art, the execution time of the updated formal verification framework has a negligible increase (about 0.2s on an average) only while ensuring the operating channel integrity.

Rest of the paper is organized as follows. Section II describes the state machines of Supplicant and Authenticator. Section III characterizes KRACK. The system and attacker models are discussed in Section IV. Our proposed scheme is stated in Section V. Experimental setup and results are presented in Section VI. Formal analysis of the proposed solution is described in Section VII. Section VIII discusses the feasibility of the proposed scheme. Related works are stated in Section IX, followed by the conclusion in Section X.

II. State Machines in WPA2

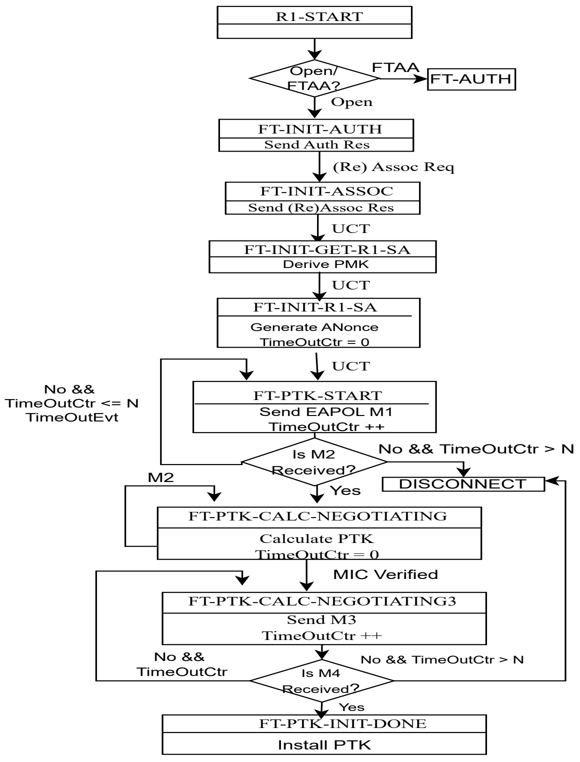
This section briefly describes the state machines of the supplicant and the authenticator in connection to the derivation of key elements in WPA2.

A. Probing State Machine of Authenticator

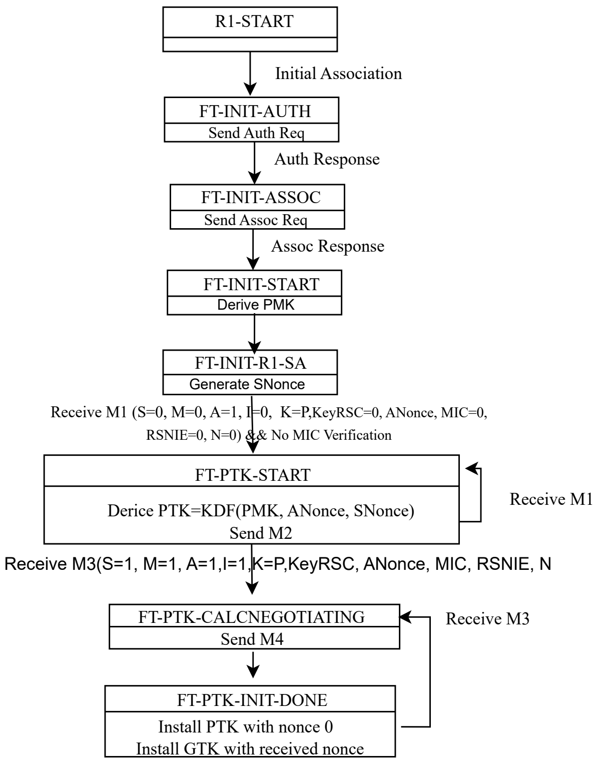
According to IEEE 802.11-2016 standard [10], an authenticator executes Fast Transition protocol Authentication Algorithms (FTAA) to accomplish an authentication process when requested by a supplicant. Figure 2a shows a simplified version of the state machine representing FTAA. The state machine begins with *R1-START* state, the authenticator waits for an authentication request from a supplicant in this state. Such a request is indicated by *MLME-AUTHENTICATE.indication()* that suggests the kind of parameters expected from the supplicant. The state machine checks if *AuthenticationType* is *Open* or *FTAA*. If so, then it tries to prepare a response, which is indicated by the primitive *MLME-AUTHENITCATE.response ()* (as specified in Section ∮ 6.3.5.5.2 in [10]). Notably, *ResultCode* parameter in the response can contain any of 108 codes (as specified in Section ∮ 9.4.1.9 in [10]), one of them is SUCCESS which we consider to continue with the state machine. Several of the parameters in both authentication request and response are optional, leaving certain scopes for the vendors. The *resultcode* as SUCCESS within the authentication response frame causes the state transition to FT-INIT-AUTH; otherwise it transmits an authentication response frame with an appropriate *resultcode* being in R1-START state.

In *FT-INIT-AUTH* state the authenticator waits to receive an association request, indicated by the primitive MLME-ASSOCIATE.requ-est(). Only *RSN* parameters specify the fields related to the cipher suite, indicating the preferences related to pairwise ciphersuite, authentication key management (AKM) suite, and a list of pair-wise master key identifiers (PMKIDs). Additionally, *supported Channels* specifies the preferred channel that the supplicant wishes to continue if the authenticator provides that option possibly in a beacon frame. The authenticator sends the response using MLME-ASSOCIATE.response() (specified in Section ∮ 6.3.7.5.2 in [10]). After verification, the authenticator prepares and sends a response indicating the result of the verification in the parameter *ResultCode* using an MLME-ASSOCIATE.response(). We consider *ResultCode* as SUCCESS to continue with the state machine, i.e., the state machine transits to FT-INIT-ASSOC state after sending the association response frame. The state machine then unconditionally transits (UCT) to FT-INIT-GET-R1-SA state; in this state, 256-bit pairwise master key (PMK) is derived based on the RSN parameters received in association request along with its own parameters (details section V-A). Once PMK is derived, the state machine unconditionally transits to FT-INIT-R1-SA state; in this state, 256-bit authenticator nonce, i.e., ANonce, is calculated by calling a pseudorandom function, i.e., PRF-256(Random number, “Init Counter”, Local MAC Address || Time) (specified in Section ∮ 12.7.5 [10]), and the time-out counter is reset.

Then, it unconditionally transits to the FT-PTK-START state; in this state, a 4-way handshake is initiated provided PMK and ANonce are already present (specified in Section ∮ 4.10.3.2 in [10]). In the 4-way handshake, the first message, i.e., EAPOL Message 1 (*M1* in short) is prepared consisting of ten main parameters as: S - is key exchange complete? M - is MIC available? A - is acknowledgment required?



(a) AP's state machine for secure Handshake.



(b) Supplicant's state machine for secure Handshake.

Fig. 2: State machines of (a) supplicant and (b) authenticator that establish secure Wi-Fi connection.

I - is installed? *K* - key type, *KeyRSC* - is it Key RSC? *ANonce/Snonce* - is authenticator or supplicant nonce? *MIC* - message integrity code, *RSNIE* - RSN information element, and *N* - index of group temporal key (GTK). For example, M1(0, 0, 1, 0, P, 0, ANonce, 0, 0, 0) indicates the respective values of ten parameters in *M1* specifying key exchange is not completed, MIC is not available, and so on. After sending *M1*, TimeOutCtr is kept on incremented till *M2* is not received or it time outs.

Assuming M2(0, 1, 0, 0, P, 0, SNonce, MIC, RSNIE, 0) is received within time out period, the state machine transits to FT-PTK-CALC-NEGOTIATING state. Note that *M2* brings in SNonce and an MIC for the first time so that the integrity of *M2* may be checked, and 0 in the last parameter indicates that the handshake is not for GTK. *Note that we shall make use of RSNIE in our proposed solution.* In this state, the authenticator calculates the pairwise transient key (PTK) (detailed in Section V-B), and the TimeOutCtr is reset. Assuming *MIC* is valid, the state machine transits to FT-PTK-CALC-NEGOTIATING3 state and sends M3(1,1,1,1,P,KeyRSC, ANonce, MIC, RSNIE, N) indicating that the initial key exchange is completed (i.e., S=1), ACK is required for *M3* (i.e., A=1), and information about PTK is ready (i.e., KeyRSC is present), and *M3* is protected with MIC (i.e., MIC is present). Though PTK is calculated, it is stored in a temporal pairwise transient key (TPTK), until *M4* is received. Assuming M4(1, 1, 0, 0, P, 0, 0, MIC, 0, 0) is received within time-out period, the state machine transits to the FT-PTK-INIT-DONE state. Through *M4*, the authenticator is informed that the supplicant has completed key exchange (S=1), and the integrity of *M4* can be checked using MIC.

B. Probing State Machine of Supplicant

Similar to authenticator's state machine, the supplicant's state machine starts with R1-START. When *initial association* is indicated, the state machine transits to FT-INIT-AUTH state and sends out an authentication request using the primitive MLME-AUTHENTICAT-E.request() specifying that the supplicant wishes to use *Open System authentication* in the *AuthenticationType* parameter.

Assuming that authentication response is received using the primitive MLME-AUTHENTICATE.confirm(), the state machine transits to FT-INIT-ASSOC state if *ResultCode* is SUCCESS in the response. In this state, the state machine transmits an association/(re)association request using the primitive MLME-ASSOCIAT-E.request() specifying the preferences like *supported channel* and RSN parameters.

Assuming that an association response is received using the primitive MLME-ASSOCIATE.confirm(), it transits to FT-INIT-START state if *ResultCode* in the response is SUCCESS. In this state, it derives 256-bit PMK (detailed in Section V-A). Then, it unconditionally transits to FT-INIT-R1-SA state and calculates 256-bit SNonce using a pseudo-random function, i.e., PRF-256(Random number, "Init Counter", Local MAC Address || Time), and waits for *M1*.

Upon receiving *M1*, the state machine enters into the FT-PTK-START state and then calculates PTK using prf-x(), where ANonce is a parameter, and stores it in TPTK (detailed in Section V-B). In this state, it transmits M2(0, 1, 0, 0, P, 0, SNonce, MIC, RSNIE, 0) providing MIC calculated using HMAC-SHA1, and waits for *M3*.

Once *M3* is received, it transits to the FT-PTK-CALC-NEGOTIAT-ING state, if MIC verification is successful. *M3* also indicates at the supplicant that the authenticator now has derived PTK and stored it in TPTK. So, the supplicant transmits M4 and unconditionally transits to FT-PTK-INIT-DONE state. In this state, TPTK is transferred to PTK, this is called PTK installed, and *nonce* is assigned 0 which is then used as an initialization vector (IV) in the subsequent data encryption suite. Thus, every time a 4-way handshake is completed, a new *PTK* and a *nonce* are freshly generated.

Note that due to the repeated *M3* and unconditional transition to FT-PTK-INIT-DONE state, the supplicant gets *nonce* as 0 which is exploited in the attacks like KRACK [5]. Basically, the attacker repeatedly injects *M3* causing the supplicant to keep using *nonce* as 0 and the attacker can now decrypt the encrypted data packets from the supplicant using this *nonce*.

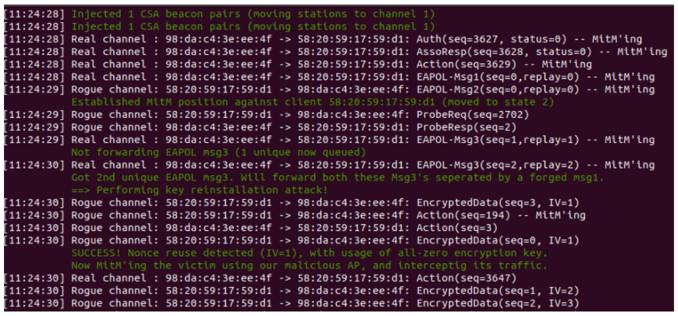
III. Characteristics of KRACK

Key reinstallation attack (KRACK) is a combination of several attacks, e.g., channel-based MiTM, evil twin, replay, crafted packet injection, de-authentication, and jamming attacks on Wi-Fi handshake phase. The attacker forces a supplicant to switch the operating channel, possibly by transmitting forged channel-switching announcements, to the one that the evil twin AP is set; we call this channel as rogue channel. And, the attacker impersonates the supplicant on the channel that the actual AP is operating; call this channel as real channel. Essentially, the attacker impersonates both the AP and the supplicant on two different channels. Such a setting allows the attacker to replicate any arbitrary Wi-Fi packet on either channel. This attack basically exploits the vulnerability that a supplicant re-initializes the *nonce* to 0 when *M3* is replicated on the rogue channel (a detailed analysis may be found in [14]).

Listing1, shows a simplified version of the flow of actions to launch KRACK [23]. In the *KRACKAttack* class, three main functions, namely *send_csa_beacon(), handle_to_client_pairwise() and handle_from_client_pairwise()*, are invoked from *run()* to launch the attack. *send_csa_beacon()* function injects on the real channel a beacon frame specifying the rogue channel; the client is expected to catch it and change its channel to the rogue channel. *handle_to_client_pairwise()* function stores *M1* and two unique *M3*s in two lists, that has been sniffed on real channel, and then transmits on rogue channel a sequence <*M3*, *M1*, *M3*> to provide an impression to the supplicant that a new handshake has been initiated by the real AP; thus attempting to launch channel-based MitM. *handle_from_client_pairwise()* function checks if a message received on the rogue channel from the client reuses an IV. If so, then it further checks if the corresponding *keystream* is matched with an old *keystream*. If so, then the script reports that KRACK is successful. Alternately, if *keystream* is new then it is stored for future use.

```python
class KRAckAttack():
 def send_csa_beacon(self, numbeacons=1):
  csabeacon = append_csa(beacon, rogue_channel, 1)
  self.sock_real.send(csabeacon)
  log(STATUS, "Injected CSA beacon pairs")
 def handle_to_client_pairwise(self, client, p):
  eapolnum = get_eapol_msgnum(p)
  if eapolnum == 1 and gotMitm:
   log(DEBUG, "Storing M1"),client.store_msg1(p)
  else if eapolnum == 3 and gotMitm:
   client.add_if_new_msg3(p)
   if len(client.msg3s) >= 2:
    log("Got 2nd unique M3 => Performing KRACK")
    pkt_list = client.msg3s
    p = set_replaynum(client.msg1, get_replaynum)
    pkt_list.insert(1, p)
    for p in pkt_list: self.sock_rogue.send(p)
    client.msg3s = []
    client.attack_start()
 def handle_from_client_pairwise(self, client, p):
  if client.is_iv_reused():
   keystream = xorstr(plaintext_M4, encrypted_M4)
   if keystream == client.get_keystream():
    log("SUCCESS! Nonce reuse detected...")
    self.hostapd_add_allzero_client(client)
  client.save_iv_keystream(keystream)
 def run():
  send_csa_beacon(numbeacons)
  handle_to_client_pairwise(client, p)
  handle_from_client_pairwise(client, p)
if __name__ == "__main__":
 attack = KRAckAttack(nic_real_mon, nic_rogue_ap,
    nic_rogue_mon, ssid, target, continuous_csa)
 attack.run()
```

Listing 1: Code Snippet of the KRACK attack in Python.



```
[11:24:28] Injected 1 CSA beacon pairs (moving stations to channel 1)
[11:24:28] Injected 1 CSA beacon pairs (moving stations to channel 1)
[11:24:28] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: Auth(seq=3627, status=0) -- MitM'ing
[11:24:28] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: AssoResp(seq=3628, status=0) -- MitM'ing
[11:24:28] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: Action(seq=3629) -- MitM'ing
[11:24:28] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: EAPOL-Msg1(seq=0,replay=0) -- MitM'ing
[11:24:29] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: EAPOL-Msg2(seq=0,replay=0) -- MitM'ing
           Established MitM position against client 58:20:59:17:59:d1 (moved to state 2)
[11:24:29] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: ProbeReq(seq=2702)
[11:24:29] Rogue channel: 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: ProbeResp(seq=2)
[11:24:29] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: EAPOL-Msg3(seq=1,replay=1) -- MitM'ing
           Not forwarding EAPOL msg3 (1 unique now queued)
[11:24:30] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: EAPOL-Msg3(seq=2,replay=2) -- MitM'ing
           Got 2nd unique EAPOL msg3. Will forward both these Msg3's seperated by a forged msg1.
           ==> Performing key reinstallation attack!
[11:24:30] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: EncryptedData(seq=3, IV=1)
[11:24:30] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: Action(seq=194) -- MitM'ing
[11:24:30] Rogue channel: 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: Action(seq=3)
[11:24:30] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: EncryptedData(seq=0, IV=1)
           SUCCESS! Nonce reuse detected (IV=1), with usage of all-zero encryption key.
           Now MitM'ing the victim using our malicious AP, and interceptig its traffic.
[11:24:30] Real channel : 98:da:c4:3e:ee:4f -> 58:20:59:17:59:d1: Action(seq=3647)
[11:24:30] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: EncryptedData(seq=1, IV=2)
[11:24:30] Rogue channel: 58:20:59:17:59:d1 -> 98:da:c4:3e:ee:4f: EncryptedData(seq=2, IV=3)
```

Fig. 3: Execution report of KRACK script.

We have run this attack script and captured the corresponding Wi-Fi traffic. It requires three Wi-Fi adapters to run in different modes; one with managed mode acting as a supplicant, another with master mode acting as an authenticator and the other with monitor mode for sniffing on two channels (rogue and real) to capture traffic on both the channels together. Figure 3 shows the diagnostic output in the attacker's terminal as a result of executing the attack script. Executing this script for a limited number of times may not result in a successful attack; on average one-time execution of this script takes about 6.8s. The attack starts by injecting two beacon frames, then obtaining the MitM position, followed by forwarding *M3* and finally showing success in detecting a *nonce* reuse. The important state transitions and events are highlighted in green color in Figure 3.

IV. System and Attacker Model

We envision a system consisting of a number of Wi-Fi hosts together with a Wi-Fi AP, as shown in Figure 4. The AP is configured to use WPA2 protocol suite so that every host can establish a secure Wi-Fi connection by executing 4-way handshake. We consider that a host, like a Laptop or desktop or an AP, executes *hostapd* daemon and hence acts as an authenticator. And, another host, like a smartphone or laptop, executes a Wi-Fi client program and hence acts as a supplicant. The supplicant initiates a Wi-Fi connection establishment by sending authentication request frame to the authenticator. The authenticator is configured to use WPA2 protocol suite and hence the supplicant agrees to use the same protocol to ensure a secure connection establishment. Both the authenticator and supplicant are capable of executing our proposed and updated *hostapd* and supplicant programs respectively.

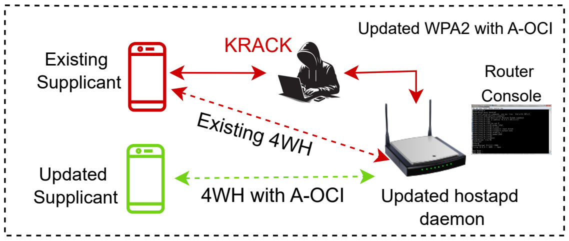


Fig. 4: System and Attacker Model.

We consider that an attacker is physically placed within the coverage area of the authenticator and capable of actively attacking a pair of Wi-Fi hosts. We assume that the attacker has a wide range of capabilities, where it works based on the well-known Dolev-Yao attacker model [24]. In particular, it can overhear, intercept, and generate any arbitrary Wi-Fi frames leading to launch attacks like KRACK. By such capabilities, an attacker can obtain channel-based MiTM, possibly by jamming the real channel, advertising the beacon being *eviltwin*, and offering a high signal strength on the rogue channel [7]. The attacker can also transmit the channel-switching announcement frame from the *eviltwin* so that the supplicant can be tricked to change its current operating channel. Essentially, the attacker has the capability to act as a legitimate authenticator on the rogue channel and as a legitimate supplicant on the real channel. It can receive on the real channel a legitimate Wi-Fi frame and selectively replicate that on the rogue channel and vice versa.

V. Our Proposed Solution

In this section, we provide the details of the derivation of key elements used in WPA2 security protocol suite, the parameters used to derive them, and our proposed solution to prevent the attacks like KRACK on 4-way handshake. Then, we provide the details of the implementation of our proposed solution in the legacy code of both supplicant and authenticator.

A. Derivation and Distribution of PMK

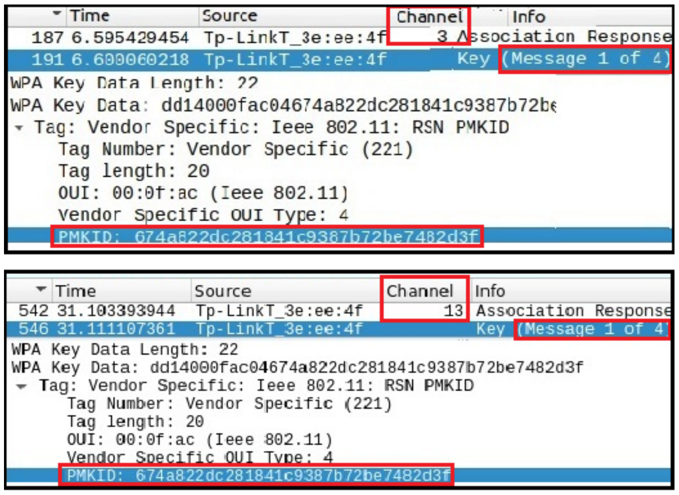
A 256-bit Pairwise Master Key (PMK) is computed using KDF(SSID, PSK, Len(PSK), HMAC-SHA1), where KDF() is a password-based key derivation function, SSID is the SSID of authenticator, PSK is pre-shared key, len() returns the length of PSK, and HMAC-SHA1 indicates the hash-based message authentication function. PSK is conventionally derived from the Wi-Fi password. Hence, it remains unchanged for any supplicant as long as the authenticator uses the same SSID and password. Both the supplicant and the authenticator use KDF() to derive the same set of PMKs. The supplicant sends certain parameters in association request using RSN field, so that the authenticator can compute a list of PMKs (as described in Section II-A). Then, the authenticator sends a PMKID in *M1*, so that the supplicant picks up the correct PMK from the list of PMKs (as specified in ∮ 5.9.5 in [9]). Note that, based on the type of Wi-Fi network, KDF() takes PSK as a parameter in WPA-Personal, whereas it takes "AAA" (Authentication, Authorization, and Accounting) key or "MSK" (Master Shared key) in WPA-Enterprise.

B. Derivation and Distribution of PTK

The derivation of PTK depends on ANonce and SNonce. ANonce is provided by the authenticator via M1(S=0, M=0, A=1, I=0, K=P, KeyRSC=0, `ANonce`, MIC=0, RSNIE=0, GTK=0), (these parameters are described in Section II-A). In particular, M=0 and MIC=0 in *M1* indicates that the authenticator has not yet derived PTK. On receiving *M1*, the supplicant derives PTK as `PTK ← PRF-x(PMK, "Pairwise key expansion", Min(AA,SPA) || Max(AA,SPA) || Min(ANonce,SNonce) || Max(ANonce,SNonce))`, where *x* in *PRF-x* is either 512 for TKIP or 384 for CCMP denoting the number of bits in the output, "Pairwise Key Expansion" is a constant string, *AA* and *SPA* denote the MAC addresses of authenticator and supplicant respectively, *Min()* and *Max()* are the function to find minimum and maximum to two parameters. Once computed, PTK is partitioned into chunks to obtain 128-bit Key Confirmation Key (KCK), 128-bit Key Encryption Key (KEK) and 256-bit Temporal Key (TK) in TKIP (or 128-bit TK in CCMP). Then, KCK is used to derive MIC as `MIC ← HMAC-SHA1(KCK, payload)`, and this MIC is present in *M2*, *M3* and *M4* (specified in both ∮ 8.5.1.2 in [9] and ∮ 12.7.6.3 in [10]). And, KEK is used to encrypt the values in the key data field in *M3* and *M4*. However, TK is not used in the 4-way handshake; it is used for integrity checks and encryption in MAC service data units (MSDUs).

C. Confirming the Lack of Channel Integrity

So far, we have analytically shown that channel information is not used for the derivation of any of PSK, PMK (or PMKID), or PTK (or MIC). To practically verify the same, we have conducted the following two experiments: i) one supplicant with an authenticator configured on two different channels, channel 3 and channel 13, and ii) two supplicants with an authenticator on one channel, channel 13. Figure 5 shows a sample of Wi-Fi traffic for the first experiment. It turns out that the value in the key data field, i.e., PMKID, remains the same in *M1* in this experiment, indicating that channel information is not part of *M1*. Note that PMKID is generated by considering PMK as one of the parameters. Next, we have seen that MIC remains the same in both channels. Recall that MIC is obtained from the KCK portion of PTK and the PTK is derived by using parameters like ANonce and SNonce. Hence, we confirm that the implementation of WPA2 standard also does not use the channel information as part of any of PMK, PTK or MIC, i.e., none of *M1*, *M2*, *M3* and *M4* care about the channel information.



```
    Time         Source              Channel    Info
187 6.595429454  Tp-LinkT_3e:ee:4f         3 Association Response
191 6.600060218  Tp-LinkT_3e:ee:4f           Key (Message 1 of 4)
WPA Key Data Length: 22
WPA Key Data: dd14000fac04674a822dc281841c9387b72be
▾ Tag: Vendor Specific: Ieee 802.11: RSN PMKID
    Tag Number: Vendor Specific (221)
    Tag length: 20
    OUI: 00:0f:ac (Ieee 802.11)
    Vendor Specific OUI Type: 4
    PMKID: 674a822dc281841c9387b72be7482d3f
```

```
    Time          Source              Channel  Info
542 31.103393944  Tp-LinkT_3e:ee:4f        13  Association Response
546 31.111107361  Tp-LinkT_3e:ee:4f            Key (Message 1 of 4
WPA Key Data Length: 22
WPA Key Data: dd14000fac04674a822dc281841c9387b72be7482d3f
▾ Tag: Vendor Specific: Ieee 802.11: RSN PMKID
     Tag Number: Vendor Specific (221)
     Tag length: 20
     OUI: 00:0f:ac (Ieee 802.11)
     Vendor Specific OUI Type: 4
     PMKID: 674a822dc281841c9387b72be7482d3f
```

Fig. 5: PMKID in M1 at channel 3 and channel 13.

In the second experiment, we have seen that the PMKID value is different for two different supplicants. Thus, this investigation indicates that PMKID is derived based on parameters that include the MAC addresses of the supplicant and the authenticator. Further, the MIC in these two supplicants are different; this is for the obvious reason that PMK and hence PTK and hence KCK are different as the MAC addresses as well as ANonce and SNonce are different. This is in line with the specification in ∮ 8.5.1.2 in [9].

D. Proposed Modification in WPA2 Protocol

We propose to use three additional pieces of information as part of the WPA Key data field in M2, namely a 160-bit signature (Sign), a 16-bit Channel frequency (CF), and an 8-bit channel number (CN); the exact frames carrying these information are shown in Figure 6. In other words, we consider the authenticated operating channel information (A-OCI) as the combination of these three data values, i.e., $A\text{-}OCI = (Sign, CF, CN)$. At the supplicant, Sign $Sign_S$ is computed as $Sign_S = HMAC\text{-}SHA1(PTK_S||ANonce||SNonce||CF_S||CN_S)$, where PTK_S, CF_S, and CN_S are the PTK, channel frequency, and channel number respectively. Note that *HMAC-SHA1()* is already available and used at both the supplicant and the authenticator to derive PTK. At the authenticator, the signature can be verified on receiving *M2* to check if the supplicant is on the same channel by recomputing the Sign $Sign_A$ as $Sign_A = HMAC\text{-}SHA1(PTK_A||ANonce||SNonce||CF_A ||CN_A)$, where PTK_A, CF_A and CN_A are the PTK, channel frequency, and channel number respectively. If the verification fails i.e., $Sign_S \neq Sign_A$, then the authenticator does not send *M3*, instead, it transmits a *deauth* frame which is optionally authenticated using the same three pieces of information.

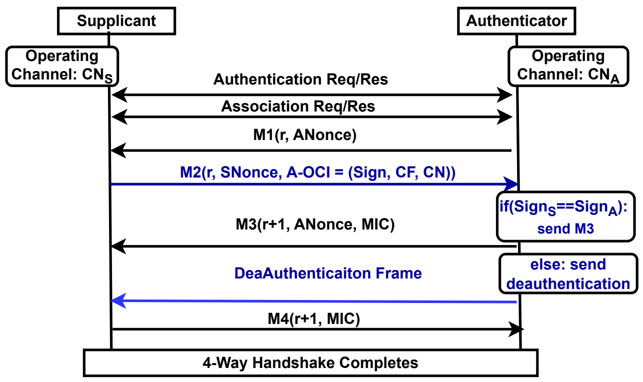

Supplicant
Authenticator
Operating Channel: CN_S
Operating Channel: CN_A
Authentication Req/Res
Association Req/Res
M1(r, ANonce)
M2(r, SNonce, A-OCI = (Sign, CF, CN))
if($Sign_S$==$Sign_A$): send M3
M3(r+1, ANonce, MIC)
else: send deauthentication
DeaAuthenticaiton Frame
M4(r+1, MIC)
4-Way Handshake Completes

Fig. 6: Proposed update (in blue) in 4-way handshake.

E. Analysis of Our Proposed Solution

After transmitting *M2*, the supplicant can receive *M3* in two scenarios. First, the verification at the authenticator has been successful and hence continuing the 4-way handshake. Second, an adversary has either replayed an old *M3* (containing all the correct values, except old ANonce and SNonce), or injects a crafted *M3* as M3(1, 1, 1, 1, P, KeyRSC, ANonce, MIC, RSNIE, N), where *ANonce* can be obtained from *M1*, and *KeyRSC*, *RSNIE*, and *N* from the old trace. In former case, *M3* contains old *ANonce* and *SNonce*, and in the later case, *M3* contains a forged *MIC* or an old *MIC*. In the later case, a valid *MIC* cannot be generated, because it is derived from *PTK*, *PTK* in turn is derived from the tuple <*PMK*, *ANonce*, *SNonce*, MAC addresses of authenticator and supplicant>, *PMK* in turn is derived from *PSK*, and *PSK* is derived from a Wi-Fi password. In other words, the adversary cannot replicate *MIC* until and unless the Wi-Fi password is leaked.

Further, "WPA key data" field contains encrypted RSNIE that contain the authenticator's pairwise cipher suite selection, the encapsulated GTK and the GTK's key identifier. This encryption is performed using *KEK* derived from *PTK*. Hence, similar to *MIC*, an adversary cannot forge this ciphertext unless and until the Wi-Fi password is compromised.

Alternately, if a supplicant receives *M3*, then it checks for the following: i) if the replay counter is fresh, ii) if *ANonce* in *M3* is the same as that in *M1*, iii) if decrypted RSNIE contains certain information identical to the one received in the beacon or probe response frame, iv) if *MIC* is correct. If any of these four checks fails, then the supplicant discards *M3*. Note that the replay counter, *ANonce*, and *MIC* are not encrypted in *M3*, whereas RSNIE is encrypted using *KEK*. Thus, we argue that these checks will fail when *M3* is either replayed or crafted. In the former case, the checks on either the replay counter or *ANonce* will fail, whereas, in the later case, the checks on either decrypted RSNIE or *MIC* will fail because of *KEK* or *KCK* respectively.

F. Implementation of Our Proposed Solution

We have updated the existing C code base [2] [25] for supplicant and authenticator to incorporate our proposed solution. Together the size of this code base is 71.5 MB and the complete code is kept in three main directories, namely *src*, *hostapd*, and *wpa_supplicant*. Identifying the exact C code files and the functions in those files that are relevant to our proposed scheme is a challenging task due to its complex control flow in the program execution. Even, it is challenging to get the code base correctly compiled and rebuilt without error in our local system. We need to separately compile and build the supplicant and the authenticator codes and there are several interdependencies in the whole code base. To identify the exact functions, we look for certain key phrases, like "sending of M1", "processing of M1", "sending of M2", "processing of M2", "sending of M3", "processing of M3", "sending of M4", and "processing of M4", in each of the files.

```
1  int wpa_supplicant_send_2_of_4(struct wpa_sm *sm,
       const struct wpa_eapol_key *key const u8 *wpa_ie
       , const u8 *nonce, ...){
2      ...
3      wpa_hexdump(MSG_DEBUG, "Replay Counter", reply->
       replay_counter, WPA_REPLAY_COUNTER_LEN);
4      u8 *wpa_ie_new=wpa_ie;// Pointer to new IE
5      CFS=get_channel_freq_from_beacon();
6      CNS=get_channel_no_from_beacon();
7      u8 *wpa_ie_sign_S=wpa_supp_sign_2_of_4();
8      wpa_ie_sign_S[20]=CFS; wpa_ie_sign_S[22]=CNS;
9      os_memcpy(wpa_ie_new, wpa_ie_sign_S, 23);
10     key_mic=(u8 *) (reply + 1);
11     WPA_PUT_BE16(key_mic + mic_len, wpa_ie_len);
12     // Copying updated Key Data
13     os_memcpy(key_mic + mic_len + 2, wpa_ie_new,
       wpa_ie_len);
14     os_free(rsn_ie_buf);
15     os_memcpy(reply->key_nonce, nonce, WPA_NONCE_LEN
       );
16     ...
17     return wpa_eapol_key_send();
18 }
```

Listing 2: Code snippet of the updated WPA supplicant.

We found that *wpa_supplicant_process_1_of_4()* and *wpa_suppli-cant_send_2_of_4()* present in *wpa.c* within *rsn_supp* subfolder of *src* directory are responsible for processing *M1* and sending *M2* respectively at the supplicant. Listing 2 shows the part of the code of *wpa_supplicant_send_2_of_4()* that we have updated based on our proposed modification. Among others, this function takes as arguments a pointer to a structure *struct wpa_sm*, a pointer to another structure *struct wpa_eapol_key*, two pointers to const structure *u8*. These structures contain certain parameters to compute different security parameters. For example, the pointer argument *wpa_ie* of *u8* structure contains parameters to compute RSNIE, *nonce* pointer contains parameters to compute *SNonce*, and *key* pointer contains parameters to compute *MIC*. Further, *wpa_ie* points to an array of utf8 encoded characters and this is the array that contains RSNIE information. This array can grow up to 256 bytes (according to Section $\oint$ 7.3.2.25 in [9]). Currently, this array contains 22 bytes of information as specified in the *WPA Key Data Length* field.

[2]https://github.com/Anonymous241429/Anonymous

The new codes are shown from Line 4-13 in Listing 2. We utilize the *wpa_ie_sign_S* array to include additional 23 bytes, where first 20 bytes contain the signature, $Sign_S$, the next 2 bytes contain the channel frequency, CF_S, in variable CFS, and the last 1 byte contains the channel number, CN_S, in variable CNS. The signature is computed using *wpa_supp_sign_2_of_4()* function that uses *HMAC-SHA1* function (as described in Section V-D). First, a new pointer *wpa_ie_new* is initialized with *wpa_ie* in line 4. Then CFS, and CNS is computed followed by obtaining $Sign_S$ in line 5-8. These three parameters are then inserted into *wpa_ie_new* array by using *os_memcpy()* function in line 9-13. Thus, a total of 9 lines are newly added into the *wpa_supplicant_send_2_of_4()* function, including the invocations of other functions. This function finally invokes *wpa_eapol_key_send()* function within the return statement of *wpa_supplicant_send_2_of_4()* function.

```
1  int ft_check_msg_2_of_4(struct wpa_eapol_ie_parse *
       kde ...){
2      const u8 *rsn_data = kde.rsn_ie;
3      size_t rsn_data_len=kde.rsn_ie_len;
4      CFA=get_channel_freq_from_config();
5      CNA=get_channel_no_from_config();
6      u8 *wpa_ie_sign_A=ft_compute_sign_2_of_4();
7      wpa_ie_sign_A[20]=CFA; wpa_ie_sign_A[22]=CNA;
8      if(os_memcmp(rsn_data + rsn_data_len - 23,
       wpa_ie_sign_A, 23)==0){wpa_send_eapol();}
9      else{ wpa_sta_disconnect(wpa_auth, sm->addr,
       WLAN_REASON_PREV_AUTH_NOT_VALID);}
10     wpa_printf(MSG_DEBUG, "WPA: Received EAPOL-Key
       from ");
11      ...}
```

Listing 3: Code snippet of the updated WPA authenticator.

Similar to the supplicant code, we have updated *ft_check_msg_2-_of_4()* function in *wpa_auth.c* file in the *ap* subfolder in the *src* directory to update the authenticator code. This function accepts as argument a pointer, i.e., kde, to the structure *wpa_eapol_ie_parse* and the structure contains the additional 23 bytes of RSNIE that is present in the received *M2* as a result of modified supplicant code. Listing 3 shows the portion of this function having our new code. In line 2, RSNIE data is stored in a pointer, i.e., rsn_data, and its length is stored in another variable, i.e., rsn_data_len in Line 3. The next two lines i.e., line 4 and line 5, collects the channel frequency and channel number in the variable in CFA and CNA respectively from the configuration file of *hostapd*. In line 6, we compute the signature $Sign_A$ using our newly added function named *ft_compute_sign_2_of_4()* and stored in an array whose pointer is returned. Line 7 places CFA and CNA in the array pointed by *wpa_ie_sign_A* pointer, making the array to be of size 20 + 2 + 1 = 23 bytes. In line 8, the *if()* statement actually compares the A-OCI at the authenticator using *os_memcmp()* function; if succeeds then *M3* is sent using *wpa_send_eapol()* function, otherwise *deauth* frames is sent using *wpa_sta_disconnect()* function in line 9. Note that the comparison starts from 23^{rd} byte as first 22 bytes contain the existing RSNIE values, and both *wpa_send_eapol()* and *wpa_sta_disconnect()* are existing functions. Thus, a total of 8 lines is added newly in *wpa_auth.c* file.

VI. Experiments and Results

A. The Setup

We use a desktop with Ubuntu 20.04 OS for the majority of the compilation and execution of the updated C codes for the supplicant and authenticator. The updated authenticator, i.e., hostapd, is executed in a Raspberry Pi model 3 B+ to create a realistic Wi-Fi router. The required dependencies are installed before compiling the supplicant and the hostapd.

To compile the code [3] for the supplicant, we use the command `make BINDIR=/usr/sbin LIBDIR=/usr/lib` within the *wpa_suppli-cant* folder. Similarly, we execute the same command within the *hostapd* to compile the code for authenticator. Essentially, both these folders contain their individual *Makefiles*; *src* folder contains the common files used in both the supplicant and the authenticator.

To execute *hostapd*, we have used Alfa adapter (Model: AWUS036-NHA) as Wi-Fi interface in master mode along with a configuration file that contains key-value pairs used in *hostapd*. The important key-value pairs include drivers, channels, BSSID, interfaces, PSK, and key management. Finally, the command takes the form: `./hostapd -i iface_name -c /etc/hostapd/hostapd.confg -d`.

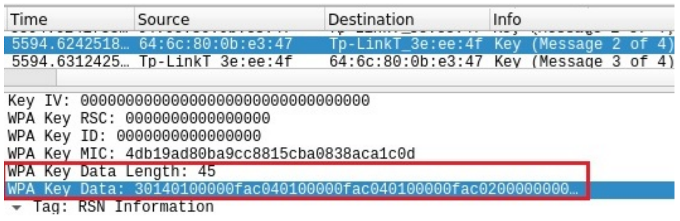
Similarly, to execute the binary of supplicant, we use a D-link (model: DWA-131) adapter as Wi-Fi interface in managed mode along with its configuration file that contains key-value pairs for the parameters like *SSID*, *key management*, and pre-shared key (PSK). So, the command becomes: `wpa_supplicant -i iface_name -c /etc/wpa_supplicant/configuration_file -d` , where “-d” switch indicates the execution is in debug mode, to execute the updated supplicant code. Our solution requires an update in the firmware and we believe that a patch can be issued to already deployed WPA2 enabled devices. Further, we found out that our solution can be easily portable to different newly deployable WPA2 compatible devices like Wi-Fi routers as the binary obtained after compiling the code base is of 6.4 MB and 5.3 MB size for the supplicant and hostapd, respectively.

B. Results of Empirical Evaluation

To check the correctness of the updated codes of supplicant and authenticator, we have captured the Wi-Fi trace and deep inspected the payloads in the Wi-Fi frames. We have configured the authenticator on channel 6 and hence the supplicant is operating on this channel for connection establishment.

Figure 7 depicts a snap of the Wi-Fi trace captured in monitor mode. The highlighted portion indicates the presence of the additional 23 bytes (22+23=45) of A-OCI in *M2* on channel 6. The next packet *M3* in figure 7, but not expanded, in the same channel indicates that the authenticator could validate the signature received in *M2*.

[3]https://github.com/Anonymous241429/Anonymous



```
Time                Source              Destination         Info
5594.6242518…  64:6c:80:0b:e3:47   Tp-LinkT_3e:ee:4f   Key (Message 2 of 4)
5594.6312425…  Tp-LinkT 3e:ee:4f   64:6c:80:0b:e3:47   Key (Message 3 of 4)

Key IV: 00000000000000000000000000000000
WPA Key RSC: 0000000000000000
WPA Key ID: 0000000000000000
WPA Key MIC: 4db19ad80ba9cc8815cba0838aca1c0d
WPA Key Data Length: 45
WPA Key Data: 30140100000fac040100000fac040100000fac0200000000…
 ▾ Tag: RSN Information
```

Fig. 7: Sign. appended with RSNIE in M2.

VII. Formal Analysis of Our Proposed Solution

In this section, we present a detailed formal analysis of our proposed WPA2 modifications using Tamarin Prover [22]. We also compare our analysis with the state-of-the-art formal analysis framework [17] of WPA2 in general and 4-way handshake, in particular.

A. WPA2 and KRACK in Tamarin Prover

The existing work in [17] have used a total of 69 rules to model the operation of WPA2 protocol, in particular the 4-way handshake, and 67 lemmas to formally verify the security properties agaist KRACK on the 4-way handshake. Among others, we consider that the following rule, termed as *Supp_Rcv_M3_repeat*, the most relevant for the purpose of the work in this paper.

```
rule Supp_Rcv_M3_repeat:
  let
    oldPTK = KDF(<PMK1, ANonce1, SNonce1>)
    newPTK = KDF(< PMK, ANonce,  SNonce>)
    ctr = S(ctr_minus_1)
    ctr_m3 = S(ctr_m3_minus_1)
    oldGTKNonce = <N(m), authID2>
    oldGTKData = <oldGTK, oldGTKNonce, $oldIndex>
    m3 = <ctr_m3, senc(newGTKData, newPTK)>
  in
   [SuppState(~suppThreadID,'PTK_INIT_DONE',
<~suppID, ~PMK, ~authThreadID, oldPTK, oldGTKData,
ANonce, ~SNonce, ctr>), In_Supp(<m3,
mic_m3>,~suppThreadID, oldPTK)]
—[SuppReceivesM3Ag(~suppThreadID,~suppID, ~PMK,
~authThreadID, oldPTK, newGTK, ANonce, ~SNonce,
ctr_m3), SuppSeesCtr(~suppThreadID, ~PMK, ctr_m3),
Eq(mic_m3, MIC(newPTK, m3)) ]→
   [SuppState(~suppID, ~PMK, oldPTK, ~suppThreadID,
'PTK_CALC_NEGOTIATING', <~suppID, ~PMK,
~authThreadID, oldPTK, oldGTKData, newGTKData,
ANonce, ~SNonce, ctr_m3>)]
```

The "fact on left" in this rule shows two facts, namely `SuppState()` and `In_Supp()`. `SuppState()` fact indicates that the supplicant is in *PTK_INIT_DONE* state and it has a fresh value of PMK, i.e., PMK, an old value of PTK, i.e., $oldPTK$, an old value of group key data, i.e., $oldGTKData$, an old value of ANonce, i.e., $ANonce$, a fresh value of SNonce, i.e., $SNonce$, and an old value of counter, i.e., ctr. `In_Supp()` indicates that the supplicant receives *M3* from the network with an $oldPTK$. This rule basically states three action facts, i.e., *SuppReceivesM3Ag()*, *SuppSeesCtr()*, and *Eq()*, that leads to the verification of

KRACK. Among others, *SuppReceivesM3Ag()* considers as inputs a fresh value of PMK, an old value of PTK and an old value of ctr_m3 and checks if the timestamp, replay counter, and MIC are fresh in the received *M3*. Similarly, *SuppSeesCtr()* checks if the replay counter, i.e., ctr_m3, is repeated. *Eq()* checks the validity of MIC in received *M3*. The "fact on right" is *SuppState()* and it specifies that the supplicant transits to *PTK_CALC_NEGOTIATING* state with a value tuple containing the values of $oldPTK$, $oldGTKData$, $ANonce$ and $SNonce$. This fact basically correlates another rule that is used to define the specified state.

```
restriction ReplayCounterM3Again:
    "All suppThreadID suppID PMK authThreadID PTK
GTK ANonce SNonce ctr_m3 #t2.
      SuppReceivesM3Ag(suppThreadID, suppID, PMK,
authThreadID, PTK, GTK, ANonce, SNonce, ctr_m3) @ t1
      ==> not Ex #t1. t1 < t2 &
SuppSeesCtr(suppThreadID, PMK, ctr_m3) @ t1"
```

The formal framework in [17] has used certain *restriction*s (a restriction is similar to a lemma in Tamarin Prover) to put in certain constrains. Among others, the above restriction specifies that if an *M3* is received with `ctr_m3` at any time `t2`, then there cannot exist a same `ctr_m3` in a previous *M3* at any time `t1`, i.e., `t1` less than `t2`. In other words, even if *M3* is received multiple times, the replay counters in any of *M3*s cannot be repeated.

B. Evaluating the Updated Framework

We have newly introduced a rule *Supp_Rcv_M1_Snd_M2* into the formal verification framework of WPA2 proposed in [17]. In the existing version of this rule, a supplicant being in *'INIT_R1_SA'* state indicated by *SuppState()* fact receives *M1* indicated by *In_Supp()* fact, obtains a fresh value of *SNonce* and messageID. *SuppReceivesM1()* and *SuppSeesCtr()* action facts indicate that the supplicant receives *M1* and finds a fresh value of PMK and a value of counter. In particular, *SuppReceiveM1()* checks for the timestamp and repetition of *M1*. *SuppSeesCtr()* checks for the replay counter in *M1*. The "fact on right" indicates that the supplicant transits to *'PTK_START'* state and sends *M2* using user-defined *SuppState()* and *OutSuppSndM2()* facts respectively. In addition, *OutSuppSndM2()* fact calculates MIC using another user-defined fact, i.e., *MIC()*, which is passed in *M2*.

To incorporate our proposed solution [4], we have inserted a set of three new parameters within the *OutSuppSnd()* fact in "fact on the right" within the existing *Supp_Rcv_M1_Snd_M2* rule, shown in pink in the following. The first new parameter is computed using *hash()* fact, the last two new parameters are channel frequency (CF) and channel number (CN). Essentially, the *hash()* takes as inputs the *newPTK*, *ANonce*, *SNonce*, *CF*, and *CN* to compute $Sign_S$.

[4]https://github.com/Anonymous241429/Anonymous/tree/main/formal-analysis

```
rule Supp_Rcv_M1_Snd_M2:
  let
     oldPTK = KDF(<PMK1, ANonce1, SNonce1>)
     newPTK = KDF(<~PMK, ANonce, ~SNonce>)
  in
   [SuppState(~suppThreadID, 'INIT_R1_SA', <~suppID,
~PMK ...>, channel), In_Supp(m1, ~suppThreadID,
oldPTK), Fr(~SNonce), Fr(~messageID)]
     —[SuppReceivesM1(~suppThreadID,~suppID...,
        SuppSeesCtr(~suppThreadID, ~PMK, ctr)...]→
   [SuppState(~suppThreadID, 'PTK_START', <~suppID,
~PMK...)>, OutSuppSndM2(<m2, SNonce, MIC(newPTK,
m2)>, hash(newPTK, ANonce, SNonce, CF, CN), CF, CN)]
```

Similarly, we have updated the existing *Auth_Rcv_M2* rule used by authenticator to receive and verify the signature, *CF* and *CN*, shown in the following. The rule newly considers $m2$ as a tuple containing ctr, $SNonce$, CF and CN, and computes $Sign_A$ using *hash()* fact by taking the parameters PTK, $SNonce$, $ANonce$, CF and CN; highlighted in pink. In "fact on left", the *AuthState()* fact indicates that the supplicant is in "PTK_START" state, *In_Auth()* fact indicates the reception of *M2* containing $Sign_S$, CF, CN. In "action fact", the *AuthReceivesM2()* is updated to check the validity of $Sign_S$ received in *M2*. In "fact on right", the *AuthState()* fact indicates that the authenticator transits to "PTK_CALC_NEGOTIATING" state. We have updated this fact to include $Sign_A$ as a new parameter, i.e., $Sign_A$ is passed as a new parameter to the next state. Thereafter, Auth_Check_MIC_M2_Snd_M3 rule sends the *M3* to the supplicant after checking the MIC and signature using *Eq(mic_m2, MIC(newPTK, m2)*, and *Eq(*$Sign_A$*,* $Sign_S$*)* functions. If MIC and signature are valid, then the protocol continues, otherwise fails. Thus, we have augmented the existing formal verification framework to verify our proposed solution.

```
rule Auth_Rcv_M2:
   let
     ctr = S(ctr_minus_1)
     m2 = <ctr, SNonce>
     PTK = KDF(<PMK1, ANonce1, SNonce1>)
     Sign_A = hash(PTK, SNonce, ANonce, CF, CN)
  in
    [ AuthState(~authThreadID, 'PTK_START', <~authID,
~PMK, ~suppThreadID, PTK, ~ANonce, oldSNonce, ctr>),
   In_Auth(<m2, mic_m2>, ~authThreadID, PTK, Sign_A,
CF, CN) ]
–[ AuthReceivesM2(~authThreadID, ~authID, ~PMK,
~suppThreadID, PTK, ~ANonce, SNonce, ctr, Sign_S, CF,
CN)] →
[ AuthState(~authThreadID,
'PTK_CALC_NEGOTIATING', <~authID, ~PMK,
~suppThreadID, PTK, ~ANonce, SNonce, ctr, <m2,
mic_m2>, Sign_A, CF, CN>) ]
```

Finally, we run the automatic proofs of the updated formal verification framework of WPA2, showing the execution of the lemmas in GUI and from the terminal). Table I shows the results of the formal verification of our proposed solution in WPA2. Basically, the existing work in [17] ensures the secrecy of PTK, PMK, and GTK. However, there are no rules or corresponding lemmas to detect the channel-based MiTM or

TABLE I: Result metrics of executing the updated framework.

Property	Object	Entity	Total Rules (69)	Total Lemmas (67)	Time (sec)
$\mathcal{A}$ ([17])	4WH	Supplicant	1	1	3 sec
		Authenticator	1	1	3 sec
$\mathcal{S}$ ([17])	PTK, PMK, GTK	Supplicant	27	6	3.3 sec
		Authenticator	39	10	3.5 sec
Property	Object	Entity	Updated Rules	Updated Lemmas	Time (sec)
$\mathcal{A}$ (Our work)	4WH	Supplicant	7	1	3 sec
		Authenticator	2	7	4.2 sec
$\mathcal{I}$ (our work)	PTK, Channel, SNonce, ANonce	Supplicant	7	1	2 sec
		Authenticator	2	7	4.2 sec
Comparison	$\mathcal{I}$ of channel	$\mathcal{I}$ of Nonce	$\mathcal{A}$ of 4WH	$\mathcal{S}$ of PTK	$\mathcal{S}$ of PMK
[17]	✗	✗	✗	✓	✓
Our Work	✓	✓	✓	✓	✓

$\mathcal{A}$: Authentication, $\mathcal{S}$: Secrecy , $\mathcal{I}$: Integrity, 4WH: 4-Way Handshake

verify the integrity of *M1*, *M2*, *M3* and *M4*. We have updated a total of 9 rules and 8 lemmas in the existing framework of the WPA2 in [17] to formally verify our proposed solution in WPA2 in general, and 4-way handshake in particular. To ensure the authenticity and integrity of the handshake message, we have introduced A-OCI in the form of signature and channel information in the *Supp_Rcv_M1_Snd_M2* rule. We also maintain the secrecy of important keys like PTK. We stop the re-installation of PTK and terminate the handshake session by sending an authenticated *deauth* frame if a channel-based MitM is discovered. The average execution time for each lemma on 10 runs is ~3.3 sec.

VIII. Discussion of Our Proposed Solution

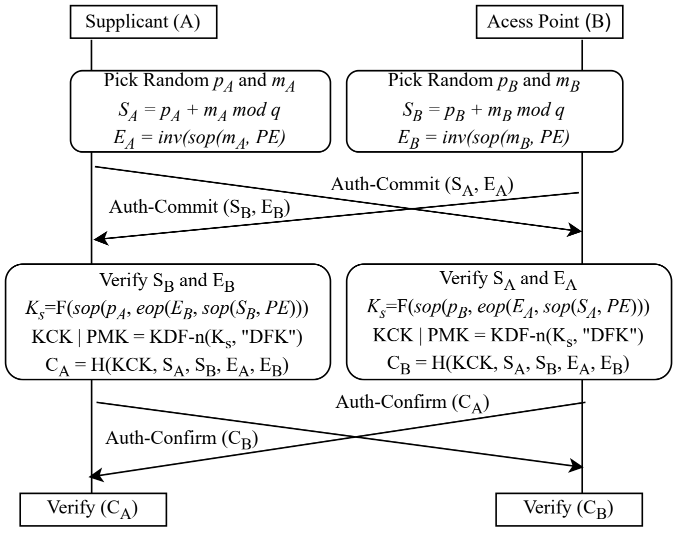
In this section, we discuss the potential of our proposed solution to counter the limitations of Dragonfly handshake in the newly proposed WPA3 protocol suite.

A. Dragonfly Handshake

Figure 8 shows the Simultaneous Authentication of Equals (SAE) algorithm in dragonfly handshake [26] supposed to authenticate the supplicant and authenticator. Unlike 4-way handshake in WPA2, SAE in WPA3 ensures a different PMK for every host even if the Wi-Fi password is same. Basically, the derivation of PMK in SAE involves three broad steps: i) Derivation of the Password Element (PE), ii) Authentication commit message (Auth-commit), and confirm message (Auth-confirm) and iii) Setting the PMK.

Derivation of PE: SAE supports Elliptic Curve Cryptography (ECC) with elliptic curves over a prime field (ECP groups), and Finite Field Cryptography (FFC) with multiplicative groups modulo a prime q (MODP groups). The handshake starts by the supplicant and the authenticator, each deriving a secret element, called the Password Element (PE) which is a $[x, y]$ point on the elliptic curve (EC). The derivation needs a seed that is generated using a function that takes as argument the Wi-Fi password, MAC Addresses of the supplicant and authenticator. This seed is then used in a *hunting and pecking* procedure to generate the PE, where the procedure uses a loop with at least k (=40) iterations.

Auth-Commit and Auth-Confirm messages: In this step, both the supplicant and the authenticator generate two random



Supplicant (A)
Acess Point (B)
Pick Random p_A and m_A
$S_A = p_A + m_A \; mod \; q$
$E_A = inv(sop(m_A, PE)$
Pick Random p_B and m_B
$S_B = p_B + m_B \; mod \; q$
$E_B = inv(sop(m_B, PE)$
Auth-Commit (S_A, E_A)
Auth-Commit (S_B, E_B)
Verify S_B and E_B
K_s=F($sop(p_A, eop(E_B, sop(S_B, PE)))$)
KCK | PMK = KDF-n(K_s, "DFK")
C_A = H(KCK, S_A, S_B, E_A, E_B)
Verify S_A and E_A
K_s=F($sop(p_B, eop(E_A, sop(S_A, PE)))$)
KCK | PMK = KDF-n(K_s, "DFK")
C_B = H(KCK, S_A, S_B, E_A, E_B)
Auth-Confirm (C_A)
Auth-Confirm (C_B)
Verify (C_A)
Verify (C_B)

Fig. 8: Dragon Fly Handshake.

numbers. The supplicant, i.e., A, computes a scalar S_A as $S_A = p_A + m_A mod\ q$, where p_A and m_A are the two random numbers called as *private* and *mask* respectively, and computes an element E_A as $E_A = inv(sop(m_A, PE))$, where $sop(x, y) = Y^x mod\ p$ is called as scalar operation function and $inv()$ is an inverse function. Similarly, the authenticator, i.e., B, computes S_B and E_B as its scalar and element. Both of them then exchange their scalar and element, followed by their verification. If the verification fails, then SAE must be terminated and authentication fails.

Using PE, scalar and element, the supplicant (vis-a-vis the authenticator) derives a shared secret K_s as $K_s = F(sop(p_A, eop(E_B, sop(S_B, PE))))$, where $eop(X, Y) = X * Ymod\ p$ is called as element operation function. To enforce key separation and cryptographic hygiene, the shared secret is stretched into two subkeys using the key define function $KDF\text{-}n(K_s, \text{“}DragonFlyKeyDerivation(DFH)\text{”})$, where $n = 512$ in SAE: i) key confirmation key (KCK) – used in the authentication confirm messages and ii) pairwise master key (PMK) – used to derive PTK for 4-way handshake. Note that KCK here is not the same as that in 4-way handshake in WPA2. Next, a confirmation data element C_A is computed at the supplicant (vis-a-vis the authenticator) using a random function *H()* that takes as input KCK, s_A, s_B, E_A, and E_B, and transmitted to the authenticator (vis-a-vis the supplicant) using *Auth-Confirm* messages. If the verification of C_B (or C_A) at the supplicant fails, then SAE fails; otherwise, it succeeds.

Setting of PMK: PMK generated as $PMK = KDF\text{-}512(keyseed, \text{“}SAE\ KCK\ and\ PMK\text{”}, (s_A + s_B)\ modulo\ r)$, where $keyseed = H(<0> 32, K_s)$, during the execution of SAE and used only after the verification of *Auth-Confirm* messages. Generating the PTK from the PMK, and 4-way handshake is same as that in WPA2.

B. *Protection of Management Frame (PMF)*

IEEE 802.11w [27] is an amendment to IEEE Std 802.11 that includes PMF. Unlike WPA2, WPA3 mandates the use of PMF. The authenticator has to set the Management Frame Protection Capable (MFPC) and Management Frame Protection Required (MFPR) fields in the RSNIE to 1 so that the supplicants are forced to use the functionalities of PMF. Basically, PMF prevents unsolicited deauth, disassoc, and channel switch announcement frames in WPA3.

C. *Operating Channel Validation*

The work in [18] has proposed to use OCI in every Wi-Fi frame that negotiates session key. In this case, the OCI contains 3 bytes to define its type, length, and value, and 3 bytes to indicate channel number and channel frequency; totaling 6 bytes of OCI. Note that the OCI in this proposal does not require any security protection, like authentication or integrity. However, the newly proposed WPA3 [13] specifies the provision of authentication and integrity protection for any frame that includes OCI, e.g., using PMF.

D. *Efficacy of Our Proposed Solution*

Our proposed solution may be considered as a combination of the proposal in [18] and the specification in [13], where we have included 23 bytes of A-OCI as part of the RSNIE in *M2* for the first time in the supplicant. A-OCI is then verified at the authenticator and, on success, it transmits *M3*.

In fact, the work in [21] indicates that WPA3 can suffer from certain attacks, like downgrade and dictionary attacks, due to the provision of *transition mode*. The *transition mode* basically ensures backward compatibility with the existing WPA2. However, as soon as the existing WPA2 is operating the attacks, like KRACK, are possible. Thus, we claim that our proposed solution can be practically viable that is inline with the proposal in [18] and the specification in [13], and has the potential to counter the attacks described in [21] as it does not require to deploy WPA3.

IX. Related Work

Despite its widespread adoption, analyzing the WPA2 protocol suite and discovering its vulnerability is not new. Table II presents a summary of our study of the most relevant existing works that specifically discover attacks on WPA2 and propose certain solution to those attacks. The work in [28] has applied symbolic execution of supplicant code in WPA2 and discovered that certain attacks, like DoS, timing side channel, and memory corruption, are possible due to the vulnerability in "key data" field length that causes the buffer overflow. The work in [18] has proposed to use 6 bytes of OCI in every frame in 4-way handshake so that the attacks like KRACK can be prevented. Consequently, the work in [29] has proposed to use OCI in other management frames, like beacons and channel switch announcements, so that attacks like channel-based MitM can be prevented.

The work in [17] has proposed to build up formal analysis framework of WPA2 so that certain attacks, like KRACK,

TABLE II: Comparison of existing work with our work.

[Metrics] / [Rel. Work]	Code		Attacks	Security		Analsysis	
	MS	MA	MLA	OCI	PMF	PA	FA
[16], [17]	✗	✗	KRACK	✗	✗	✗	✓
[18]	✓	✓	C-MiTM	✓	✗	✗	✗
[29]	✓	✓	FBF	✗	✓	✗	✗
[30]	✗	✗	FragAttacks	✗	✗	✗	✓
[28]	✓	✗	DOS, MC, TSC	✗	✗	✓	✗
Our Proposed Work	✓	✓	Modified KRACK	✓	✓	✓	✓

MS: Modified Supplicant, MA: Modified Authenticator, MLA: MAC Layer Attacks, PA: Packet Analysis, FA: Formal Analysis, FBF: Forged Beacon Frames, MC: Memory Corruption, TSC: Timing Side Channel.

on this protocol suite can be formally demonstrated and root cause for the same can be discovered. In particular, the work has shown that the attack is possible due to the lack of fresh replay counter at the supplicant. Similarly, the work in [16] has theoretically shown that KRACK is also possible due to the lack of fresh PTK or GTK at the supplicant.

The work in [31] has described an attack, called fragment attack, on WPA2, which is due to the design flaw in the frame aggregation functionality that accepts fragments encrypted under different keys. Because of this flaw an adversary can exfitrate supplicant data through vulnerable APs. Consequently, the work in [30] has build up a formal analysis framework in Tamarin prover and theoretically shows that fragment attack is possible in WPA2 as well due to certain shortcomings in its specification i.e., unprotected MAC header fields, rather than an implementation.

Comparing the state-of-the-art, our proposed solution has the potential to prevent the attacks like KRACK and channel-based MitM, and does not require migration to WPA3. Overall, none of the existing works has looked at any possibility of updating the existing WPA2 code base and demonstrated any empirical investigation of the efficacy of their proposed solution.

X. Conclusion and Future Work

In this paper, we have addressed the problem of *KRACK* on WPA2 by updating the parameters exchanged during a 4-way handshake. Specifically, we have proposed to include authenticated operating channel information, i.e., A-OCI, in certain messages in a 4-way handshake in WPA2, and this essentially flunks the channel-based MiTM attack that is one of the precondition to launch KRACK. A-OCI includes three pieces of information (20 bytes of signature, 2 bytes of channel frequency, and 1 byte of channel number) totaling 23 bytes that are included as part of RSNIE in *M2*. The existing WPA2 standard have the provisioning to include a maximum of 256 bytes of information and currently 22 bytes are used to transfer RSN information. We propose to consider 23 more bytes to include A-OCI by the supplicant. On receiving *M2*, the authenticator checks A-OCI; if succeed, then it sends out *M3*, otherwise, it sends out deauthentication frame to the supplicant leading to an unsuccessful handshake. We have implemented our proposed solution into the existing C code base of WPA2, specifically in *wpa_supplicant_send_2_of_4()* in “wpa.c" file and *ft_check_msg_2_of_4()* in “wpa_auth.c" file for the supplicant and authenticator respectively. An authenticator verifies the authenticity of the signature sent in *M2* before processing the other parameters in the packet. We have successfully rebuilt the entire code base and executed the code in our customized experimental setup. We have validated our proposed solution in two fronts - i) empirically by capturing Wi-Fi traffic, and ii) formally by integrating rules and lemmas in Tamarin Prover. Using the traffic of WPA2 4-way handshake, We have shown that if *M2* carries a valid 43 bytes of RSNIE then the handshake successfully completes; alternately, deauthentication frames are seen when invalid RSNIE is present in *M2*. We have newly added 9 rules and 8 lemmas in the existing Tamarin Prover framework to implement our proposed solution. Comparing with the state-of-the-art, our proposed solution in the framework satisfies the integrity of channel, Nonce and Authentication, in addition to the secrecy of PTK and PMK. As part of our future works, we plan to integrate our proposed solution in the other versions of WPA2 and the other existing code bases.

REFERENCES

[1] A. Agrawal and R. R. Maiti, “Guarding the wi-fi 4-way handshake against channel-based mitm: A case study on krack attack,” in *Proceedings of the 14th ACM CODASPY*, 2024, p. 147–149.

[2] Cisco. (Last Updated: March 2020) Cisco annual internet report (2018–2023) white paper. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[3] Gartner. (Last Updated: Sep 2022) Capturing value from next-generation wireless. [Online]. Available: https://www.gartner.com/en/articles/capturing-value-from-next-generation-wireless

[4] F. M. Insights. (Last Updated: July 2022) Cwireless access point market outlook (2022-2032). [Online]. Available: https://www.futuremarketinsights.com/reports/wireless-access-point-market

[5] M. Vanhoef and F. Piessens, “Key reinstallation attacks: Forcing nonce reuse in WPA2,” in *Proc. of the 24th ACM CCS*, 2017.

[6] V. Mathy and P. Frank, “Predicting, decrypting, and abusing WPA2/802.11 group keys,” in *25th USENIX Security 16*, 2016.

[7] A. Cassola, W. Robertson, E. Kirda, and G. Noubir, “A practical, targeted, and stealthy attack against wpa enterprise authentication,” in *Proc. of NDSS*, 2013.

[8] E. N. Lorente, C. Meijer, and R. Verdult, “Scrutinizing wpa2 password generating algorithms in wireless routers,” in *USENIX WOOT*, 2015.

[9] *IEEE Std 802.11i. 2004. Amendment 6: Medium Access Control (MAC) Security Enhancements.*, pp. 1–190, 2004.

[10] “Ieee std 802.11. 2016. wireless lan medium access control (mac) and physical layer (phy) spec.” pp. 1–3534, 2016.

[11] M. Thankappan, H. Rifà-Pous, and C. Garrigues, “Multi-channel man-in-the-middle attacks against protected wi-fi networks: A state of the art review,” *Expert Systems with Applications*, 2022.

[12] M. Vanhoef and F. Piessens, “Advanced wi-fi attacks using commodity hardware,” in *Proc. of ACSAC*, 2014.

[13] W.-F. Alliance. (Dec 2020) Wi-fi certified wpa december 2020 update brings new protections against active attacks: Operating channel validation and beacon protection. [Online]. Available: https://www.wi-fi.org/beacon/thomas-derham-nehru-bhandaru/wi-fi-certified-wpa3-december-2020-update-brings-new

[14] A. Agrawal, U. Chatterjee, and R. R. Maiti, “Checkshake: Passively detecting anomaly in wi-fi security handshake using gradient boosting based ensemble learning,” *IEEE Transactions TDSC*, 2023.

[15] A. Anand, C. Urbi, and M. R. Rajib, “ktracker: Passively tracking krack using ml model,” in *Proc. of the 12th ACM CODASPY*, 2022.

[16] R. R. Singh, J. Moreira, T. Chothia, and M. D. Ryan, “Modelling of 802.11 4-way handshake attacks and analysis of security properties,” in *Security and Trust Management*, 2020.

[17] C. Cremers, B. Kiesl, and N. Medinger, “A formal analysis of IEEE 802.11’s WPA2: Countering the kracks caused by cracking the counters,” in *29th USENIX Security Symposium*, 2020.

[18] M. Vanhoef, N. Bhandaru, T. Derham, I. Ouzieli, and F. Piessens, “Operating channel validation: Preventing multi-channel man-in-the-middle attacks against protected wi-fi networks,” in *ACM Wisec*, 2018.

[19] W.-F. Alliance. (Last Updated: oct 2024) Wpa3 specification version 3.4.

[20] C. Hoffman. (Oct 2018) What is wpa3, and when will i get it on my wi-fi? [Online]. Available: https://www.howtogeek.com/339765/what-is-wpa3-and-when-will-i-get-it-on-my-wi-fi

[21] M. Vanhoef and E. Ronen, “Dragonblood: Analyzing the dragonfly handshake of wpa3 and eap-pwd,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 517–533.

[22] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The tamarin prover for the symbolic analysis of security protocols,” in *Computer Aided Verification*, Berlin, Heidelberg, 2013.

[23] M. Vanhoef. (Last Updated: 6 March 2018) krackattacks-poc-zerokey. [Online]. Available: https://github.com/vanhoefm/krackattacks-poc-zerokey

[24] D. Dolev and A. Yao, “On the security of public key protocols,” *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[25] J. Malinen. (2019) Developers’ doc for wpa_supplicant and hostapd. [Online]. Available: https://w1.fi/wpa_supplicant/devel/files.html

[26] D. Harkins. (Last Updated: Nov 2015) Dragonfly key exchange (rfc: 7664). [Online]. Available: https://www.rfc-editor.org/rfc/rfc7664.html

[27] IEEE, “Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 4: Protected management frames,” *IEEE Std 802.11w-2009*, pp. 1–111, 2009.

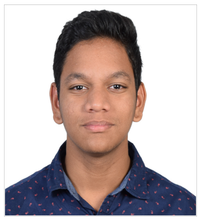
[28] M. Vanhoef and F. Piessens, “Symbolic execution of security protocol implementations: Handling cryptographic primitives,” in *12th USENIX WOOT*, 2018.

[29] M. Vanhoef, P. Adhikari, and C. Pöpper, “Protecting wi-fi beacons from outsider forgeries,” in *Proc of ACM Wisec*, 2020.

[30] Z. Shen, I. Karim, and E. Bertino, “Segment-based formal verification of wifi fragmentation and power save mode,” in *ACM Asia CCS*, 2024.

[31] M. Vanhoef, “Fragment and forge: Breaking Wi-Fi through frame aggregation and fragmentation,” in *30th USENIX Security*, 2021.

Anand Agrawal is working as a PhD student at BITS Pilani, Hyderabad campus. He was associated with iTRUST Lab in SUTD, NCL Lab of NUS, and MoMA Lab of NYUAD. His research interest lies in analyzing the Wi-Fi traffic and understanding the WPA/WPA2 security standard. Email ID: p20200434@hyderabad.bits-pilani.ac.in.



Ashlin Chirakkal is currently in the final year of his computer science bachelor’s degree at BITS Pilani, Hyderabad campus. He looked at the formal analysis of the WPA2 Wi-Fi security protocol using tools such as AVISPA and Tamarin Prover. Currently, his main interests include cybersecurity and software development. Email ID: ashlinchirakkal@gmail.com

Urbi Chatterjee is working as an Assistant Professor in the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur since March, 2021. She has completed her Ph.D. in the CSE department, IIT Kharagpur. Her area of research is Hardware Security.

Rajib Ranjan Maiti is currently an Assistant Professor in CSIS, BITS Pilani, Hyderabad campus India. He has done his PhD in CSE at IIT Kharagpur, India. His work has been published in Transaction on Mobile Computing, Esorics, WiSec, and AsiaCCS. He is currently executing two sponsored projects related to cybersecurity, one funded by SERB, DST, India, and the other funded by Axiado, India.