Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd

Mathy Vanhoef
New York University Abu Dhabi
Mathy.Vanhoef@nyu.edu

Eyal Ronen
Tel Aviv University and KU Leuven
eyal.ronen@cs.tau.ac.il

***Abstract*—The WPA3 certification aims to secure home networks, while EAP-pwd is used by certain enterprise Wi-Fi networks to authenticate users. Both use the Dragonfly handshake to provide forward secrecy and resistance to dictionary attacks. In this paper, we systematically evaluate Dragonfly's security. First, we audit implementations, and present timing leaks and authentication bypasses in EAP-pwd and WPA3 daemons. We then study Dragonfly's design and discuss downgrade and denial-of-service attacks. Our next and main results are side-channel attacks against Dragonfly's password encoding method (e.g. hash-to-curve). We believe that these side-channel leaks are inherent to Dragonfly. For example, after our initial disclosure, patched software was still affected by a novel side-channel leak. We also analyze the complexity of using the leaked information to brute-force the password. For instance, brute-forcing a dictionary of size 10^{10} requires less than $1 in Amazon EC2 instances. These results are also of general interest due to ongoing standardization efforts on Dragonfly as a TLS handshake, Password-Authenticated Key Exchanges (PAKEs), and hash-to-curve. Finally, we discuss backwards-compatible defenses, and propose protocol fixes that prevent attacks. Our work resulted in a new draft of the protocols incorporating our proposed design changes.**

I. Introduction

After the disclosure of key reinstallation attacks (KRACKs) in WPA2 [1], the Wi-Fi Alliance released WPA3 as the successor of WPA2 [2]. It is important to remark that WPA3 does not define new protocols. Instead, it is a certification that defines which existing protocols a device must support. Simplified, WPA3 mandates support of the Dragonfly handshake, and its only new feature is a transition mode where WPA2 and WPA3 are simultaneously supported for backwards-compatibility. Unfortunately, the security guarantees of WPA3 and its Dragonfly handshake are unclear. For example, a close variant of Dragonfly received significant criticism while being standardized [3]–[5], while a different variant of Dragonfly has a formal security proof [6]. These contradictory claims raise the question of whether Dragonfly is secure in practice.

We systematically evaluate the security of Dragonfly and its usage in WPA3 and EAP-pwd. The EAP-pwd protocol is used by some enterprise Wi-Fi networks to authenticate users, while WPA3 is used in personal Wi-Fi networks. Both protocols rely on Dragonfly to provide forward secrecy and protection against offline dictionary attacks. The Dragonfly variant used in WPA3 is also known as Simultaneous Authentication of Equals (SAE). To evaluate the security of Dragonfly, we audit and reverse engineer implementations, study risks specific to WPA3, analyze standards for timing and cache side-channels, use tools to detect implementation-specific side-channels, and explain how the side-channel leaks enable offline brute-force attacks. We confirmed all results against proprietary and open source implementations of WPA3 and EAP-pwd. Additionally, we released open source tools so users can check if implementations are vulnerable, and so results are easier to replicate [7].

To inspect WPA3 and WPA-pwd implementations, we wrote a test harness to see if edge cases in the Dragonfly handshake are properly handled. This revealed authentication bypasses and reflection attacks in multiple implementations. We also audited and reverse engineered firmware, revealing additional vulnerabilities such as known and novel side-channel leaks.

We then study the usage of Dragonfly in WPA3. Here we bypass WPA3-SAE's denial-of-service defense, and overload the CPU of a high-end Access Point (AP). Surprisingly, the cause of SAE's high overhead are defenses against known timing side-channels. Second, we demonstrate a downgrade and dictionary attack against WPA3 when it is operating in transition mode, and we discover a downgrade attack against SAE itself. Additionally, we present implementation-specific downgrade and dictionary attacks that work even when the victim uses WPA3-only networks.

Our main results are side-channel attacks against Dragonfly. These were found by analyzing specifications, and using tools to detect implementation-specific leaks. Our attacks consist of timing side-channels and new micro-architectural cache side-channels. These attacks apply to WPA3 and EAP-pwd, leak information about the password, and work even against implementations that have defenses against known side-channel leaks. As a result, our findings are also of more general interest since they affect ongoing standardization efforts in PAKE and hash-to-element algorithms [8]–[10]. We also show that these timing and cache leaks enable offline brute-force attacks. For instance, searching through a dictionary of size 10^{10}, which is larger than all available password dumps, can be done for less than $1 in GPU-enabled Amazon EC2 instances.

We believe more openness while creating WPA3 and Dragonfly could have prevented most attacks. For example, excluding the MAC addresses from Dragonfly's password encoding method, or using constant-time algorithms, would have mitigated most attacks.

In collaboration with the Wi-Fi Alliance and CERT/CC we notified affected vendors, and we also helped write patches to prevent most attacks. Affected vendors and allocated Common Vulnerabilities and Exposures (CVE) IDs can be found at [11]. During this coordinate disclosure, the Wi-Fi Alliance privately created recommendations to securely implement WPA3, in

DOI 10.1109/SP40000.2020.00031

which they claim Brainpool curves are safe to use [12]. However, using Brainpool curves requires extra defenses, and we found that patched WPA3 implementations were still vulnerable when using Brainpool curves. This resulted in a second disclosure round, and highlights the difficulty of implementing Dragonfly without side-channels leaks. It also affirms that security protocols should be designed to be efficient and easy to implement securely. Fortunately, our proposed protocol changes that prevent most attacks are being incorporated into WPA3 and EAP-pwd [13]–[15].

Summarized, our contributions are:

- We audit EAP-pwd and SAE implementations, and find several vulnerabilities that range from novel side-channel leaks to authentication bypasses (Section III).
- We present denial-of-service, downgrade, and dictionary attacks against WPA3 and SAE (Section IV).
- We discuss known and novel timing side-channels in SAE and EAP-pwd, and abuse them in practice (Section V).
- We exploit cache side-channels in SAE (Section VI).
- We show how our side-channel leaks enable offline brute-force attacks (Section VII).

Finally, we discuss related work in Section VIII, and we give concluding remarks and recommendations in Section IX.

II. Background

In this section we introduce Dragonfly as used in WPA3 and EAP-pwd [2], [16] and cover parts of the 802.11 standard [17].

A. *The Dragonfly Handshake*

The Dragonfly handshake prevents offline dictionary attacks and provides forward secrecy [18]. It is a Password Authenticated Key Exchange (PAKE), meaning it turns a password into a high-entropy key. It was designed by Harkins in 2008, and is used in practice by both WPA3 and EAP-pwd [2], [16]. Variants are also used in TLS-PWD and IKE-PSK [19]–[21]. However, only 802.11 and WPA3 officially adopted Dragonfly. None of the other RFCs that define a Dragonfly variant are standards-track RFCs, meaning they are not endorsed by e.g. the Internet Engineering Task Force (IETF).

Dragonfly supports Elliptic Curve Cryptography (ECC) with elliptic curves over a prime field (ECP groups), and Finite Field Cryptography (FFC) with multiplicative groups modulo a prime (MODP groups). We use G for the generator of a group and q for the order of G. Lowercase letters denote scalars, and uppercase letters denote group elements. Elliptic curves are defined over the equation $y^2 = x^3 + ax + b \bmod p$ where p is a prime and a, b, and p depend on the curve being used. We use $\mathcal{O}$ to represent the point at infinity. For both MODP and ECP groups, all calculations are done modulo their prime p.

Password Derivation. Before initiating the Dragonfly handshake, the pre-shared password is converted to a group element using a hash-to-element method. The hash-to-element method for MODP groups is called hash-to-group, and the one for elliptic curves is called hash-to-curve. In both algorithms the password element P is generated using a try-and-increment

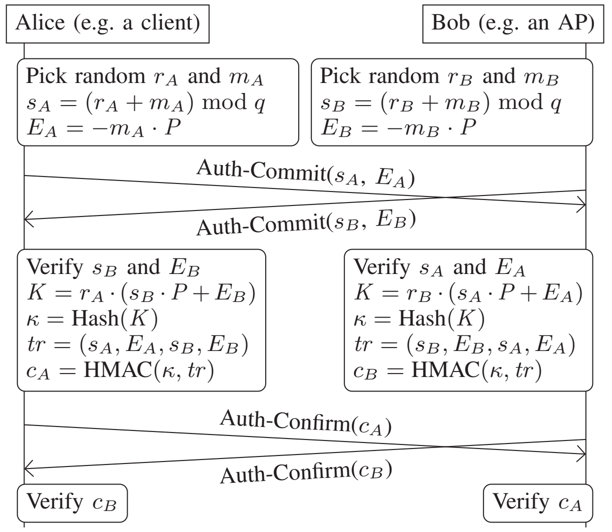
Listing 1: Hash-to-curve method. If token is `None` the SAE variant is executed, otherwise it executes the EAP-pwd variant.

```
1  def hash_to_curve(password, id1, id2, token=None):
2    found, counter, base = False, 0, password
3    label = "EAP-pwd" if token else "SAE"
4    k = 0 if token else 40
5    while counter < k or not found:
6      counter += 1
7      seed = Hash(token, id1, id2, base, counter)
8      value = KDF(seed, label + " Hunting and Pecking", p)
9      if value >= p: continue
10     if is_quadratic_residue(value^3 + a * value + b, p):
11       if not found:
12         x, save, found = value, seed, True
13         base = random()
14
15   y = sqrt(x^3 + a * x + b) mod p
16   P = (x, y) if LSB(save) == LSB(y) else (x, p - y)
17   return P
```

strategy, where in each iteration a hash is first computed over the password, an incremental counter, and the peer's identities (IDs). With EAP-pwd, the input of the hash also includes a random token generated by the server. The hash-to-curve variant uses the hash output as the x coordinate of a point, and it then checks if there is a solution for y over the equation $y^2 = x^3 + ax + b \bmod p$ (see Listing 1). If a solution exists, the password element is the point (x, y). Otherwise, the counter is incremented, and another attempt is made to find a solution for y using the new x value. We discuss the hash-to-group method in Section V, and unless otherwise noted, we assume the elliptic curve variant is used since it is more widely deployed.

To mitigate timing leaks, WPA3-SAE executes the while loop k times no matter when P is found. However, no value for k is suggested, and EAP-pwd does not even have this defense. Other variants of Dragonfly use $k = 40$ [20], [22], and several SAE implementation also use this value (see Section III-D). In the extra iterations, operations are based on a random password. Information may also leak when checking if there is a solution for y in line 10. The EAP-pwd standard does not realize this, and directly tries to calculate y, which may take longer when there is a solution. In contrast, WPA3 recommends to first check if there is a solution using the Legendre function before calculating y. Unfortunately, even a Legendre function can leak info if not carefully implemented [23]. To prevent this, an update to 802.11 recommends Quadratic Residue (QR) blinding [24], [25]. With this defense, a random number is first generated, squared, and multiplied to the number being checked. The result is then multiplied with a random quadratic (non-)residue, before executing the Legendre function [17, §12.4.4.2.2].

Commit and Confirm Phase. The Dragonfly protocol itself consists of a commit and confirm phase. Figure 1 illustrates these phases, and the corresponding curve operations. Both peers can initiate the handshake concurrently, which may happen in mesh networks after a connection loss. However, in infrastructure WPA3 networks the client always sends the first commit, while with EAP-pwd the RADIUS authentication



Alice (e.g. a client)
Bob (e.g. an AP)
Pick random r_A and m_A
$s_A = (r_A + m_A) \bmod q$
$E_A = -m_A \cdot P$
Pick random r_B and m_B
$s_B = (r_B + m_B) \bmod q$
$E_B = -m_B \cdot P$
Auth-Commit(s_A, E_A)
Auth-Commit(s_B, E_B)
Verify s_B and E_B
$K = r_A \cdot (s_B \cdot P + E_B)$
$\kappa = \text{Hash}(K)$
$tr = (s_A, E_A, s_B, E_B)$
$c_A = \text{HMAC}(\kappa, tr)$
Verify s_A and E_A
$K = r_B \cdot (s_A \cdot P + E_A)$
$\kappa = \text{Hash}(K)$
$tr = (s_B, E_B, s_A, E_A)$
$c_B = \text{HMAC}(\kappa, tr)$
Auth-Confirm(c_A)
Auth-Confirm(c_B)
Verify c_B
Verify c_A

Fig. 1: WPA3's SAE handshake. Both stations can simultaneously initiate the handshake, hence the crossed arrows. We assume elliptic curves are used, though similar operations are performed when using multiplicative groups.

server always sends the first commit frame. In this paper we consider the RADIUS server and AP to be the same entity.

In the commit phase, each peer picks two random numbers $r_i, m_i \in [2, q[$ such that $r_i + m_i \in [2, q[$ (see Fig. 1). They then calculate $E_i = -m_i \cdot P$ and send s_i and E_i to each other using a commit frame. On reception of these values, each peer verifies that the received s_i is within the range $[1, q[$, and that E_i is a valid point on the curve [17, §12.4.5.4]. If one of these checks fails, the handshake is aborted. Forward secrecy is provided since deriving m_i given P and E_i is hard, i.e., it relies on the elliptic curve discrete logarithm problem.

In the confirm phase, each peer calculates the secret point K. The x-coordinate of this point is processed using a hash function to derive the key κ, and a HMAC is computed over the handshake summary tr with as key κ. The result of this hash, denoted by c_i, is sent to the other peer in a confirm frame. On reception of c_i, the receiver verifies its value. It if differs from the expected value, the confirm frame is ignored. Otherwise the handshake succeeds and the negotiated key is κ.

B. Dragonfly in WPA3 and EAP-pwd

Dragonfly is mainly used in personal WPA3 networks, and in enterprise WPA2/3 networks that authenticate clients using EAP-pwd. In other words, only Wi-Fi networks employ Dragonfly, and our attacks apply to both network configurations.

The Dragonfly variant used in personal WPA3 networks is called Simultaneous Authentication of Equals (SAE). The SAE handshake was added to 802.11 in 2011 [26], and requires that the pre-shared key is stored in plaintext. In the hash-to-element algorithm, the identities of both peers are their MAC addresses. After executing SAE, the negotiated high-entropy key is used in a 4-way handshake to derive a fresh session key. Although WPA3 still uses WPA2's 4-way handshake, it is not vulnerable to dictionary attacks because the key generated by SAE has much higher entropy than an ordinary password. The SAE handshake explicitly supports mesh networks, by allowing both peers to initiate the handshake concurrently.

Enterprise Wi-Fi networks can use various EAP-based authentication methods. We focus on EAP-pwd, which was defined in 2010 and is based on Dragonfly [16]. It allows devices to store passwords in plaintext or in hashed forms. Note that the difference between enterprise WPA2 and WPA3 networks, is that with WPA3 all ciphers must offer at least 192 bits of security (e.g. at least 384-bit elliptic curves must be used). In EAP-pwd, the AP initiates the handshake, and commit and confirm frames are encapsulated in 802.1X frames.

C. WPA3-SAE Transition Mode

To accommodate devices that do not support WPA3, a network can operate in a transition mode where WPA2 and WPA3 are simultaneously supported using the same password. In this mode the AP advertises Management Frame Protection (MFP) as optional. Older clients then connect using WPA2 without MFP, while modern clients connect using WPA3's SAE with MFP enabled. The only requirement placed on WPA3 clients is that they must use MFP, even though it is advertised as optional. The WPA3 certification does not discuss the security of transition mode [2]. Nevertheless, one would expect that if all devices support WPA3, it is as secure as normal WPA3. Unfortunately, in Section IV-A we show this is not the case.

III. A Systematic Analysis of Dragonfly

In this section we describe our methodology, and we evaluate the security of EAP-pwd and WPA3-SAE implementations.

A. Methodology

We first evaluate the security of EAP-pwd and SAE implementations. This is done by creating a black-box test harness to check if corner cases in the specification are properly handled. We also audit open source implementations, and reverse engineer closed ones. This revealed several vulnerabilities, where the most severe ones can be abused to bypass authentication.

In a second step we inspect the WPA3-SAE specification, and study the risks of deploying Dragonfly in Wi-Fi networks. This uncovered two downgrade attacks, and a denial-of-service attack caused by Dragonfly's high overhead (see Section IV).

Most importantly, we also inspect EAP-pwd and SAE for new side-channel leaks. We do this by analyzing their specifications for design flaws, and by using MicroWalk to detect side-channel leaks in implementations [27]. This revealed several novel attacks (see Section V and VI).

B. Threat Models

All our attacks are against Wi-Fi clients or APs, and several apply to both. This means we must always be within range of the target to perform our attacks. Unless otherwise noted, when the target is an AP, we masquerade as a client and send (malicious) frames to the AP. When the target is a client, we create a rogue AP that clones a network saved by the victim,

TABLE I: EAP-pwd (top) and SAE (bottom) tools that accept an invalid scalar/element (2nd column), do not detect reflection attacks (3rd column), or have known timing leaks (k-columns). In Section V we cover novel timing attacks.

Software	Invalid	Reflect	$k = 0$	$k \leq 4$
FreeRADIUS	●	●	●	●
Radiator	●	●	●	●
hostapd 2.0-2.7	●	●	2.0-2.6	2.0-2.6
wpa_supplicant 2.0-2.7	●	—	2.0-2.6	2.0-2.6
Aruba client	●	—	●	●
iwd 0.2-0.16	●	—	0.2-0.14	0.2-0.14
hostapd 2.1-2.7	○	—	○	2.1-2.4
wpa_supplicant 2.1-2.7	○	2.1-2.4	○	2.1-2.4
iwd 0.7-0.16	●	○	○	○

meaning the targeted client will automatically connect to it. We can even force the victim into connecting to our rogue AP by using a higher signal strength than the legitimate AP, or by jamming the legitimate AP [28].

In our side-channel attacks of Section V and VI, we need to perform numerous handshakes with different MAC addresses. When targeting an AP this can be done by masquerading as different clients. When targeting a client, we can set up multiple rogue APs that each use a different MAC address but all advertise the same network. One obstacle when attacking clients is that most will temporarily blacklist the rogue AP if the handshake repeatedly fails. However, all clients we tested blacklist only a specific MAC address. Hence, if a handshake fails, the targeted client will still connect to a rogue AP with a different MAC address.

In our cache attacks of Section VI, we assume the adversary can also monitor cache access patterns on the target's machine. This can only be done when running code on the same physical hardware (e.g. in a different process or virtual machine). This threat model is similar to the one in recent attacks against TLS [29]–[31]. However, we can also target clients, and can run our attack from any unprivileged user-mode process (e.g. Android application). It is even possible to perform such attacks by injecting JavaScript in older browsers [32].

C. Test Harness and Discovered Flaws

We created black-box tests for EAP-pwd and SAE to verify whether the following three checks are implemented. First, the receiver of a commit frame must check that the scalar is in the range $[2, q[$, and must check that the element is a member of the group (e.g. that point E_A is on the curve). Additionally, the initiator must detect reflection attacks where the peer reflects the scalar and element. Table I lists the tested implementations and discovered attacks. For hostapd and wpa_supplicant we tested version 2.0 up to 2.7. Note that support for SAE was added in version 2.1 of these tools. The Aruba client is a driver that adds support for EAP-pwd to Windows.

Surprisingly, none of the EAP-pwd implementations validate the received scalar or element. We abuse this in an invalid curve attack, where the adversary sends a point that is on an invalid curve with a very small number of elements, making the key K guessable [33], [34]. The attack works both against clients and APs. To attack an AP (i.e. RADIUS server) we send a commit frame with an invalid point, and wait for the server's confirm frame. We then brute-force the key κ by guessing its value, and verifying a guess by reconstructing the server's confirm frame and comparing it with the received one. To attack a client, we send an invalid point that causes K to have only three possible values, namely the point at infinity and two points with the same x-coordinate. We then guess the key κ and send a confirm frame. If the guess was correct, which has a 66% chance since κ is only based on the x-coordinate of K, the victim replies with a confirm frame. In both attacks, we bypass authentication. We confirmed this attack against all client and server-side implementations of EAP-pwd.

All server-side EAP-pwd implementations were vulnerable to reflection attacks. This attack allows the adversary to authenticate as the victim, but does not reveal the session key κ. Note that clients cannot be vulnerable to reflection attacks, because they do not initiate the EAP-pwd handshake.

For SAE, wpa_supplicant 2.1 to 2.4 are affected by reflection attacks. This can be abused to set up a rogue AP, and complete the SAE handshake, though traffic cannot be intercepted since κ is unknown. We also found that iwd did not verify the received scalar. To exploit this, we send a scalar s_B equal to the order of the elliptic curve such that $s_B \cdot P$ equals the point at infinity $\mathcal{O}$. We then construct a valid point E_B such that $\mathcal{O} + E_B$, when computed by iwd, equals the point at infinity $\mathcal{O}$, causing κ to be zero. Since iwd can be forced into using this curve, the bug is exploitable, and allows an attacker to act as a rogue AP and intercept the client's traffic.

D. Code Audits and Reverse Engineering

We also checked for known timing leaks by auditing open source implementations and reverse engineering closed ones.

Known timing leaks. The initial specification of EAP-pwd and SAE did not perform extra iterations in the hash-to-curve method, meaning they stop once a solution for y is found [16], [26]. Only SAE got updated to perform extra iterations [35]. The CFRG proposed this defense, and advised to always perform 40 iterations based on a back-of-the-envelope calculation by Igoe [22]. Information may also leak when checking if there is a solution for y in line 10 of Listing 1. The EAP-pwd standard has no defenses against this (recall Section II-A), while 802.11 adopted a blinded quadratic residue test in an update to the standard [17, §12.4.4.2.2].

Since Dragonfly has a history of side-channels leaks, we evaluate which defenses are deployed in practice. One may also wonder why Dragonfly did not use an alternative design that avoids side-channel leaks. For example, CFRG members suggested to exclude the peer's identities from the hash-to-element method [36]–[40]. With this change, the password element can be calculated offline, reducing the impact of side-channel leaks. Unfortunately, the designers dismissed this advice, meaning the above defenses are essential in practice.

Results. Code audits revealed that version 2.1 up to 2.4 of hostapd and wpa_supplicant use $k = 4$ for SAE, while newer versions use $k = 40$. However, this update was not considered security-critical, meaning these old versions remain vulnerable to timing attacks. All versions of iwd use $k = 20$ for SAE, making timing attacks hard but in theory possible. No extra iterations are performed in EAP-pwd's hash-to-curve of FreeRADIUS, Radiator, Aruba, iwd 0.16 and lower, and in version 2.0 to 2.6 of hostapd and wpa_supplicant. This can be abused against clients to recover the point P, but is non-trivial to exploit against APs (see Section V-D).

In FreeRADIUS, the hash-to-curve algorithm of EAP-pwd aborts when 11 or more iterations are needed. This means that one out of every 2048 handshakes fails, which reveals that the password element was not found in the first 10 iterations. We successfully abused this side-channel leak to brute-force the password using the techniques of Section VII.

After reversing Aruba's EAP-pwd client for Windows, we found that it generated random numbers based on the current system time. This allows an adversary to predict m_A and recover the password from E_A. Aruba's client also aborts the hash-to-curve algorithm when more than 30 iterations are needed. This means one in every billion handshakes fails, in which case enough info is leaked to brute-force the password.

In addition to the devices in Table I, we reverse engineered two firmware images of Cypress. These firmware images are run on Wi-Fi radios, and allow devices to offload the SAE handshake to the Wi-Fi radio. Interestingly, they execute at minimum only $k = 8$ iterations, which is considered insufficient to prevent information leaks. We conjecture this was done because always executing 40 iterations is too costly for these lightweight radios (see Section IV).

IV. Wi-Fi-Centric Attacks

In this section we present downgrade and dictionary attacks against WPA3-SAE. We also compare Dragonfly's high overhead with other hash-to-curve methods, and abuse its high overhead by defeating SAE's denial-of-service (DoS) defense.

A. Downgrade & Dictionary Attacks

1) Attacking WPA3 Transition Mode. In the transition mode of WPA3, an AP accepts connections using WPA3-SAE and WPA2 with the same password. This provides compatibility with older clients, while WPA2's 4-way handshake detects downgrade attacks. That is, if an adversary modifies beacons to trick the client into thinking the AP only supports WPA2, the client will detect this downgrade attack during WPA2's 4-way handshake. This is because the 4-way handshake contains an authenticated RSNE element listing the AP's supported cipher suites, allowing a client to detect if an adversary forged the RSNEs in beacons. This means WPA3 provides forward secrecy, even when using the transition mode of WPA3-SAE.

The problem is that, although downgrade attacks are detected by WPA2's 4-way handshake, by that point an adversary has captured enough data to perform a dictionary attack. This is because capturing a single authenticated 4-way handshake message to carry out a dictionary attack [41]. Moreover, a man-in-the-middle position is not needed to perform the attack.

TABLE II: Clients affected by downgrade attacks when the AP operates in transition mode (column Trans) or in WPA3-only mode (column 3-Only). On the last 3 devices the network must be configured manually, while on other devices the network is selected from a list of nearby ones.

Device	Software	Trans	3-Only
MSI GE60	iwd v0.14	●	●
Latitude 7490	Net. Manager 1.17	○	○
Google Pixel 3	qpp1.190205.018.b4	○	○
Galaxy S10	g975usqu1asba	●	●
AP of vendor A	Firmware 10.20.0168	●	○
RaspberryPi 1 b+	OpenWRT r9576	●	○
MSI GE60	wpa_supplicant 2.7	●	○

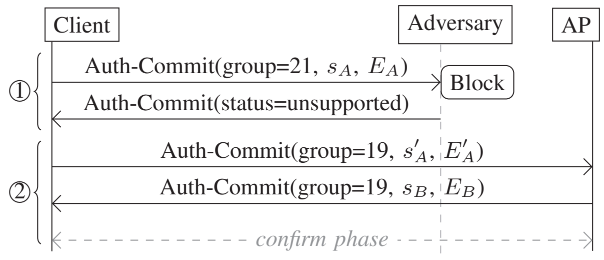
The only requirements are that we know the SSID of the network, and that we are close to a client. If these conditions are met, the adversary can broadcast a WPA2-only network with the given SSID. This causes the client to connect to our rogue AP using WPA2. The adversary can forge the first message of the 4-way handshake, since this message is not authenticated (stage ③ of Fig. 7 in the Appendix). In response, the victim will transmit message 2 of the 4-way handshake, which is authenticated. Based on this authenticated handshake message, a dictionary attack can be carried out [41].

We tested the above attack against the client-side implementations of WPA3 listed in Table II. Note that with the first four devices, the network to connect to is selected by the user from a list of nearby ones. Here iwd and the Galaxy S10 are vulnerable, though Linux's NetworkManager and the Google Pixel 3 were not affected. With the last three devices, the network to connect with must be manually configured. That is, we had to specify the name of the network, and that it uses WPA3 in transition mode. We then let this device connect to the WPA3 network. After that we replaced the WPA3 network by a rogue WPA2-only network with the same name. This revealed that these three devices all tried to connect to the WPA2 network, allowing subsequent dictionary attacks.

We also discovered an implementation-specific downgrade attack. More precisely, some devices connect to the rogue WPA2 network, even when the legitimate (i.e. original) network only supports WPA3 (column 3-Only in Table II). For example, iwd and the Galaxy S10 are affected by this attack, meaning downgrade to dictionary attacks are possible even if the target network only supports WPA3.

2) Attacking SAE's Group Negotiation. The SAE handshake can be run using different elliptic curve or multiplicative groups, and the 802.11 standard allows station to prioritize groups in a user-configurable order [17, §12.4.4.1]. Although this provides flexibility, it requires a secure method to negotiate the group to use. Unfortunately, the mechanism that SAE uses to negotiate the desired group is straightforward to attack.

With SAE, the group is negotiated by letting the client include its desired group in the commit frame, along with a valid scalar s_i and element E_i. If the AP does not support this group, it replies using a commit frame with a status field equal to "unsupported group" (stage ① in Fig. 2). In turn the client



Client
Adversary
AP
①
Auth-Commit(group=21, s_A, E_A)
Block
Auth-Commit(status=unsupported)
②
Auth-Commit(group=19, s'_A, E'_A)
Auth-Commit(group=19, s_B, E_B)
confirm phase

Fig. 2: Downgrade attack against SAE's group selection: a man-on-the-side can force the client (initiator) into using a different cryptographic group during the SAE handshake.

sends a new commit frame using its next preferred group, along with a new s_i and E_i. This process continues until the client selects a curve that the AP supports. Unfortunately, there is no mechanism to detect if someone interfered with this process. This makes it straightforward to force the client into using a different group by forging a commit frame that indicates the AP does not support the selected group.

Figure 2 illustrates the resulting group downgrade attack, where the adversary acts as man-on-the-side. The client first constructs a commit frame requesting group 21 (curve P-521). However, the adversary blocks this frame from arriving at the AP (stage ① in Fig. 2). The adversary then forges a commit frame that indicates the AP does not support the requested group. In response, the client picks its second preferred group, which in our example is group 19 (curve P-256). From this point onwards, a normal SAE handshake is executed using group 19 (stage ② in Fig. 2). This negotiation process is never cryptographically validated, meaning the downgrade attack is not detected. We confirmed this attack against wpa_supplicant. To block specific commit frames, we modified Atheros firmware to read the header of frames while they are still being transmitted, to then jam the remaining content in case it is a commit frame we want to block [28].

It is also possible to force the victim into using a bigger or more secure cryptographic group. This may be useful when performing denial-of-service or timing attacks.

3) Countermeasures. To mitigate our downgrade to dictionary attack, a client should remember if a network supports WPA3-SAE. That is, after successfully connecting using SAE, the client should store that the network supports SAE. From this point onward, the client must never connect to the network using a weaker handshake. This trust-on-first-usage is similar to the one of SSH, and similar to the Strict-Transport-Security header of HTTPS [42]. Note that Linux's NetworkManager and the Google Pixel 3 already employ a similar defense. In case the client notices that the network no longer supports WPA3-SAE, it can prompt the user for the network's password. This would prevent automatic downgrade attacks, while still allowing the user to override our defense by reentering the password. To handle networks where only some APs support WPA3, a flag can be added to the RSNE that indicates only some APs of a network support WPA3, meaning downgrade attacks against this network cannot be prevented. Another defense, which requires no software patches, is to deploy separate WPA2 and WPA3 networks with different passwords.

Group downgrade attacks can be mitigated by including a bitmap of the supported groups in the RSNE during the 4-way handshake. This will enable a station to detect if a downgrade attack took place, and to subsequently abort the handshake.

TABLE III: Operations needed to hash to a curve if a constant time Legendre function is used [8] instead of quadratic residue blinding. With Brainpool curves costs double (see Section V).

Method	Hash	$x+y$	$x \cdot y$	x^2	x^y	x^{-1}	$\sqrt{x}$
Dragonfly	80	80	40	40	80	0	1
Icart	1	5	6	3	1	1	0
SWU	2	8	6	5	2	1	1
S-SWU	1	6	4	4	1	1	1

B. The High Overhead of Dragonfly

We now compare Dragonfly's overhead with other methods. This reveals that, due to defenses against known side-channels, its overhead is intolerably high. We then defeat SAE's denial-of-service defense and abuse this high overhead in practice.

Analyzing and Comparing Dragonfly's Overhead. When counting the number of operations that Dragonfly's hash-to-curve method requires, we find that it requires an order of magnitude more operations than alternative methods (see Table III). This high overhead is caused by the try-and-increment loop, where at least 40 iterations are always executed to mitigate timing leaks. When using Dragonfly with Brainpool curves, roughly 80 instead of 40 iterations must be executed at minimum (see Section V-B), meaning all operations except the square root further double in number. Note that in Table III we assume Dragonfly uses a constant-time Legendre function to determine if a number is a quadratic residue. This underestimates the running time of implementations, because the 802.11 standard recommends using a blinding technique instead of a constant-time Legendre function (recall Section III-C). Using the blinding technique would add an additional 120 multiplications and 40 random number generations, making the algorithm even less efficient.

The designers of Dragonfly realized that an adversary can abuse the high overhead by spoofing commit frames in a DoS attack. To defend against this an anti-clogging mechanism was added to SAE [26, §8.2a.6]. In this defense the client must reflect a secret cookie sent by the AP, before the AP processes the client's commit frame. This is inspired by IP-based protocols such as IKEv2, where such defenses prevent an adversary from initiating handshakes using spoofed IP addresses [43], [44]. Although an adversary can still use its real address in forged frames, the idea is that these frames can then be throttled based on their source address.

Defeating SAE's Anti-Clogging Defense. The anti-clogging mechanism of SAE was designed to prevent DoS attacks that flood a victim with commit frames from forged MAC addresses [17, §12.4.6]. However, unlike IP addresses, it is trivial to spoof MAC addresses. Additionally, in any broadcast medium like Wi-Fi, an adversary can easily capture and replay secret cookies. To demonstrate this, we created a tool where the adversary acts as a client, injects commit frames using spoofed MAC addresses, and reflects any secret cookies it receives. We used the virtual interface support of Atheros chips to acknowledge all frames sent to forged MAC addresses. This assures the AP does not retransmit replies, meaning more commit exchanges can be spoofed to overload the AP.

In our experiments, the adversary uses a Raspberry Pi B+ with a 700 MHz CPU and a WNDA3200 Wi-Fi dongle, and the target is a professional AP with a 1200 MHz CPU. We first perform the attack using curve P-521, and found that spoofing 8 commit exchanges per second causes the AP's CPU usage to reach 100% (see Fig. 5 in the Appendix). Clients that now try to connect using WPA3 either face long delays, or cannot connect at all. During the attack, the CPU usage of the attacker was only 2.7%. It is worrying that such a devastating attack is possible against a modern security protocol. When using curve P-256, the target's CPU overloads when spoofing 70 commit exchanges per second (see Figure 6 in the Appendix). This attack consumes 14.2% of the attacker's CPU. Since all APs must support curve P-256, this shows our attack can be performed against any WPA3 network using cheap devices.

Figure 5 and 6 also show the amount of airtime consumed by the injected frames. This is just a fraction of the available airtime, showing our attack is more efficient than a straightforward DoS where an attacker simply jams the channel.

Countermeasures. Dragonfly can be modified such that the password element is independent of the peers' identities. The password element can then be calculated offline and reused in all handshakes, preventing our attack. Another option is using a more efficient hash-to-curve method (e.g. one of Table III).

A backwards-compatible defense is to derive the password element in a low-priority background thread. Although legitimate WPA3 clients would still be unable to connect during an attack, this assures other functionality is not impacted. Additionally, larger curves or MODP groups can be disabled by default, to reduce the impact of DoS attacks.

Side-Channel Defenses are Too Costly. Our DoS attack shows that Dragonfly's timing leak defenses are too costly. This overhead even caused handshake timeouts when quadratic residue blinding was added to hostapd [45]. To avoid timeouts the standard was updated to give stations 2 seconds, instead of 40 ms, to process commit frames [46]. However, even with this longer timeout, we believe that lightweight devices will not fully implement all defenses because they are too costly. In fact, in Section III-D we found two Wi-Fi radios that perform at minimum only 8 instead of 40 iterations, making them vulnerable to timing attacks. For Brainpool curves the overhead is even more problematic, because for them the cost of side-channel defenses doubles (see Section V).

V. Timing Attacks

In this section we show that the hash-to-group and hash-to-curve methods are vulnerable to (novel) timing attacks. The obtained info can be used to recover the victim's password.

Listing 2: Hash-to-group method in Python-like pseudocode. If token is None the SAE variant is executed [17, §12.4.4.3.2], otherwise it executes the EAP-pwd variant [16].

```
1 def hash_to_group(password, id1, id2, token=None):
2   label = "EAP-pwd" if token else "SAE"
3   for counter in range(1, 256):
4     seed = Hash(token, id1, id2, password, counter)
5     value = KDF(seed, label + " Hunting and Pecking", p)
6     if value >= p: continue
7
8     P = $value^{(p-1)/q} \bmod p$
9     if P > 1: return P
```

A. *Variable Number of Iterations*

Both the SAE and EAP-pwd protocol also support MODP groups, in which case the hash-to-group method in Listing 2 is used. Although the Crypto Forum Research Group (CFRG) warned that lines 5 and 9 cause timing leaks [47], their proposed defenses are not part of SAE or EAP-pwd. We believe this is due to misunderstandings about which defenses are required. In particular, line 9 in fact causes no leaks. The CFRG's false warning was due to a misinterpretation of the hash-to-group method [48]. This false warning likely caused EAP-pwd and SAE not to use any defenses at all. Surprisingly, this misunderstanding also caused an unnecessary defense to be used in Dragonfly's TLS-PWD variant [20]. To remedy these misunderstandings, we analyze both hash-to-group and hash-to-curve for (novel) timing leaks, exploit the leaks in practice, and discuss backwards-compatible defenses.

We first clarify why line 9 causes no leaks. This line takes a random $value$ and turns it to a member of a q-sized subgroup. It only results in 0 or 1 if $value$ is a member of a different-sized subgroup. For all MODP groups, this probability is lower than 2^{-322}, meaning in practice it never causes extra iterations.

Line 5 causes extra iterations when the output of the Key Derivation Function (KDF) returns a number bigger or equal to the prime p. The CFRG warned about this, but did not analyze the leak in detail [47]. The number of bits returned by KDF is equal to the number of bits needed to represent p, meaning the probability that $value$ is bigger than p depends on the group being used. For most MODP groups this probability is negligible, because p is close to a power of two. However, for the RFC 5114 groups 22, 23, and 24, the probability that the output of KDF is bigger than p is high [49]. For example, for group 22 this probability equals 30.84%, and for group 24 the probability is 47.01% (see column 3 in Table IV).

Since the KDF output depends on the password, the number of executed iterations also depends on the password. If someone learns this number, they learn that passwords which need a different number of iterations are not being used. For hash-to-group the number of executed iterations X follows a geometric distribution:

$$\Pr[X = n] = \Pr[value \geq p]^{n-1} \cdot (1 - \Pr[value \geq p]) \quad (1)$$

TABLE IV: Timing leaks for MODP groups (top) and Brainpool curves (bottom). Column 3 shows the probability that the KDF output is bigger or equal to p, column 4 shows the average number of iterations needed to find the password element, and the last column contains the lowest k such that needing more than k iterations has a probability below 2^{-40}.

Group	len(p)	$\Pr[value \geq p]$	$E[X]$	k
22	1024	30.84%	1.44	24
23	2048	32.40%	1.48	25
24	2048	47.01%	1.89	37
27	224	15.72%	$2 \cdot 1.19$	51
28	256	33.60%	$2 \cdot 1.51$	69
29	384	45.03%	$2 \cdot 1.82$	86
30	512	33.26%	$2 \cdot 1.50$	68

Hence the average number of iterations needed to derive P for MODP groups equals $E[X] = (1 - \Pr[value \geq p])^{-1}$. For group 22, this equals 1.45, and for group 24 this equals 1.89 iterations. In other words, on average one timing measurement allows the adversary to learn the result of multiple iterations. Moreover, the MAC addresses (i.e. identities) of the peers also influence the output of the KDF, and hence also influence the number of executed iterations. This means we can attack clients and APs by spoofing MAC addresses, and for each address measure the number of executed iterations. We show in Section VII how this info can be used to recover the password.

B. Timing Attacks against Brainpool Curves

During our initial coordinated disclosure, the Wi-Fi Alliance privately created recommendations to mitigate our attacks [12]. These recommendations state that Brainpool curves are safe to use, and that no extra defenses are needed when using them. However, even though the hash-to-curve method already has timing leak defenses, it still suffers from timing leaks when using Brainpool curves. The problem is that, similar to hash-to-group, the hash-to-curve method also checks if the KDF output is smaller than p (line 8 in Listing 1). For most curves this is not an issue, since their prime is close to a power of two, but with Brainpool curves this check can fail with high probability (see group 27 to 30 in Table IV).

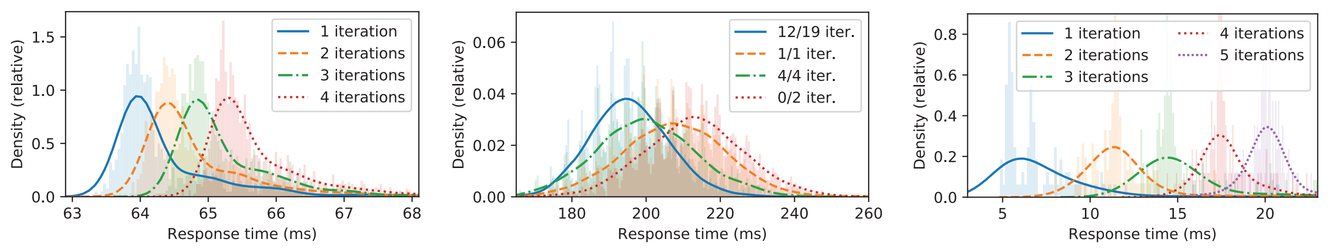
When the KDF output is too big, the hash-to-curve algorithm does not check if the output is a quadratic residue. Since the KDF output depends on the password, the execution time depends on the password as well, resulting in a timing leak. However, due to the existing defense of executing extra iterations based on a random password, this leak is non-trivial to exploit. The problem is that, in the extra iterations, a random number of them have a KDF output smaller than p, meaning the execution time of the hash-to-curve method is also random. Nevertheless, the variance of the execution time depends on when the password element is found. That is, the more iterations that use the real password (called real iterations), the lower the variance. Additionally, the *average* execution time depends on the number of real iterations and on how many of those real iterations have a KDF output smaller than p. In other words, when using Brainpool curves, the variance and average of the execution time leak info about the password. However, we cannot determine in which iteration(s) the KDF output was smaller than p. For example, if in 1 out of 5 real iterations the KDF output was below p, timing info cannot reveal in which iteration this occurred. On top of this, determining the number of real iterations is non-trivial in practice. Although the variance of the timing measurements depends on the number of real iterations, a large number of measurements are needed to accurately differentiate MAC addresses that result in a different number of real iterations. Nevertheless, the variance and average execution time *differences* do form a fingerprint of the password. In Section VII we show how this fingerprint can be used to recover the password.

C. Experiments against WPA3-Enabled APs

Our first two experiments target APs (recall Section III-B for our threat models). We used a Raspberry Pi 1 B+ for the AP because its 700 MHz CPU matches the typical CPU of home routers and professional APs [50]. The Raspberry Pi used a WNDA3200 Wi-Fi dongle. Hostapd was used as the AP daemon, since it is the most widely used daemon in both professional and home routers. We wrote a tool that spoofs commit frames, and measures response times. After each measurement, a deauthentication frame is sent, causing the target to clear all state related to the spoofed address and enabling us to rapidly perform new measurements.

Two optimizations are important. First, we use virtual interface support of Atheros chips to acknowledge frames sent to spoofed MAC addresses. This stops the AP from retransmitting frames, making the attack more reliable. Second, response times are influenced by background traffic and background tasks on the AP. Both sources of noise are problematic because they are not constant throughout an attack. To handle this, we interleave the time measurements of spoofed MAC addresses, instead of performing all measurements for each address one by one. As a result, temporal noise equally influences the timings of all addresses, instead of only affecting one address.

In our setup we attacked hostapd 2.6 using MODP group 22. We spoofed 20 addresses and made 1 000 measurements for each address. Figure 3a shows the response time distributions of selected MAC addresses that result in a various number of iterations. We evaluated several statistical tests to differentiate addresses that result in a different number of iterations, such as simple averages, Student's and Welch's T-test, Mann-Whitney U test, Wilcoxon signed-rank test, one-way ANOVA, and Crosby's box test [51]. Here there is a trade-off between the number of differences detected, and the chance of false positives. We want to detect as many differences without false positives. Even under these conditions, Crosby's box test outperformed all classical tests. When using this test with a low percentile of 5 and high percentile of 35, we need 75 measurements per address to differentiate all addresses that require a different number of iterations with 99.5% confidence. These pairwise comparisons are then used to sort MAC addresses based on the number of executed iterations. From this ranking we derive bounds on how many iterations each MAC address



(a) WPA3 AP with MODP group 22. (b) WPA3 AP with Brainpool curve 29. (c) EAP-pwd client with curve P-256.

Fig. 3: Response time distributions of example timing attacks for selected parameters. The victim uses a Raspberry Pi 1 B+. Graph 3a targets hostapd 2.6, graph 3b hostapd 2.8, and graph 3c iwd 0.14. The legend in 3b shows the parameters n/m, where m is the number of real iterations, and n the number of real iteration with a too high KDF output.

executed. We tested this approach by successfully using it to recover the password with the techniques in Section VII.

We tested the Brainpool timing attack against hostapd 2.8 using Brainpool curve 29. We spoofed 20 MAC addresses, and for each address made 2 000 measurements. The resulting timing distributions of four selected addresses are shown in Figure 3b. Recall that the response time of each address is determined by the number of real iterations, and by how many real iterations have a KDF output smaller than the prime p. The various tops around the median of each distribution are caused by the dummy iterations performed on a random password, and correspond to how many of the extra iterations had a KDF output smaller than p. Also notice how addresses with fewer real iterations have a lower variance. To differentiate addresses that result in a different average timing response, we again use Crosby's box test. To differentiate addresses with a difference variance, we use the Levene test. In our setup, Crosby's box test with a low percentile of 45 and high percentile of 60 correctly (i.e. without false positives) found most differences with 300 timing measurements per address.

D. Attacking SAE and EAP-pwd Clients

Our next experiment targets clients (recall Section III-B). To simulate devices that offload the SAE handshake to their Wi-Fi chip, we tested our attacks against a Raspberry Pi 1 B+ with iwd as a lightweight client. Running iwd on the Raspberry Pi required recompiling Linux to enable recent kernel features.

To attack a WPA3-SAE client, we need to know when it starts executing the hash-to-element method. Since the client initiates the handshake, we cannot do this for the first commit frame it sends. Instead, the rogue AP responds to the client that the offered group is not supported. This causes the client to build a commit frame for another group, which requires executing the hash-to-element from scratch. We can measure how long this takes, and hence perform timing attacks against both WPA3-SAE and EAP-pwd clients.

As an example, we perform a timing attack against an iwd client using EAP-pwd with curve P-256. With EAP-pwd, the number of executed iterations are influenced by the client's username, the identity of the server, and by a token generated by the server (see e.g. line 7 in Listing 1). Because the server always generates a new random token, we cannot attack it. Instead we attack the client by spoofing 20 different tokens. The resulting timing measurements for selected tokens are shown in Figure 3c. Using Crosby's box test with a low percentile of 5 and high percentile of 45, we can recover the number of iterations using 30 timing measurements per token.

E. Discussion and Countermeasures

We recommend preventing the MODP timing attacks by disabling groups 22, 23, and 24 since these groups are also considered insecure due to their many subgroups [52]. Following RFC 8247, groups 1, 2, and 5 should also be disabled [53]. A backwards-compatible defense is executing extra dummy iterations like was done in hash-to-curve. To have the same security guarantees as with NIST curves in hash-to-curve, implementations should set the security parameter k such that the probability of needing more than k iterations to find P is below 2^{-40} (see Table IV for these values).

A backwards-compatible defense against the Brainpool timing attack is executing line 10 even if the KDF output is too big, and using constant-time select utilities to return the proper result. Additionally, to have the same security guarantees as with NIST curves, implementations should set the security parameter k such that the probability of needing more than k iterations is below 2^{-40} (see Table IV). Unfortunately, resource-constrained devices may be unable to perform that many iterations due to the resulting overhead. In fact, in Section III-D we already found firmware for lightweight Wi-Fi radios that uses only $k = 8$. This highlights that the hash-to-curve algorithm of Dragonfly is flawed by design, and that leaks are hard to mitigate in practice.

A better defense is to exclude the MAC addresses (i.e. identities) from the hash-to-element methods. Similar to our defense from Section IV-B against DoS attacks, this would allow the password element to be calculated offline and then reused. Although timing leaks may still occur, for a given password the execution time would then always be identical, meaning on average only two password bits are leaked. This change also makes it harder to trigger and measure executions of the algorithm. Another option is to use a constant-time hash-to-curve method (e.g. one of Table III).

VI. Cache-Based Attacks on ECC groups

In this section we demonstrate that implementations of the hash-to-curve algorithm of SAE may be vulnerable to cache-based side-channel attacks. The leaked information will later on be used to recover a target's password.

The goal of our attack is to learn if the Quadratic Residue (QR) test in the first iteration of the hash-to-curve algorithm succeeded or not. This information will be used in the offline password brute-force attack of Section VII to recover the target's password. Unlike the hash-to-element method, the implementation of the hash-to-curve algorithm for ECC groups does include mitigations against side-channel attacks. Those mitigations include performing extra dummy iterations on random data [17, §12.4.4.3.2], and blinding of the underlying cryptographic calculation of the quadratic residue test [25]. The resulting code of wpa_supplicant and hostapd implementation we reviewed is pseudo-constant time, i.e., there might be some minor variation in run time, but they are too minute to be measured by an adversary. However, such pseudo-constant time implementations might still be vulnerable to different types of micro-architectural side-channel attacks [29]–[31].

A. *Micro-Architectural Side-Channel Attacks*

Modern processors try to optimize their behavior (e.g. memory access, branch prediction) by saving an internal state that depends on the past. Micro-architectural side-channel attacks exploit leaked information about the running of other programs due to sharing of this state (for a survey see [54]). Cache-based side-channel attacks exploit the state of the memory cache (either instructions or data) and have been widely used to break cryptographic primitives [55]–[59]. Cache attacks can be seen as a way to partly circumvent process (or virtual machine) isolation. Although an attacker running code in an unprivileged process is not able to read the memory of the target process, he can still learn information about the memory access patterns.

In the Flush+Reload attack [58], the adversary starts by evicting (or flushing) a memory location from the cache. After waiting for a predetermined interval, he measures the time it takes to reload the flushed location and then flushes it again. If during the interval the victim accesses this memory location, it will be cached, and the reload time for the attacker will be short. Otherwise, the reloading of the flushed memory location will be much slower. In this way, the attacker can trace the victim's memory access patterns.

B. *Attacking the hostap Implementation*

We target the `sae_derive_pwe_ecc` function of the latest hostap code before our initial disclosure (commit `0eb34f8f2`) with the default curve P-256. Our test machine uses a 4-core Intel Core i7-7500 processor, with a 4 MiB cache and 16 GiB memory, running Ubuntu 18.04.1. We monitor the instruction cache accesses of wpa_supplicant with an unprivileged user-mode spy process (recall Section III-B for our thread model). This is accomplished using the Flush+Reload attack of the Mastik toolkit [58], [60].

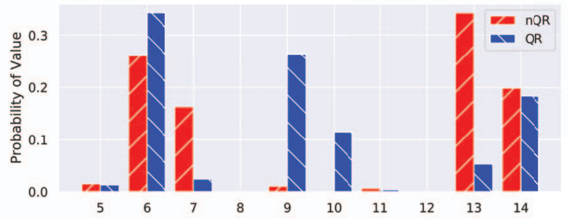


Fig. 4: Probability distribution for attack results

We want to leak the result of the QR test in the first iteration of the hash-to-curve algorithm. We can try to attack the blinded QR test code, or the code that checks the result of the test. A simple cache attack against the blinded QR test is infeasible as the two possible code paths are compiled into a single cache line (see Listing 4 line 21).[1]

The two code paths of the branch inside the iteration loop (see Listing 6 line 29) are compiled into two separate cache lines. Therefore we can monitor cache access to nQR case cache line which is the target of the conditional jump (see Listing 7 line 9). To differentiate between the branches taken in the first and subsequent iterations, we created a synchronization "clock" by monitoring another cache line that is accessed once every iteration (similarly to [63]).

On our test platform, monitoring two cache lines repeatedly over time caused a high rate of false positives (i.e. false detection of access to cache lines). This error rate increases considerably when the monitored cache lines are close. Consequently, for our "clock" monitor a cache line far away from the nQR cache line (in our case the function `sha256_prf_bits`).

1) Template Attack. We want to learn the result of the QR test in the first iteration for each cache trace we measured. However, our measurements are noisy, and the measured cache access patterns to the two monitored cache lines show a high variance between different traces with the same result. This might be due to OS-related noise, speculative execution, or due to the influence of the random dummy iterations on the branch predictor. To overcome this, we perform a simplified variant of a cache template attack [64], [65]. That is, we measure a trace of the cache access pattern by monitoring the two addresses (the "clock" and the non-QR case) in fixed intervals of $5 \cdot 10^4$ clock cycles (each iteration takes roughly $2 \cdot 10^5$ clock cycles on our test machine). We encode each measurement as a bit, with value one if the measured cache line was accessed and zero if it wasn't. Each interval corresponds to two bits. We encode each interval in the trace into two bits that correspond to the two memory locations.

Our attack returns the first two non zero intervals. This means the return value consists of 4 bits (resulting in 9 possible return values). Figure 4 shows the distribution of these return values when the first iteration of the hash-to-curve algorithm results in a non-QR number (nQR), and when the first iteration results in a QR number (QR).

To overcome the noise and achieve a high success rate, we repeat the attack for 20 times for each MAC address, and use

[1]More advanced micro architectural attacks targeting the branch predictor [31], [61], [62] will fail due to the extra random iterations.

a simple linear classifier to get the result.

We trained our classifier with two training sets of $100 \cdot 20$ traces for each of the non-QR and the QR cases. We then tested our attack and linear classifier on a larger test set of $400 \cdot 20$ traces for each case. We achieved a 100% success rate (400 out of 400) in the non-QR case, and a 99.5% success rate (398 out of 400) in the QR case.

C. Cache Attacks against Brainpool Curves

After our initial coordinate disclosure, hostap mitigated the vulnerabilities described in Section VI-B. However, as discussed in Section V-B, the patched code still has a secret-dependent branch that can be exploited when using Brainpool Curves (see Listing 5). This allows for a new cache attack similar to our original one in the same attack scenario. Using the same test setup and technique, targeting the latest patched version of hostap (commit `e0e15fc23`).

The fixed interval of our attack is reduced to $5 \cdot 10^3$ clock cycles (if the resulting hash is larger than the modulus then the iteration is very short). For our clock we monitor a cache line inside the `hmac_sha256_vector` function that is accessed once in each iteration (called by `sha256_prf_bits`). The second monitored cache line is inside the `crypto_bignum_init_set` function that is called only if the resulting hash is smaller than the modulus (see Listing 5 Line 16). This new attack is more robust than the original one, achieving 100% success rate using only 10 traces for each MAC address.

D. Discussion and Countermeasures

We believe that all Dragonfly variants are affected by our attack. Similar to our timing attacks, the ideal solution is to use a constant-time hash-to-curve method, and to exclude the peer's identities from the password element computation. As a backwards-compatible defense, a constant-time Legendre function can be used, secret-dependent branches can be replaced with constant-time select utilities, and at least k iterations must always be executed. Additionally, when using Brainpool curves, line 10 in the hash-to-curve algorithm of Listing 1 should always be executed.

VII. Brute-forcing the Password

In this section we show how to recover the password using the information obtained from our side-channel attacks.

A. Abusing Leaked Information

Most side-channels reveal if an element test in an iteration failed or not. An element test refers to any check done in a single iteration, such as testing if the KDF output is smaller than p, or checking if a number if a quadratic residue. A *failed element test* causes another iteration, while a *successful* element test means the current iteration continues executing. A successful element test does not imply the password element is found. For instance, the test that checks if the KDF output is lower than p might succeed, but this value may not result in a quadratic residue. We let p_e denote the probability that the element test failed. For quadratic residue tests, p_e is close to

Listing 3: Code to check whether a password causes the same element test results as recovered by our side-channels, i.e., it checks whether the password may be in use by the victim.

```
def check_password(pw, data, id2):
  # pw: Password to test.
  # data: Element test results for each spoofed identity or
  #      token, and counter value used in each element test.
  # id2: identity of the target (e.g. MAC address).
  for id1, token, counter, result in tests:
    if simulate(pw, token, id1, id2, counter) != result:
      return False
  return True
```

50%, and for KDF output tests the values for p_e are listed in Table IV under $\Pr[value \geq p]$. Recall that one MODP timing measurement can reveal the result of multiple (failed) element tests, and a cache-attack reveals the result of a single element test. Also note that if we are unsure whether a spoofed address resulted in say 4 or 5 iterations, this still learns us that the first three element tests failed.

Element tests are used to prune wrong passwords by simulating the test on candidate passwords, and pruning passwords if the simulated result differs from the real result. By representing side-channel leaks as element tests, we can use the same brute-force method for all our side-channel attacks. Moreover, we can mix element tests of different side-channel leaks.

Our goal is to recover the password from a given dictionary. We do this by iterating over the dictionary, and using element tests to prune bad passwords. If this prunes all passwords, the target's password was not in the dictionary. Passwords that are not pruned can be tested by using them to connect to the network. The algorithm that implements this brute-force search is straightforward: it gets as input a dictionary, and a set of element test results. For every password in the dictionary, it uses the function shown in Listing 3 to check if the password may be in use by the victim. Notice that this algorithm can be run offline, without requiring any interactions with the target.

A different approach is needed for Brainpool timing attacks, because there we cannot recover test results of specific iterations. For address pairs where one has a larger variance or execution time than the other, we simulate the hash-to-curve on the guessed password, and prune the password if the simulated execution time does not match the measured differences.

B. Brute-force Success Analysis

The probability of pruning a password using a single element test depends on whether it is a failed or successful test. A failed element test has a probability of $1 - p_e$ to prune a random (incorrect) password, while a successful test has a probability of p_e to prune a password. Therefore, when we are given n element tests, we first want to know the probability that k of them are failed element tests. This probability equals

$$\Pr[S_n = k] = \binom{n}{k} \cdot (1 - p_e)^k \cdot p_e^{n-k} \tag{2}$$

where S_n follows a binomial distribution with a success probability of $1 - p_e$. Note that every element test is independent, because in each iteration the hash inputs are different, resulting in independent hash outputs and quadratic tests. The probability that a password is *not* pruned by all n element tests now is $(1-p_e)^k \cdot p_e^{n-k}$, meaning the probability that d incorrect passwords *are* eliminated equals:

$$\Pr[E = d \mid S_n = k] = \left(1 - (1-p_e)^k \cdot p_e^{n-k}\right)^d \quad (3)$$

Here random variable E denotes the number of pruned passwords when given n element test results, and the conditional probability assumes that k out of the n element tests are failed ones. Based on the above two formulas, we can calculate the probability of eliminating d passwords given n element tests:

$$\Pr[Z_d \leq n] = \sum_{k=0}^{n} \Pr[S_n = k] \cdot \Pr[E = d \mid S_n = k] \quad (4)$$

Here random variable Z_d denotes the number of element tests needed to prune d random passwords. Intuitively, this formula calculates the probability of pruning $E = d$ passwords, given that k out of n elements tests are failed ones. We confirmed this formula by running 10^5 runs of the brute-force algorithm on 10^3 passwords, where each run used random simulated element test results (e.g. simulated timing measurements).

Taking the RockYou dump as reference [66], which contains about $1.4 \cdot 10^7$ passwords, brute-force attacks with curve P-256 require 29 element tests to uniquely recover the password with a probability above 95%. Since our cache attack can detect a QR with 100% accuracy, and a non-QR with 99.5% accuracy, the probability that on average all measurement are correct is $0.995^{12.5} = 0.939$. The probability of uniquely recovering the password becomes at least 0.892. That is, with 25 cache-based element test results, the probability of uniquely recovering the password from the RockYou dump is close to 90%.

We can also determine the average number of element tests ℓ needed to prune all incorrect passwords from the dictionary:

$$\ell = \sum_{i=1}^{\infty} i \cdot \Pr[Z_d = i] = \sum_{i=1}^{\infty} i \cdot (\Pr[Z_d \leq i] - \Pr[Z_d \leq i-1]) \quad (5)$$

We confirmed this formula by brute-forcing a password 10^5 times with random element tests. For the RockYou dump, on average 28.28 MODP-based element tests are needed to uniquely recover the password. With tests based on curve P-256, the adversary needs on average 25.11 element tests.

C. Computational Requirements

To estimate the offline computational costs of brute-force attacks using a dictionary of size d, we derive the expected number of element tests that must be simulated in the brute-force algorithm (line 7 in Listing 3). We assume the algorithm first uses failed element tests, since these tests have a higher probability to prune passwords. Namely, a failed element test has a probability of $1 - p_e$ to prune a random password. Assuming k' out of n element tests are failed ones, the average number of simulated element tests to prune one password is

$$\sum_{n=1}^{k'} n \cdot p_e^{n-1} \cdot (1-p_e) + p_e^{k'} \cdot \sum_{n=1}^{\infty} (k'+n) \cdot (1-p_e)^{n-1} \cdot p_e \quad (6)$$

TABLE V: Cost of a brute-force attack for various dictionaries.

Group / Dictionary	Dictionary Size	\$ for MODP 22 Brainpool 28	\$ for P-256
RockYou [66]	$1.4 \cdot 10^7$	$2.1 \cdot 10^{-6}$	$4.4 \cdot 10^{-4}$
HaveIBeenPwned [68]	$5.5 \cdot 10^8$	$8.0 \cdot 10^{-5}$	$1.7 \cdot 10^{-2}$
Probable Wordlists [69]	$8.0 \cdot 10^9$	$1.2 \cdot 10^{-3}$	$2.5 \cdot 10^{-1}$
8 Low Case	$2.1 \cdot 10^{11}$	$3.0 \cdot 10^{-2}$	6.5
8 Letters	$5.3 \cdot 10^{13}$	7.8	$1.7 \cdot 10^3$
8 Alphanumerics	$2.2 \cdot 10^{14}$	$3.2 \cdot 10^1$	$6.7 \cdot 10^3$
8 Symbols	$4.6 \cdot 10^{14}$	$6.7 \cdot 10^1$	$1.4 \cdot 10^4$

The first term assumes the password is pruned by a failed element tests, and the second term assumes it is pruned by a successful element test. In case $p_e = 0.5$ this formula equals 2. If we collect the average number of element tests ℓ needed to brute-force a dictionary of size d, on average at least $\lfloor \ell \cdot p_e \rfloor$ element tests are failed ones. For MODP group 22 and a dictionary of size $d \geq 10^5$, this would mean that on average we have to perform $1.45d$ element tests, and with the Brainpool curve 28 cache-attack we have to perform $1.51d$ tests. With curve P-256 we always need $2d$ tests on average. We confirmed this formula by running 10^5 runs of the brute-force algorithm, and in each run used 10^3 random passwords, assuming $p_e = 0.3084$. Our formula also implies that if $p_e < 0.5$, having more failed element tests increases the efficiency of the brute-force algorithm. Hence there is a trade-off between performing additional side-channel attacks to obtain extra failed element tests, and performing a more expensive brute-force search.

D. Computational Costs in Practice

To estimate the cost of performing an offline brute-force attack, we run benchmarks on a NVIDIA V100 GPU. When brute-forcing using element test results based on MOD group 22, or cache attacks against Brainpool curve 28, we assume that the computational cost is dominated by the SHA56 operation. When using element tests based on P-256, we assume the cost is dominated by the Legendre function.

Based on the Hashcat benchmark for SHA256, we can evaluate $7.56 \cdot 10^9$ hashes per second. For Brainpool curve 28, we need an average of 1.51 element tests to prune a password, and one element test requires three SHA256 operations. Hence we get a total rate of roughly $1.67 \cdot 10^9$ passwords per second for Brainpool (and a slightly higher rate for MODP group 22). We also benchmarked the Legendre symbol calculation for P-256 using the PowMod function from the XMP library [67], achieving $15.74 \cdot 10^9$ operations per second. This was done without any curve specific optimizations. Since we need an average of two elements tests to prune a password, we get a total rate of roughly $7.87 \cdot 10^9$ passwords per second.

Table V shows the resulting cost of brute-forcing dictionaries of various sizes for MODP group 22 and Brainpool curve 28, and the more costly attack against the P-256 curve. In our calculations we used the current \$7.344 per hour spot price on Amazon AWS cloud for p3.16xlarge with 8 V100 GPUs [70].

When using Brainpool timing leaks to brute-force passwords, we cannot use the same analysis. This is because, if we treat the detected variance or execution time differences as element tests, then element tests may no longer be independent. For example, when using the observation that address A resulted in a higher execution time than address B, then when using this test to prune passwords, on average 0.43 percent of passwords remain. When also using the observation that address A has a higher execution than address C, this combination does not mean 0.43^2 passwords remain, meaning both tests are dependent. When only using tests based on average execution times that do not share a MAC address, on average 6.02 hashes and 4.00 quadratic tests are needed to prune one password (for Brainpool curve 28). More efficient brute-forcing strategies for Brainpool timing measurements, including their costs analysis, are left as future work.

VIII. Related Work

After the introduction of WPA, it was quickly found to be vulnerable to dictionary attacks [41]. Later, He and Mitchell formally analyzed WPA's 4-way handshake, and discovered a DoS vulnerability [71], [72]. This resulted in the standardization of a slightly improved variant [17]. He et al. continued to analyze the 4-way handshake, and proved its correctness [73]. However, implementations of the 4-way handshake were still vulnerable to downgrade attacks [74]. Recently, Vanhoef and Piessens discovered that WPA2 was vulnerable to key reinstallation attacks [1], [75]. Finally, Kohlios and Hayajneh provide an overview of WPA2 and the differences with WPA3 [76].

Researchers also discovered several DoS attack against Wi-Fi networks. The most well-known is the deauthentication attack [77]. Other DoS attacks exploit weaknesses in TKIP [78]. Additionally, Könings et al. found several DoS vulnerabilities in the physical and MAC layer of 802.11 [79], and other researchers constructed jammers using commodity hardware [28], [80]. A detailed survey of DoS attacks at the physical and MAC layer is given by Bicakci and Tavli [81]. Aiello et al. show how susceptibility to denial-of-service attacks can be balanced with the need for perfect forward secrecy [82]. To the best of our knowledge, our clogging attack against WPA3 is the first that overloads the CPU of the victim.

An initial version of Dragonfly was vulnerable to an offline dictionary attack [83]. A modified variant was then specified in 2008 [18]. Several close variants of it have been defined over the years [16], [19]–[21]. Trevor Perrin did a review of an improved draft of the handshake [37], and later provided an overview of other people's comments on the handshake [3]. Struik reviewed a draft of the handshake [4]. Clarke and Hao discovered a small subgroup attack against a draft of Dragonfly, which was mitigated in a new draft [84]. Lancrenon and Skrobot provided a security proof of a close variant of Dragonfly [6]. Finally, Alharbi et al. designed a variant of Dragonfly that attempts to keep computational costs low [85].

Other types of PAKEs have also been proposed by researchers over the years [86]–[94], some of which have been submitted as RFCs [95]–[100], [100]. Finally, there is also research into post-quantum PAKEs [101], [102].

IX. Conclusion and Recommendations

In light of our attacks, we believe that WPA3 does not meet the standards of a modern security protocol. Since EAP-pwd uses a close variant of WPA3's Dragonfly handshake, it is affected by similar flaws. We believe that a more open design process would have avoided these weaknesses.

Most of our attacks abuse the password encoding method of Dragonfly, i.e., abuse its hash-to-group and hash-to-curve method. This indicates that implementing these methods without side-channel leaks is very tedious. Additionally, Dragonfly supports a large variety of cryptographic groups, making it hard to fully analyze the handshake. Both points are evidenced by the fact that after our initial disclosure, patched implementations were still vulnerable to a novel side-channel attack.

Interestingly, a minor change to Dragonfly's password encoding algorithm would have prevented most attacks. That is, the peer's MAC addresses (i.e. identities) can be excluded from the password encoding algorithm, and instead included later on in the handshake. For EAP-pwd the server's random token must also be excluded. This allows the password element to be computed offline, meaning an attacker can no longer actively trigger executions of the password encoding method. It also means that for a given password the execution time of the password encoding method is always the same, limiting the amount of info being leaked, which cannot help an attacker to guess the password by much [103]. Surprisingly, when the CFRG was reviewing a variant of Dragonfly, they in fact suggested this type of change [36]–[40]. If this criticism would have incorporated, most of our attacks would have been avoided. Fortunately, our work resulted in (draft) updates to the standard that do incorporate our proposed design changes [13]–[15].

We conjecture that resource-constrained devices will not fully implement all backwards-compatible side-channel defenses, because the resulting overhead is too high. In fact, we already found Wi-Fi radios that only partly mitigate timing attacks. Moreover, correctly implementing all backwards-compatible side-channel countermeasures is non-trivial. This is worrisome, because security protocols should be designed to reduce the change of implementation vulnerabilities. Finally, although WPA3 and its Dragonfly handshake have their flaws, we still consider it an improvement over WPA2.

Acknowledgment

We thank Yuval Yarom for his helpful insights. We also want to thank Philipp Ebbecke, and an anonymous contributor, for their help in testing downgrade attacks against the Pixel 3 and Galaxy S10. Mathy Vanhoef holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO). This work is partially supported by an ISF grant number 1523/14, and by the Center for Cyber Security at New York University Abu Dhabi (NYUAD). Eyal Ronen is a member of CPIIS.

References

[1] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *CCS*, 2017.

[2] Wi-Fi Alliance, "WPA3 specification version 1.0," Retrieved 6 April 2019 from https://www.wi-fi.org/file/wpa3-specification-v10, Apr. 2018.
[3] T. Perrin, "[TLS] question regarding CFRG process," Retrieved 29 October 2018 from https://www.ietf.org/mail-archive/web/tls/current/msg10962.html, 2013.
[4] R. Struik, "[Cfrg] review of draft-irtf-dragonfly-02 (triggered by [TLS] working group last call for draft-ietf-tls-pwd)," Retrieved 9 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03527.html, Nov. 2013.
[5] J. Salowey, "[TLS] conclusion of WGLC draft-ietf-tls-pwd," Retrieved 7 April from https://mailarchive.ietf.org/arch/msg/tls/Fep2-E7xQX7OQKzfxOoFInVFtm4, Dec. 2013.
[6] J. Lancrenon and M. Škrobot, "On the provable security of the Dragonfly protocol," in *Information Security*. Springer International Publishing, 2015.
[7] M. Vanhoef and E. Ronen. (2019) Dragonblood tools: Dragonslayer, dragondrain, dragontime and dragonforce. [Online]. Available: https://wpa3.mathyvanhoef.com/#tools
[8] S. Scott, N. Sullivan, and C. A. Wood, "Hashing to Elliptic Curves," Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-hash-to-curve-03, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-03
[9] S. V. Smyshlyaev, "Overview of existing PAKEs and PAKE selection criteria," Retrieved 31 May 2019 from https://www.ietf.org/proceedings/104/slides/slides-104-cfrg-pake-selection-01.pdf, Mar. 2019.
[10] N. Sullivan, D. H. Krawczyk, O. Friel, and R. Barnes, "Usage of OPAQUE with TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-sullivan-tls-opaque-00, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-sullivan-tls-opaque-00
[11] CERT/CC. (2019) Vulnerability note vu#871675: Security issues with WPA3. [Online]. Available: http://www.kb.cert.org/vuls/id/871675
[12] Wi-Fi Alliance, "WPA3 security considerations overview," Retrieved 24 May 2019 from https://www.wi-fi.org/file/wpa3-security-considerations, Apr. 2019.
[13] D. Harkins, "Disposition of some SAE comments from LB236 and some comments made outside of LB236," Retrieved 3 August 2019 form https://mentor.ieee.org/802.11/dcn/19/11-19-0387-02-000m-addressing-some-sae-comments.docx, Mar. 2019.
[14] ——, "Finding PWE in constant time," Retrieved 24 July 2019 from https://mentor.ieee.org/802.11/dcn/15/11-19-1173-08-000m-pwe-in-constant-time.docx, Jul. 2019.
[15] ——, "Improved Extensible Authentication Protocol Using Only a Password," Internet Engineering Task Force, Internet-Draft draft-harkins-eap-pwd-prime-00, Jul. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-harkins-eap-pwd-prime-00
[16] D. Harkins and G. Zorn, "Extensible authentication protocol (EAP) authentication using only a password," RFC 5931, Aug. 2010.
[17] IEEE Std 802.11, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2016.
[18] D. Harkins, "Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks," in *The Second International Conference on Sensor Technologies and Applications (SENSORCOMM)*, Aug 2008, pp. 839–844.
[19] ——, "Secure pre-shared key (PSK) authentication for the internet key exchange protocol (IKE)," RFC 6617, Jun. 2012.
[20] ——, "Secure Password Ciphersuites for Transport Layer Security (TLS)," RFC 8492, 2019.
[21] ——, "Dragonfly key exchange," RFC 7664, Nov. 2015.
[22] K. M. Igoe, "Re: [Cfrg] status of DragonFly," Retrieved 9 September 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03264.html, Dec. 2012.
[23] T. Icart, "How to hash into elliptic curves," in *Advances in Cryptology (CRYPTO)*, 2009.
[24] S. Fluhrer, "Re: [cfrg] requesting removal of CFRG co-chair," Retrieved 7 April 2019 from https://mailarchive.ietf.org/arch/msg/cfrg/WXyM6pHDjGRZXZzSc_HlERnp0Iw, Jan. 2014.
[25] D. Harkins, "Addressing a side-channel attack on SAE," Retrieved 9 September 2018 from https://mentor.ieee.org/802.11/dcn/14/11-14-0640-00-000m-side-channel-attack.docx, 2014.
[26] IEEE Std 802.11s, *Amendment 10: Mesh Networking*, 2011.
[27] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *ACSAC*, 2018.
[28] M. Vanhoef and F. Piessens, "Advanced Wi-Fi attacks using commodity hardware," in *ACSAC*, 2014.
[29] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *ASIA CCS*, 2015.
[30] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of TLS are only pseudo secure," in *CCS*, 2018.
[31] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom, "The 9 lives of bleichenbacher's CAT: new cache attacks on TLS implementations," in *To appear in the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2019.
[32] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *CCS*, 2015.
[33] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Advances in Cryptology (CRYPTO)*. Springer, 2000.
[34] A. Antipa, D. Brown, A. Menezes, R. Struik, and S. Vanstone, "Validation of elliptic curve public keys," in *Public Key Cryptography (PKC)*. Springer, 2002, pp. 211–223.
[35] IEEE Std 802.11, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2012.
[36] D. Kügler, "Re: [IPsec] PAKE selection: SPSK," Retrieved 23 April 2019 from https://mailarchive.ietf.org/arch/msg/ipsec/NEicYFDYJYcQuNdknY0etLyfITA, May 2010.
[37] T. Perrin, "[TLS] review of Dragonfly PAKE," Retrieved 9 September 2018 from https://www.ietf.org/mail-archive/web/tls/current/msg10922.html, Dec. 2013.
[38] K. M. Igoe, "[Cfrg] status of DragonFly," Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03258.html, Dec. 2012.
[39] ——, "[Cfrg] status of DragonFly," Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03261.html, Dec. 2012.
[40] R. Struik, "Re: [cfrg] small editorial error in and question on draft-irtf-cfrg-dragonfly-01 (was: Re: CFRG meeting at IETF 87)," Retrieved 10 April 2019 from https://mailarchive.ietf.org/arch/msg/cfrg/Z-nnOKTA4ddmFd17l5KzlRwWm5Y, Jul. 2013.
[41] R. Moskowitz, "Weakness in passphrase choice in WPA interface," Retrieved 26 September 2018 from https://wifinetnews.com/archives/2003/11/weakness_in_passphrase_choice_in_wpa_interface.html, 2003.
[42] Mozilla, "Strict-transport-security - HTTP," Retrieved 3 February 2019 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security, 2019.
[43] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "Internet key exchange protocol version 2 (IKEv2)," RFC 7296, Oct. 2014.
[44] R. Oppliger, "Protecting key exchange and management protocols against resource clogging attacks," in *Secure Information Networks*. Springer, 1999, pp. 163–175.
[45] M. Honma, "[PATCH] mesh: Fix mesh SAE auth on low spec devices," Retrieved 19 September 2018 from http://lists.shmoo.com/pipermail/hostap/2015-July/033304.html, Jul. 2015.
[46] G. Bajko, "SAE reauthentication timer value," Retrieved 19 September 2018 from https://mentor.ieee.org/802.11/dcn/17/11-17-1030-01-000m-sae-retry-timeout-clarification.docx, Jul. 2017.
[47] S. Fluhrer, "Re: [Cfrg] status of DragonFly," Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03265.html, Dec. 2012.
[48] ——, private communication, Nov. 2018.
[49] M. Lepinski and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards," RFC 5114, 2008.
[50] WikiDevi, "Semantic search: wireless routers," Last retrieved 14 November 2018 form https://wikidevi.com/, 2018.
[51] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, 2009.
[52] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger, "Measuring small subgroup attacks against diffie-hellman," in *24th Annual Network and Distributed System Security Symposium NDSS*, 2017.

[53] Y. Nir, T. Kivinen, P. Wouters, and D. Migault, "Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2)," RFC 8247, 2017.
[54] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, 2018.
[55] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006.
[56] D. J. Bernstein, "Cache-timing attacks on AES," 2005.
[57] O. Acıiçmez, "Yet another microarchitectural attack: Exploiting I-Cache," in *CSAW*, 2007.
[58] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.
[59] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018.
[60] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf, 2017.
[61] O. Acıiçmez, S. Gueron, and J. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures," in *IMA Int. Conf.*, 2007.
[62] D. Evtyushkin, R. Riley, N. B. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018.
[63] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time RSA," in *CHES*, ser. Lecture Notes in Computer Science, vol. 9813. Springer, 2016, pp. 346–367.
[64] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 5912. Springer, 2009, pp. 667–684.
[65] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.
[66] N. Cubrilovic, "RockYou hack: From bad to worse," Retrieved 15 November 2018 from https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/, 2009.
[67] NVlabs, "Xmp - cuda accelerated(x) multi-precision library," 2016. [Online]. Available: https://github.com/NVlabs/xmp
[68] T. Hunt, "Have i been pwned?" Last retrieved 23 June 2019 from https://haveibeenpwned.com/, 2019.
[69] Ben, "Probable wordlists - version 2.0," Last retrieved 23 June from https://github.com/berzerk0/Probable-Wordlists, 2019.
[70] Amazon, "Amazon EC2 spot instances pricing," Retrieved 31 May 2019 from https://aws.amazon.com/ec2/spot/pricing/, 2019.
[71] C. He and J. C. Mitchell, "Analysis of the 802.1 i 4-Way handshake," in *WiSe*. ACM, 2004.
[72] J. Mitchell and C. He, "Security analysis and improvements for IEEE 802.11i," in *NDSS*, 2005.
[73] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell, "A modular correctness proof of IEEE 802.11i and TLS," in *CCS*, 2005.
[74] M. Vanhoef and F. Piessens, "Predicting, decrypting, and abusing WPA2/802.11 group keys," in *USENIX Security*, 2016.
[75] ——, "Release the kraken: new KRACKs in the 802.11 standard," in *CCS*, 2018.
[76] C. P. Kohlios and T. Hayajneh, "A comprehensive attack flow model and security analysis for Wi-Fi and WPA3," 2018.
[77] J. Bellardo and S. Savage, "802.11 denial-of-service attacks: real vulnerabilities and practical solutions," in *USENIX Security*, 2003.
[78] S. M. Glass and V. Muthukkumarasamy, "A study of the TKIP cryptographic DoS attack," in *International Conf. on Networks*. IEEE, 2007.
[79] B. Könings, F. Schaub, F. Kargl, and S. Dietzel, "Channel switch and quiet attack: New DoS attacks exploiting the 802.11 standard," in *LCN*, 2009.
[80] M. Schulz, F. Gringoli, D. Steinmetzer, M. Koch, and M. Hollick, "Massive reactive smartphone-based jamming using arbitrary waveforms and adaptive power control," in *WiSec*, 2017.
[81] K. Bicakci and B. Tavli, "Denial-of-service attacks and countermeasures in IEEE 802.11 wireless networks," *Comput. Stand. Interfaces*, vol. 31, no. 5, 2009.
[82] W. Aiello, S. M. Bellovin, M. Blaze, J. Ioannidis, O. Reingold, R. Canetti, and A. D. Keromytis, "Efficient, DoS-resistant, secure key exchange for internet protocols," in *CCS*, 2002.
[83] S. Fluhrer, "Re: [Cfrg] I-D for password-authenticated EAP method," Retrieved 9 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg02206.html, Feb. 2008.
[84] D. Clarke and F. Hao, "Cryptanalysis of the dragonfly key exchange protocol," *IET Information Security*, vol. 8, no. 6, pp. 283–289, 2014.
[85] E. Alharbi, N. Alsulami, and O. Batarfi, "An enhanced Dragonfly key exchange protocol against offline dictionary attack," *Journal of Information Security*, vol. 6, no. 02, p. 69, 2015.
[86] S. M. Bellovin and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks," in *IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, 1992.
[87] M. Steiner, G. Tsudik, and M. Waidner, "Refinement and extension of encrypted key exchange," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 3, pp. 22–30, 1995.
[88] D. P. Jablon, "Strong password-only authenticated key exchange," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 5, pp. 5–26, 1996.
[89] T. D. Wu *et al.*, "The secure remote password protocol." in *NDSS*, vol. 98. Citeseer, 1998, pp. 97–111.
[90] S. Shin, K. Kobara, and H. Imai, "Security proof of AugPAKE." *IACR Cryptology ePrint Archive*, vol. 2010, p. 334, 2010.
[91] S. V. Smyshlyaev, I. B. Oshkin, E. K. Alekseev, and L. R. Ahmetzyanova, "On the security of one password authenticated key exchange protocol," Cryptology ePrint Archive, Report 2015/1237, 2015, https://eprint.iacr.org/2015/1237.
[92] S. Jarecki, H. Krawczyk, and J. Xu, "Opaque: An asymmetric pake protocol secure against pre-computation attacks," Cryptology ePrint Archive, Report 2018/163, 2018, https://eprint.iacr.org/2018/163.
[93] M. Abdalla and D. Pointcheval, "Simple password-based encrypted key exchange protocols," in *CT-RSA*. Springer, 2005, pp. 191–208.
[94] J. Becerra, D. Ostrev, and M. Škrobot, "Forward secrecy of SPAKE2," in *International Conference on Provable Security (ProvSec)*. Springer, 2018.
[95] T. Wu, "The SRP authentication and key exchange system," RFC 2945, Sep. 2000.
[96] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin, "Using the secure remote password (SRP) protocol for TLS authentication," RFC 5054, Sep. 2007.
[97] S. Shin and K. Kobara, "Efficient augmented password-only authentication and key exchange for IKEv2," RFC 6628, Jun. 2012.
[98] S. Smyshlyaev, E. Alekseev, I. Oshkin, and V. Popov, "The security evaluated standardized password-authenticated key exchange (SES-PAKE) protocol," RFC 8133, Mar. 2017.
[99] F. Hao, "J-PAKE: Password-authenticated key exchange by juggling," RFC 8236, Sep. 2017.
[100] W. Ladd and B. Kaduk, "SPAKE2, a PAKE," Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-spake2-07, Nov. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-spake2-07
[101] J. Ding, S. Alsayigh, J. Lancrenon, R. Saraswathy, and M. Snook, "Provably secure password authenticated key exchange based on RLWE for the post-quantum world," in *CT-RSA*. Springer, 2017, pp. 183–204.
[102] X. Gao, J. Ding, L. Li, S. RV, and J. Liu, "Efficient implementation of password-based authenticated key exchange from RLWE and post-quantum TLS," Cryptology ePrint Archive, Report 2017/1192, 2017, https://eprint.iacr.org/2017/1192.
[103] M. Naor, B. Pinkas, and E. Ronen, "How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior," *IACR Cryptology ePrint Archive*, vol. 2018, p. 3, 2018.

APPENDIX A
EXPERIMENTS

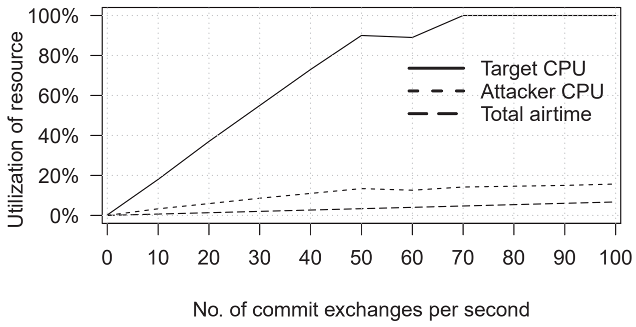


Fig. 5: DoS attack against an AP from vendor A using curve P-256. The attacker uses a Raspberry Pi 1 B+, and its CPU usage is shown in the small dashed line. The long dashed line shows the airtime consumed by all SAE frames.

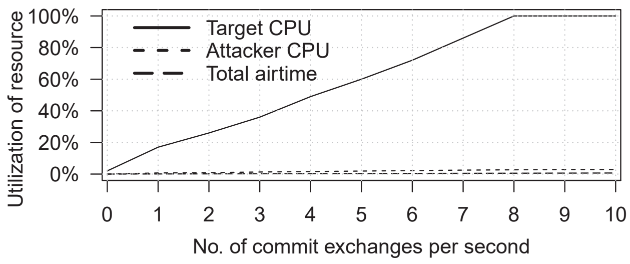


Fig. 6: DoS attack against an AP from vendor A using curve P-521. The attacker uses a Raspberry Pi 1 B+, and its CPU usage is shown in the small dashed line. The long dashed line shows the airtime consumed by all SAE frames.

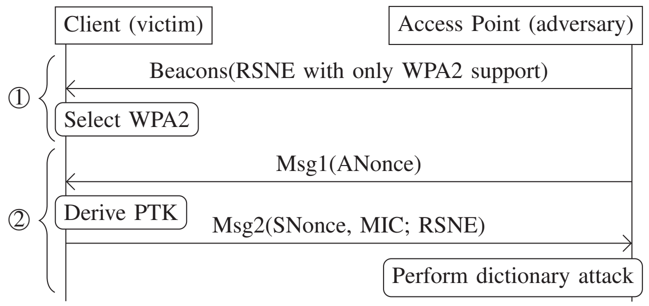


Fig. 7: Dictionary attack against WPA3-SAE when it is operating in transition mode, by attempting to downgrade the client into directly using WPA2's 4-way handshake.

APPENDIX B
SOURCE CODE

Listing 4: Side-channel protected quadratic residue test.

```c
static int is_quadratic_residue_blind(
      struct sae_data *sae, const u8 *prime, size_t bits,
      const struct crypto_bignum *qr,
      const struct crypto_bignum *qnr,
      const struct crypto_bignum *y_sqr)
{
   struct crypto_bignum *r, *num;
   int r_odd, check, res = -1;

   /* Use the blinding technique to mask y_sqr while
    * determining whether it's a quadratic residue mod p
    * to avoid leaking timing info while determining
    * the Legendre symbol.
    *   v = y_sqr
    *   r = a random number between 1 and p-1, inclusive
    *   num = (v * r * r) modulo p
    */
   r = get_rand_1_to_p_1(prime, sae->tmp->prime_len,
                          bits, &r_odd);
   ...
   if (r_odd) {
      /* num = (num * qr) module p
       * LGR(num, p) = 1 ==> quadratic residue */
      if (crypto_bignum_mulmod(num, qr, sae->tmp->prime,
                                 num) < 0)
         goto fail;
      check = 1;
   } else {
      /* num = (num * qnr) module p
       * LGR(num, p) = -1 ==> quadratic residue */
      if (crypto_bignum_mulmod(num, qnr, sae->tmp->prime,
                                 num) < 0)
         goto fail;
      check = -1;
   }
   res = crypto_bignum_legendre(num, sae->tmp->prime);
   ...
   res = res == check;
   ...
```

Listing 5: Verification of the KDF output in the patched version of hash-to-curve.

```c
static int sae_test_pwd_seed_ecc(struct sae_data *sae,
      const u8 *pwd_seed, const u8 *prime, const u8 *qr,
      const u8 *qnr, u8 *pwd_value)
{
   ...
   if (sha256_prf_bits(pwd_seed, SHA256_MAC_LEN,
                       "SAE Hunting and Pecking", prime,
                       sae->tmp->prime_len, pwd_value,
                       bits) < 0)
      return -1;
   ...
   if (const_time_memcmp(pwd_value, prime,
                          sae->tmp->prime_len) >= 0)
      return 0;
   x_cand = crypto_bignum_init_set(pwd_value,
                                    sae->tmp->prime_len);
   ...
}
```

Listing 6: SAE password derivation using hash-to-curve.

```c
static int sae_derive_pwe_ecc(
      struct sae_data *sae, const u8 *addr1,
      const u8 *addr2, const u8 *password,
      size_t password_len, const char *identifier)
{
   ...
   if (random_get_bytes(dummy_password,
                        dummy_password_len) < 0)
      return -1;
   ...
   /* Create a random quadratic residue (qr) and quadratic
    * non-residue (qnr) mod p for blinding purposes during
    * the loop.
    */
   if (get_random_qr_qnr(prime, prime_len, sae->tmp->prime,
                        bits, &qr, &qnr) < 0)
         return -1;
   ...
   /* Continue for at least k iterations to protect against
    * side-channel attacks that attempt to determine the
    * number of iterations required in the loop.
    */
   for (counter = 1; counter <= k || !x; counter++) {
      ...
      res = sae_test_pwd_seed_ecc(sae, pwd_seed, prime
                                  qr, qnr, &x_cand);
      if (res < 0)
         goto fail;
      if (res > 0 && !x) {
         ...
         x = x_cand; /* saves the current x value */
         ...
         /* Use a dummy password for the following
          * rounds, if any. */
         addr[0] = dummy_password;
         len[0] = dummy_password_len;
      } else if (res > 0) {
         crypto_bignum_deinit(x_cand, 1);
      }
   }
   ...
```

Listing 7: Assembly output of SAE's hash-to-curve method.

```
000000000002efe0 <sae_derive_pwe_ecc>:
...
2f2c8:  e8f3170500     callq  80ac0 <sha256_prf_bits>
...

2f719:  e8f2fa0400     callq  7f210 <crypto_bignum_legendre>
...
2f751:  e81af70400     callq  7ee70 <crypto_bignum_deinit>
2f75d:  0f8559010000   jne
2f8bc <sae_derive_pwe_ecc+0x8dc>
...
; handle qr case code range
2f7d2:  0f8660faffff   jbe
2f238 <sae_derive_pwe_ecc+0x258>
...
; start nqr case code
2f8bc:  488b7c2440     mov    0x40(%rsp),%rdi
2f8c1:  be01000000     mov    $0x1,%esi
2f8c6:  e8a5f50400     callq  7ee70 <crypto_bignum_deinit>
2f8cb:  e994fafffff    jmpq
2f364 <sae_derive_pwe_ecc+0x384>
; end nqr case code
...
```