Dragonblood: A Security Analysis of WPA3's SAE Handshake

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 Wi-Fi networks, and provides several advantages over its predecessor WPA2, such as protection against offline dictionary attacks and forward secrecy. Unfortunately, we show that WPA3 is affected by several design flaws, and analyze these flaws both theoretically and practically. Most prominently, we show that WPA3's Simultaneous Authentication of Equals (SAE) handshake, commonly known as Dragonfly, is affected by password partitioning attacks. These attacks resemble dictionary attacks and allow an adversary to recover the password by abusing timing or cache-based side-channel leaks. Our side-channel attacks target the protocol's password encoding method. For instance, our cache-based attack exploits SAE's hash-to-curve algorithm. The resulting attacks are efficient and low cost: brute-forcing all 8-character lowercase password requires less than 125$ in Amazon EC2 instances. In light of ongoing standardization efforts on hash-to-curve, Password-Authenticated Key Exchanges (PAKEs), and Dragonfly as a TLS handshake, our findings are also of more general interest. Finally, we discuss how to mitigate our attacks in a backwards-compatible manner, and explain how minor changes to the protocol could have prevented most of our attacks.

1 INTRODUCTION

The Wi-Fi Alliance recently announced WPA3 as the more secure successor of WPA2. Unfortunately, it was created without public review, meaning experts could not critique any of WPA3's new features before they were released. Moreover, although the new handshake of WPA3 was designed in an open manner, its security guarantees are unclear. On one hand there is a security proof of a close variant of WPA3's handshake [59], but on the other hand another close variant of the handshake received significant criticism during its standardization [68, 79]. These issues raise the question whether WPA3 is secure in practice.

We remark that WPA3 does not define new protocols, but instead mandates which existing protocols a device must support. This means WPA3 is not a specification, but a certification. Put differently, devices can now become WPA3-certified, which assures they implement certain protocols in an interoperable manner. The only novelty in the WPA3 certification is a transition mode where WPA2 and WPA3 are simultaneously supported for backward compatibility (see Section 2.2). Although WPA3 follows recommended practice by using existing standards, we believe more openness to alternative protocols could have increased its security.

In this paper we perform a security analysis of WPA3's Simultaneous Authentication of Equals (SAE) handshake. This handshake is designed to prevent dictionary attacks, and constitutes the biggest improvement over WPA2. We systematically analyzed its security by reading specifications, inspecting formal proofs, and auditing open-source implementations. This analysis revealed several design and implementation flaws. For instance, when verifying the assumptions made by the formal proof of the SAE handshake [59], we discovered both timing and cache-based side-channel vulnerabilities in its password encoding method. We empirically confirmed all our findings against both open source and recently-released proprietary implementations of WPA3.

All combined, our work resulted in the following contributions:

- We provide a self-contained and high-level description of WPA3 and its SAE handshake (Section 2 and 3).
- We show that the anti-clogging mechanisms of SAE is unable to prevent denial-of-service attacks (Section 4). In particular, by abusing the overhead of SAE's defenses against already-known side-channels, a resource-constrained device can overload the CPU of a professional Access Point (AP).
- We present a dictionary attack against WPA3 when it is operating in transition mode (Section 5). This is accomplished by trying to downgrade clients to WPA2. Although WPA2's 4-way handshake detects the downgrade and aborts, the frames sent during the partial 4-way handshake provide enough information for a dictionary attack. We also present a downgrade attack against SAE, and discuss implementation-specific downgrade attacks when a client improperly auto-connects to a previously used WPA3-only network.
- We empirically investigate the feasibility of timing attacks against WPA3's SAE handshake (Section 6). This confirms timing attacks are possible and leak info about the password.
- We present a novel micro-architectural cache-based side-channel attack against the SAE handshake (Section 7). This attack leaks information about the password being used. Our attack even works against hash-to-curve algorithm implementations that include countermeasures against side-channel attacks. This type of attack against hash-to-curve algorithms is of independent interest due to current standardization efforts surrounding hash-to-curve methods [73].
- We show both theoretically and empirically how the recovered timing and cache info can be used to perform an offline password partitioning attack (Section 8). This enables an adversary to recover the password used by the victim.

Finally, we will discuss related work in Section 9, and we give concluding remarks in Section 10.

1.1 Responsible Disclosure

We collaborated with the Wi-Fi Alliance and CERT/CC to notify all affected vendors in a coordinated manner, and helped with implementing backwards-compatible countermeasures. An overview of affected products and vendors, including allocated Common Vulnerabilities and Exposures (CVE) identifiers, can be found at [17]. We hope that our work will influence the deployment of WPA3 before it becomes widespread and hard to patch.

2 BACKGROUND

In this section we introduce the (relatively few) new features that are defined in the WPA3 certification [90], and we explain relevant aspects of the 802.11 standard [45].

2.1 An Overview of WPA3

The WPA3 certification was created with two types of networks in mind. The first one is a home network, where users authenticate using a pre-shared password, and the second one is an enterprise network, where more advanced authentication methods can be used (e.g. certificates, smart cards, and so on). To differentiate both types, the term WPA3-SAE is used for home networks, and the term WPA3-Enterprise is used for enterprise networks.

WPA3-Enterprise uses existing handshakes, but requires that ciphers used during authentication provide at least 192 bits of security. That is, the ciphersuites must use at least 384-bit curves for elliptic curve cryptography, and use at least 3072-bit moduli when using RSA or DHE. Currently the WPA3 certification does not mention requirements on the length of session keys or hash functions used after authentication [90]. However, a security level of at least 192-bits will likely also be used after authentication [18].

The WPA3-SAE mode for home networks is more interesting. It mandates support for the existing Simultaneous Authentication of Equals (SAE) handshake. This handshake is a Password Authenticated Key Exchange (PAKE), meaning authentication is performed based on a password. The SAE handshake provides forward secrecy and resistance against offline dictionary attacks, and was added to the 802.11 standard in 2011 [47]. Several variants of this handshake are also used in other protocols (see Section 3.1). The output of WPA3's SAE handshake is a Pairwise Master Key (PMK), which is subsequently used to perform a 4-way handshake to derive a Pairwise Transient Key (PTK) (see Figure 1). Note that, even though WPA3 still uses WPA2's 4-way handshake, it is not vulnerable to dictionary attacks. This is because the PMK generated by the SAE handshake has much higher entropy than the password itself.

Finally, in both modes, Management Frame Protection (MFP) must be used. Most notably, MFP prevents deauthentication attacks where an adversary forcibly disconnects victims from the AP.

2.2 WPA3-SAE Transition Mode

Because existing devices may not receive support for SAE of MFP, they will not be able to use WPA3. To accommodate these older devices, the WPA3 certification defines how a network can simultaneously support WPA2's 4-way handshake and WPA3's SAE handshake. In this transition mode, the AP advertises that MFP is optional, and that it supports both the 4-way and SAE handshake. Older WPA2 clients can then connect using the 4-way handshake without MFP, while new WPA3 clients can connect using SAE with MFP enabled. The only requirement placed on WPA3 clients is that they must use MFP when connecting to a WPA3-capable AP, even though the AP advertises MFP as optional.

2.3 Downgrade Protection

An AP advertises its supported cipher suites, i.e., authentication and encryption algorithms, in the Robust Security Network Element (RSNE). The RSNE is included unauthenticated in beacons

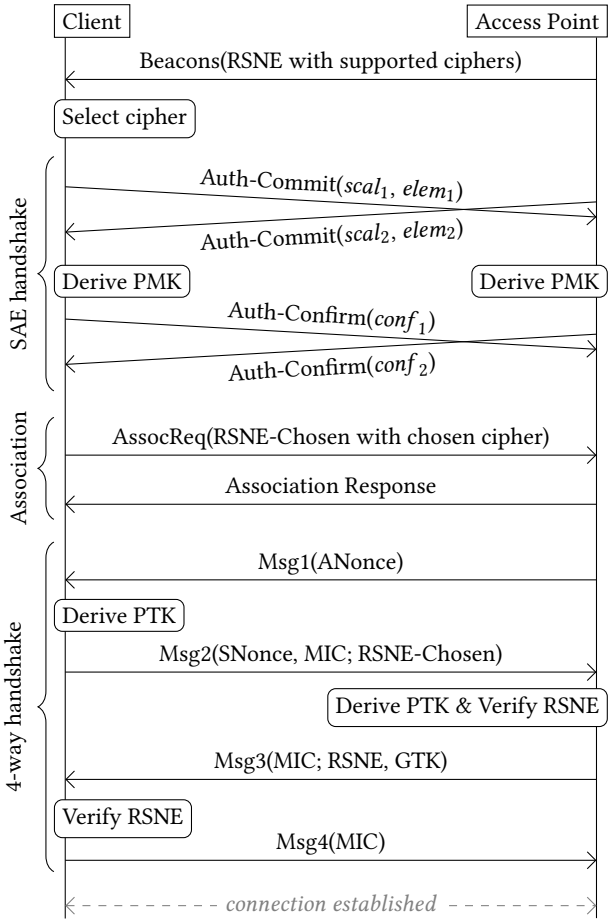


Figure 1: Connecting to an AP using WPA3. First the SAE handshake negotiates the master key (PMK), and then the 4-way handshake derives a session key (PTK). To support mesh networks, the SAE handshake was made so both parties can initiate it in parallel (hence the crossed arrows).

that are transmitted periodically to advertise the presence of a network. Clients also use the RSNE in association requests to inform the AP of the cipher suites they wish to use. Example authentication algorithms are the 4-way handshake, the 802.1X protocol, the SAE handshake, etc. Example encryption algorithms are GCMP or (AES-)CCMP. Because the RSNE is not authenticated in beacons, an adversary can spoof this element by forging beacons. To detect this, the received RSNE is cryptographically verified during WPA2's 4-way handshake. In particular, when the AP receives message 2 from the client, the AP verifies that the RSNE in the client's association request was not altered (see Msg2 in Figure 1). Similarly, when the client receives message 3 from the AP, the client verifies that the RSNE included in beacons was genuine. Since the 4-way handshake is always executed at some point when a station (i.e. a client or AP) connects for the first time to a network, the RSNE is always verified. In case a mismatch is detected, the handshake is aborted. This prevents an adversary from spoofing the RSNE, and thereby tricking the client into using a weaker cipher suite.

3 THE SAE "DRAGONFLY" HANDSHAKE

In this section we introduce WPA3's Simultaneous Authentication of Equals (SAE) handshake, and discuss side-channel defenses that were added in various revisions of the handshake.

3.1 Background and History

The WPA3 certification mandates support for the SAE handshake. This handshake was first introduced by Harkins in 2008 [33], and was added to the 802.11 standard in 2011 [47]. Several close variants of the SAE handshake are also used in other protocols. The term Dragonfly is commonly used to refer to this family of handshakes.

The SAE handshake is a balanced Password Authenticated Key Exchange (PAKE). Note that in a balanced PAKE, both endpoints store the password in the same representation (which is in plaintext when using SAE). The SAE handshake takes as input a pre-shared secret, and outputs a high-entropy Pairwise Master Key (PMK). After executing SAE, the 4-way handshake is used to negotiate a session key called the PTK (recall Figure 1). Finally, the SAE handshake explicitly supports mesh networks, by allowing both endpoints to initiate the handshake concurrently.

3.2 Protocol Details

The SAE handshake supports both Finite Field Cryptography (FFC) using multiplicative groups modulo a prime (MODP groups), and it supports Elliptic Curve Cryptography (ECC) using elliptic curve groups modulo a prime (ECP groups). The 802.11 standard mandates that if a station advertises support for SAE, it must implement the elliptic curve NIST P-256 [45, §12.4.4.1][1, 43]. Support for other groups is optional, meaning there is no mandated support for MODP groups. Therefore we assume elliptic curves are used, unless mentioned otherwise.

When describing elliptic curve operations, we use lowercase letters to denote scalars (i.e. integers), and uppercase letters to denote elliptic curve points. With SAE, all elliptic curves are defined over the equation $y^2 = x^3 + ax + b \bmod p$ where p is a prime and the values a, b, and p depend on the curve being used. We use G to denote the generator of a group, and q to denote the prime order of G. When executing the SAE handshake, the user-readable password is converted into a group element. For MODP groups this is done using a hash-to-group algorithm, and for elliptic curves using a hash-to-curve algorithm. The resulting password element is denoted by P, and its generation is described in detail in Section 3.3.

The handshake itself consists of a commit phase followed by a confirm phase. These two phases, along with the accompanying elliptic curve operations, are illustrated in Figure 2. Note that the handshake can be initiated concurrently by both participants (which may happen in mesh networks after connection loss). Nevertheless, in the more widely-used infrastructure mode, the client will initiate the handshake by sending its commit frame, and subsequently the AP will reply using a commit and confirm frame. In turn the client sends its confirm frame, completing the handshake.

In the commit phase, each participant first picks a random number $r_i \in [2, q[$ and a random mask $m_i \in [2, q[$ (see Figure 1). The sum of these numbers (modulo q) must also lie in the range $[2, q[$. They then calculate the public group element $E_i = -m_i \cdot P$, after which they send both the scalar s_i and the group element E_i to

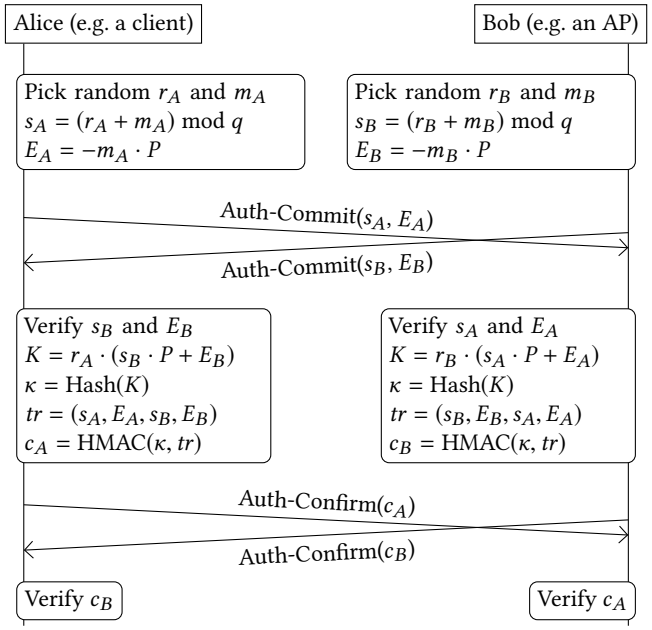


Figure 2: Details of the SAE handshake. Recall that it supports mesh networks where two stations may simultaneously initiate the handshake (hence the crossed arrows). We assume elliptic curves are used, since all implementations of SAE (and hence also WPA3) are required to support it.

the other participant using a commit frame. On reception of these values, each participant verifies that the received scalar s_i is within the range $[1, q[$, and that the received group element E_i is a valid point on the curve being used [45, §12.4.5.4]. If one of these checks fails, the handshake is aborted. Forward secrecy is provided by relying on the difficulty of deriving m_i given P and E_i, i.e., it relies on the hardness of the elliptic curve discrete logarithm problem.

In confirm phase, each participant calculates the shared secret point K (see Figure 2). The x-coordinate of this point is processed using a hash function to derive the key κ. Finally, a HMAC over the handshake summary tr is calculated using the key κ. The result of this hash, denoted by c_i, is sent to the other participant in a confirm frame. On reception of c_i, the receiver verifies its value. If it equals the expected value, the handshake succeeds, and the negotiated key κ becomes the Pairwise Master Key (PMK). Otherwise the confirm frame is ignored, and the handshake eventually times out.

For details on how the SAE handshake negotiates which cryptographic group is used, we refer to Section 5.2. The handshake also has a mechanism to prevent denial-of-service (DoS) attacks, but unfortunately this defense is flawed (see Section 4).

3.3 Password Derivation

When constructing the commit frame, the pre-shared password is first converted into a curve point using a hash-to-curve algorithm. The specific algorithm used in SAE is based on a try-and-increment method, and is shown in Listing 1. Summarized, it first hashes the password, together with a counter and the MAC addresses of both stations, and uses the output of the hash as the x-coordinate

Listing 1: Converting the pre-shared password into an elliptic curve point in Python-like pseudocode [45, §12.4.4.2.2].

```
1  def password_to_element_ecc(password, MAC1, MAC2, k=40):
2     found = False
3     counter = 0
4     base = password
5     while counter < k or not found:
6        counter += 1
7        seed = Hash(MAC1, MAC2, base, counter)
8        value = KDF(seed, "SAE Hunting and Pecking", p)
9        if value >= p: continue
10
11       if is_quadratic_residue(value^3 + a * value + b, p):
12          if not found:
13             x = value
14             save = seed
15             found = True
16             base = random()
17
18    y = sqrt(x^3 + a * x + b) mod p
19    if LSB(save) == LSB(y):
20       P = (x, y)
21    else:
22       P = (x, p - y)
23    return P
```

of the curve point. It then tries to find a solution for y over the equation $y^2 = x^3 + ax + b \bmod p$. In case a solution exists, the point (x, y) becomes the password element P. If no solution is found, the counter is increased, and another attempt is made to find a solution for y using the new value for x. Finally, to mitigate timing attacks, the main loop is always executed k times, no matter when a solution for y is found. In the extra iterations, calculations are based on a randomly generated password instead of the real one.

Interestingly, this algorithm underwent various changes, and all these changes attempt to reduce side-channel leaks. In particular, the original 802.11s amendment that introduced SAE did not include the additional iterations [47, §8.2a.4.2.2]. Put differently, originally the algorithm immediately stops when a solution for y is found. An amendment proposed in 2011 added the extra iterations to mitigate timing attacks [34], and this amendment was incorporated into the 2012 version of the 802.11 standard [46, §11.3.4.2.2].

Sensitive information can also leak in the Legendre function on line 11 which checks if the left-hand side of the curve equation is a quadratic residue. If it is a quadratic residue, a solution for y exists. However, depending on how the Legendre function is implemented, its running time may leak information [44]. To prevent timing attacks, an amendment to 802.11 recommends (but does not mandate) the use of quadratic residue blinding [24, 36]. This amendment got incorporated into the 2016 version of 802.11 [45, §12.4.4.2.2].

We also note that the size of the counter value is only 1 byte. However, because the probability of finding a solution for y is approximately 50%, the chance of the counter overflowing and causing an infinite loop is only 2^{-255}. Even an adversary who forges 2^{63} unicast MAC addresses in an attempt to cause an overflow, only has a chance of 2^{-192} of triggering an overflow and infinite loop.

Another critical remark is that the 802.11 standard does not specify a minimum value for k. This parameter denotes the total number of iterations that must always be executed to mitigate timing attacks. In practice, version 2.1 to 2.4 of wpa_supplicant and hostapd use $k = 4$, while newer versions use $k = 40$. This increase was based on security advice given in a close variant of the SAE handshake [60]. However, this update was not backported to older versions, meaning products using a version that use $k = 4$ are vulnerable to timing attacks (see Section 6.5). Intel's iwd client uses the value $k = 20$, and the reference implementation of SAE by Harkins uses value 40 [32]. For comparison, the Dragonfly specification in RFC 7664, which is a close variant of SAE, explicitly recommends a value of at least $k = 40$ [37, §4]. The value 40 is based on a back-of-the-envelope calculation by Igoe [50].

Given the above history of timing side-channels and defenses, one may wonder why an alternative design was not used that avoids side-channel attacks. In particular, during its review by the CFRG, suggestions were made to exclude the MAC addresses from the hash-to-curve algorithm [48, 49, 69, 80]. With this modification, the curve point can be generated offline, and can then be reused in every connection attempt. This makes side-channel attacks significantly harder, since the password element is generated only once.

3.4 Variants of Dragonfly

Several minor variants of the SAE handshake exist, and the resulting family of handshakes is often referred to as Dragonfly [35, 37–39]. Note that the RFCs describing these variants are not standards-track RFCs, instead, they are either informational or experimental ones. This means they are independent submissions, and not officially endorsed by e.g. the Internet Engineering Task Force (IETF). For example, there was no consensus in the TLS working group to adopt the TLS-PWD variant of Dragonfly [71]. To the best of our knowledge, only the 802.11 standard officially adopted Dragonfly.

In all variants, the password element must be computed online, because it depends on nonces or on the identities of the participants. For example, the TLS variant of Dragonfly specified in RFC 8492 includes random nonces from the client and server in the hash-to-curve algorithm. As a result, the password element must be computed online, meaning implementations may be affected by the same timing and cache-based side-channel attacks we will present in this paper. Similarly, implementations of TLS-PWD will also have a large overhead due to the required side-channel countermeasures.

4 ABUSING SAE'S SIDE-CHANNEL DEFENSES

In this section, we show how side-channel defenses of SAE (against already-known leaks), introduce overhead that can be abused in a denial-of-service (DoS) attack. Simultaneously, we bypass SAE's anti-clogging mechanism that is supposed to prevent DoS attacks.

4.1 Background on (Anti-)Clogging Methods

All versions of the SAE handshake contain an anti-clogging method to mitigate DoS attacks. This is needed because an AP must perform costly operations when receiving a commit frame, which an adversary can otherwise abuse in a DoS attack by forging commit frames [47, §8.2a.6]. This problem is worse in amended versions of SAE that contain side-channel defenses against (already-known) information leaks, as these defenses further increase the cost of processing commit frames. More precisely, using quadratic residue blinding, and a fixed number of 40 iterations in the hash-to-curve algorithm, further increases processing time. For example, when hostapd added quadratic residue blinding, the increased processing

time even caused timeouts on resource constrained devices [42]. To prevent timeouts, the 802.11 standard was updated to give stations 2 seconds (instead of 40 ms) to process commit frames [9].

The anti-clogging defense of SAE consists of a cookie exchange procedure. Simplified, in this procedure the client must reflect a random (secret) cookie sent by the AP, before the AP will process the client's commit frame. This is inspired by anti-clogging defenses from IP-based networks. More precisely, such a defense was also used by a precursor of IPsec called Photuris, and a similar variant of this cookie mechanism is also part of IKEv2 [55]. How a cookie is generated is implementation dependent, but it must satisfy the following requirements [54]: (1) the cookie depends on the identities of both parties; (2) only the responder (server) can generate valid cookies; and (3) the cookie generation and verification must be fast. The recommended method to meet these requirements, is to generate a secret value, and calculate the cookie as follows:

$$\text{Cookie} = \text{Hash}(\text{ConnectionID} \parallel \text{InitiatorID} \parallel \text{secret})$$

The hash function should be a secure one-way hash such as SHA256. With this cookie exchange, an adversary can no longer initiate handshakes using spoofed IP addresses. This prevents an adversary from, for example, trigger expensive public key operations by forging frames with spoofed IP addresses [65]. Although the attacker can still use its real IP address in forged frames, these requests can be throttled based on the IP-address. Additionally, because an attacker must use its real IP address, the threat of attacks is reduced [65].

4.2 Defeating SAE's Anti-Clogging

The SAE handshake of WPA3 also uses a cookie exchange procedure to mitigate clogging attacks. More precisely, this mechanism is supposed to prevent DoS attacks that flood the victim with bogus SAE commit messages from forged MAC addresses [45, §12.4.6]. However, in contrast to spoofing IP addresses, it is trivial to spoof MAC addresses. Even if the AP employs a cookie exchange mechanism, the adversary can trivially capture all cookies (i.e. anti-clogging tokens), since everyone within range of the AP can capture and reflect the secret cookies. More generally, a cookie exchange mechanism can trivially be defeated in any broadcast network, since everyone will receive the (supposedly secret) cookies.

4.3 Experiments

We implemented a Proof-of-Concept (PoC) of a clogging attack where the adversary inject commit frames using spoofed MAC addresses, and reflects any cookies (i.e., anti-clogging tokens) it receives. Out tool is written in C for performance reasons, and build on top of aircrack-ng. The tool can forge commit frames using any elliptic curve supported by SAE. It is essential that the adversary acknowledges all frames sent to forged MAC addresses. Otherwise, the AP will retransmit replies up to eight times, making it difficult for the adversary to inject enough commit frames to overload the target. Fortunately, by relying on the virtual Wi-Fi interface support of Atheros chips, we can easily make it acknowledge frames sent to any forged MAC address [84, §5.3].

In our clogging experiments, the adversary used a Raspberry Pi Model B+ having a 700 MHz processor. The Raspberry Pi used a WNDA3200 wireless dongle. Our target was the professional AP from vendor A, which has a 1200 MHz processor. In our first experiment, the attack is performed using curve P-256. We found that spoofing more than 70 commit exchanges every second causes the CPU usage of the AP to reach 100% (see Figure 3). As a result, clients that try to connect using WPA3 either face long delays, or cannot connect at all. In contrast, the CPU usage of the attacker is only 14.2%. Since all APs are required to support NIST curve P-256, this shows an adversary can perform denial-of-service attacks using cheap resource-constrained devices.

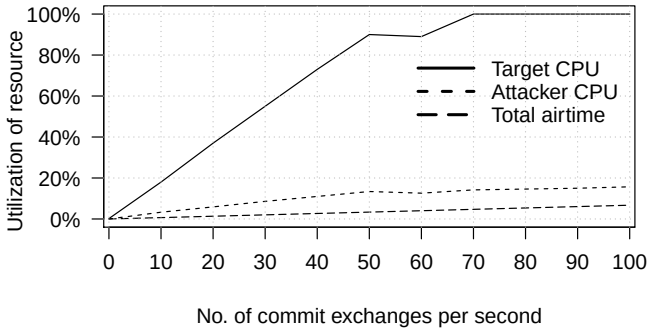


Figure 3: Clogging attack against a professional AP from vendor A using curve P-256. The attacker uses a Raspberry Pi 1 model B+, and its CPU usage is shown in the small dashed line. The total amount of airtime consumed by all SAE frames is shown in the long dashed line.

In a second experiment, we forged commit exchanges using the P-521 curve. With this curve the impact is an order of magnitude more catastrophic. Now the target's CPU can be overloaded by forging merely 8 commit exchanges every second (see Figure 8 in the Appendix). On our Raspberry Pi, this attack consumes 2.7% of the CPU. In other words, a weak adversary is able to clog a high-end AP. We consider it worrying that such a devastating attack is possible against a modern security protocol.

Figure 3 and 8 also show the amount of airtime consumed by the injected frames. We observe that this is just a fraction of the total available airtime, showing that our attack is more efficient than a straightforward DoS where an attacker simply jams the channel.

4.4 Attack Optimizations

Our clogging attack can be optimized if we take into account how APs generate the anti-clogging token. Note that 802.11 standard recommends to generate anti-clogging tokens by computing a hash over a secret value and the MAC address of the sender. All devices we inspected indeed follow hashing-based mechanism. Additionally, the 802.11 standards recommends to track the number of open and unfinished handshakes. When this number hits a given threshold, the secret value should be renewed [45, §8.2a.6]. This implies that when the number of open and unfinished handshakes is always above this threshold, the threshold itself will not be hit, meaning the secret value is not updated.

If the secret value is infrequently updated, we can reuse previously captured anti-clogging tokens. In particular, we were able to inspect the devices shown in Table 1. The first one is an AP from vendor A, which generates anti-clogging tokens as recommended

Table 1: Renewal interval of the secret that is used to generate anti-clogging tokens for various devices and standards.

Standard or Implementation	Version	Renewal time
AP from vendor A	10.20.0168	threshold reached
Hostapd	v2.6	every 1 minute
Reference Implementation	May 2014	never renewed
Phorious [54]	1999	every 1 minute
IKEv2 [55]	2014	not specified
802.11 [45]	2016	threshold reached

by the standard, meaning it is easy to bypass. The second implementation we tested is the open source hostapd. It also generates anti-clogging tokens as recommended by the standard, except that it renews the secret value every minute. This means we can reuse anti-clogging token for one minute. Against hostapd we also discovered that, if an adversary keeps forging commit frames using the same MAC address, the AP will constantly (re)process the frame without requiring anti-clogging tokens, leading an an efficient DoS. Harkins' reference implementation of SAE also follows the 802.11 standard, but it never renews the secret value [32]. Table 1 also shows when other protocols such as IKEv2 renew the secret value.

4.5 Countermeasures

To reduce the impact of an attack, the derivation of the password element can be done in a low-priority background thread. Although legitimate WPA3 clients will be unable to connect during an attack, this at least assures other network functionality is not impacted. Additionally, larger elliptic curves or MODP groups can be disabled by default, to reduce the impact of DoS attacks.

A better solution is to use a constant-time and more efficient hash-to-curve algorithm [73]. Another solution would be to modify the SAE handshake such that the password element is independent of the MAC addresses. Doing this would allow both the client and AP to calculate the password element offline, and reuse it in all handshakes, preventing our attack. However, SAE's security proof must then be updated to take this change into account [59], to verify whether the protocol would remain secure with this change.

5 DOWNGRADE & DICTIONARY ATTACKS

In this section we present a dictionary attack against WPA3-SAE when it is operating in transition mode, and discuss an implementation-specific downgrade attack against WPA3-only networks. We also present a downgrade attack against the SAE handshake itself.

5.1 Downgrade to Dictionary Attack

Our first attack is against WPA3-SAE transition mode. Recall from Section 2.2 that in this mode the AP is configured to accept connections using both WPA3-SAE and WPA2. This provides backward compatibility with older clients. Moreover, WPA2's 4-way handshake detects downgrade attacks, meaning an attacker cannot trick a WPA3-capable client into successfully establishing a connection using WPA2. Put differently, if an adversary attempts to perform a man-in-the-middle against a WPA3-capable AP and client, and

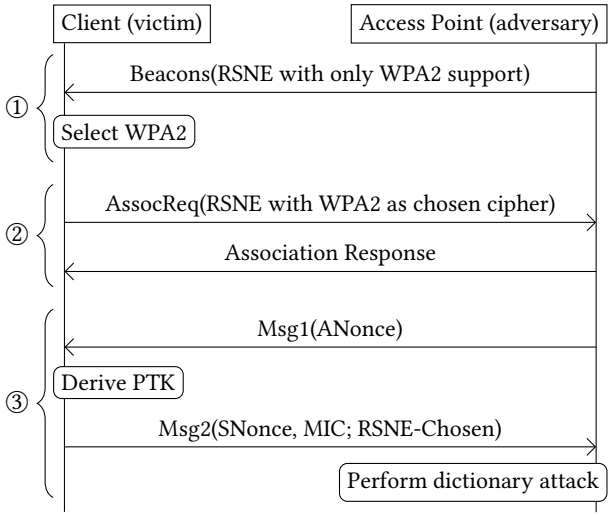


Figure 4: Dictionary attack against WPA3-SAE when it is operating in transition mode, by attempting to downgrade the client into directly using WPA2's 4-way handshake.

modifies beacons so the client thinks the AP only supports WPA2, the client will detect the downgrade and abort the 4-way handshake of WPA2. More precisely, message 3 of WPA2's 4-way handshake contains all the supported cipher suites of the AP in the authenticated RSNE element (recall Figure 1). Because this handshake message is authenticated under the session key (PTK), the adversary cannot modify it. As a result, the client will detect that RSNE in message 3 does not match with the RSNE received in beacons, and will subsequently abort the handshake. Hence it is indeed not possible to force a WPA3-capable client and AP to use WPA2.

The problem is that, although downgrade attacks are detected by the 4-way handshake of WPA2, by that point an adversary has captured enough data to perform a dictionary attack. This is because an adversary only needs a single authenticated 4-way handshake message to carry out a dictionary attack [62]. Therefore, even though the downgrade is detected, by this point it is too late. Moreover, a man-in-the-middle position is not needed to carry out the attack. The only requirements are that we know the SSID of the WPA3-SAE network, and that we are close to a client (see Figure 4). If these conditions are met, the adversary can broadcast a WPA2-only network with the given SSID (stage ① in Figure 4). 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 yet authenticated (stage ③ in Figure 4). 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 [62].

We tested the above attack against several client-side implementations of WPA3 (see Table 2). With the first three tested devices, the network to connect with must be manually configured. That is, we had to specify the name of the network to connect with, and that it uses WPA3 in transition mode. We then let this device connect to the WPA3 network, after which we put up a rogue WPA2 AP. This revealed that these three devices tried to connect to the WPA2

Table 2: Result of downgrade attacks against WPA3 clients when the AP operates in transition mode (column Trans) or in WPA3-only mode (column 3-Only). On the first 3 devices the network must be configured manually, while on other devices the network is selected from a list of nearby ones.

Device	Software	Vulnerable?	
		Trans.	3-Only
AP from vendor A	firmware 10.20.0168	✓	✗
RaspBerry Pi 1 B+	OpenWRT r9576	✓	✗
MSI GE60 Laptop	wpa_supplicant v2.7	✓	✗
MSI GE60 Laptop	iwd v0.14	✓	✓
Dell Latitude 7490	NetworkManager 1.17	✗	✗
Google Pixel 3	QPP1.190205.018.B4	✗	✗
Galaxy S10	G975USQU1ASBA	✓	✓

network, allowing subsequent dictionary attacks. With the last four devices in Table 2, the desired network is selected from a list of nearby ones. We found that iwd and the Galaxy S10 are vulnerable. However, Linux's NetworkManager and the Google Pixel 3 refused to connect to the rogue WPA2 network, preventing our attack.

We also discovered an implementation-specific downgrade attack when using WPA3-only networks. More precisely, we noticed that some devices will connect to the rogue WPA2 network, even if originally the network only supported WPA3 (see column 3-Only in Table 2). In particular, iwd and the Samsung Galaxy S10 are affected by this attack, meaning downgrade to dictionary attacks remain possible even if the network is configured to only support WPA3.

5.2 Attacking SAE's Group Negotiation

The SAE handshake can be run using different elliptic curve or multiplicative groups mod p (i.e. ECP or MODP groups). The "Group Description" of [43] gives an overview of supported groups. Additionally, the 802.11 standard allows station to prioritize groups in a user-configurable order [45, §12.4.4.1]. Although this provides flexibility, it requires a secure method to negotiate the group that will be used. Unfortunately, the mechanism that negotiates which group or curve is used during the SAE handshake is trivial to attack.

With SAE, the used group is negotiated as follows. When a client connects to an AP, it includes its desired group in the commit frame, along with a valid scalar s_i and element E_i. In case the AP does not support this group, it will reply using a commit frame with a status field equal to "unsupported finite cyclic group" (see stage ① in Figure 5). In turn the client will attempt to use its next preferred group, and send a new commit frame with this group, and corresponding new scalar s_i and element E_i. This process continues until the client selected a curve that the AP supports. Unfortunately, there is no mechanism that detects if someone interfered with this process. This makes it trivial to force the client into using a different group: simply forge a commit frame that indicates the AP does not support the currently selected group.

Figure 5 illustrates the resulting downgrade attack. Here the client first construct a commit frame requesting group 21 (i.e. curve P-521). However, the adversary blocks this frame from arriving

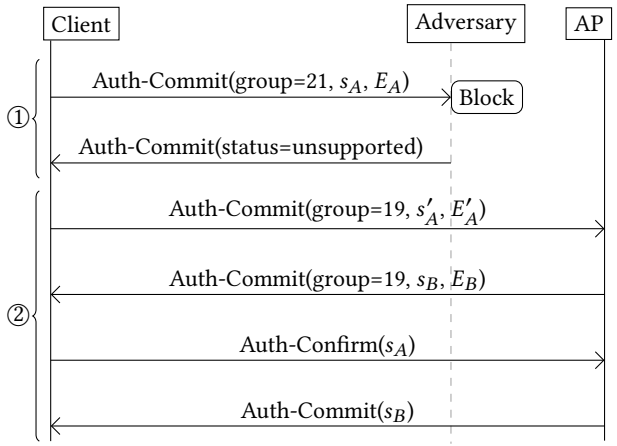


Figure 5: Downgrade attack against SAE's group selection: a man-on-the-side can force the client (initiator) into using a different cryptographic group during the SAE handshake.

at the AP (see stage ① in Figure 5). This can be accomplished by jamming the frame [84], or by forging channel-switch announcements [83]. The adversary then forges a commit frame that indicates the AP does not support the request group. In response, the client will pick its second preferred group, which in our example is group 19 (i.e. curve P-256). From this point onwards, a normal SAE handshake is executed using group 19 (see stage ② in Figure 5). Notice that this negotiation process is never cryptographically validated, meaning the downgrade attack is not detected.

It is also possible to perform an upgrade attack, where the victim is forced to use a more secure cryptographic group. That is, if the victim prefers small cryptographic groups, our attack can force the victim into using bigger groups. This may be useful when performing denial-of-service attacks (recall Section 4), or to amplify timing side-channels (see Section 6).

5.3 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 this network using a weaker handshake. This trust-on-first-usage idea is similar to the one of SSH, and similar to the Strict-Transport-Security header of HTTPS [63]. Notice from Table 2 that Linux's NetworkManager and the Google Pixel 3 already employ a similar defense. Optionally, in case the client notices the security configuration of the network changes, the client can prompt the user for the password of the network. 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 could be added to the RSNE that indicates some APs only support WPA2, meaning downgrade attacks cannot be prevented against this network.

Another possible defense, which requires minimal modifications on clients or APs, would be to deploy separate networks with separate passwords for both WPA2 and WPA3.

Listing 2: Algorithm that converts the pre-shared password into a MODP group element [45, §12.4.4.3.2]. The variables (p, G, q) define the MODP group being used, with p a prime, G a generator, and q the (prime) order of $G \bmod p$.

```
1  def password_to_element_ffc(password, MAC1, MAC2, k=40):
2    found = False
3    counter = 0
4    while not found:
5      counter += 1
6      seed = Hash(MAC1, MAC2, password, counter)
7      value = KDF(seed, "SAE Hunting and Pecking", p)
8      if value >= p: continue
9
10     P = value^((p-1)/q) mod p
11     if P > 1: found = True
12   return P
```

In principle, group downgrade attacks can also be mitigated by remembering which groups a network supports. However, the supported groups of an AP are more likely to change over time, and therefore we do not recommend such a defense. Instead, the supported groups can be included as a bitmap 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.

6 TIMING ATTACKS ON MODP GROUPS

In this section we empirically show that the hash-to-group method that converts a password into a MODP element is vulnerable to timing attacks. The obtained info will later on be used in password partitioning attacks, allowing one to recover the victim's password.

6.1 Background

Up to this point, we assumed the SAE handshake is executed using elliptic curves. Although this is a natural assumption, since any station that supports SAE must implement elliptic curve P-256, the SAE handshake can also be performed using multiplicative groups mod a prime p (MODP groups). When employing MODP groups, the algorithm in Listing 2 is used to convert the password into a group element. In contrast to the algorithm for elliptic curves, the one for MODP groups does not employ any side-channel defenses such as performing extra iterations [45, §12.4.4.3.2].

Although elliptic curves are generally more performant than MODP groups, this is not necessarily the case with SAE. In particular, due to the extra iterations needed in the hash-to-curve method, the hash-to-group method for MODP groups may be slightly more efficient. Recall from Section 3.3 that these extra iterations are needed to mitigate timing side-channels when using elliptic curves. As a result, users may prefer MODP groups over elliptic curves, especially because this would also reduce the impact of our clogging attack of Section 4. Unfortunately, we show that the hash-to-group method for MODP groups is also affected by timing side-channels. In practice this means that for certain MODP groups, extra iterations must be performed in order to mitigate timing attacks.

6.2 Variable Number of Iterations

When converting a password to a MODP element, the algorithm in Listing 2 performs a variable number of iterations. In fact, the CFRG already warned about this when they were reviewing a close variant of Dragonfly [23]. Unfortunately, countermeasures against this timing leak were not incorporated into the SAE handshake. We will now analyze what the practical impact of this decision is, and we will determine whether this can be exploited in practice.

Table 3: Overview of MODP groups that cause timing side-channels when deriving the password element. The third column shows the probability that an extra iteration is needed. The last column denotes the average number of iterations that are needed to derive the password element.

Group ID [43]	len(p)	$\Pr[value \geq p]$	$E[X]$
22	1024	30.84%	1.44
23	2048	32.40%	1.48
24	2048	47.01%	1.89

The first cause of extra iterations is when the output of the Key Derivation Function (KDF) on line 7 returns a number bigger than the prime p of the MODP group. Note that the number of bits returned by KDF is equal to the number of bits needed to represent p. That is, the number of bits returned by the KDF function depends on the MODP group being used. This also implies that the probability that *value* is bigger than p depends on the MODP group being used. Fortunately, for most MODP groups this probability is extremely small, because the prime p is only slightly smaller than a power of two. However, for the MODP groups shown in Table 3, the probability that the output of KDF is bigger than p is high. For example, for group 22 this probability equals 30.84%, and for group 24 the probability is 47.01% (see column 3 in Table 3). Finally, the if-condition on line 11 can in principle also cause an extra iteration to be executed. However, for all supported MODP groups, the probability of this happening is negligible.

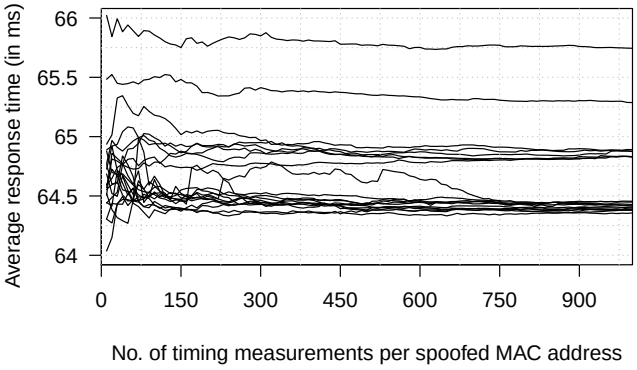
Since the output of the KDF depends on the password, the number of performed iterations also depends on the password being used. If an adversary learns this number, they learn that passwords which require a different number of iterations are not used by the victim. Note that the number of executed iterations X follows a geometric distribution:

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

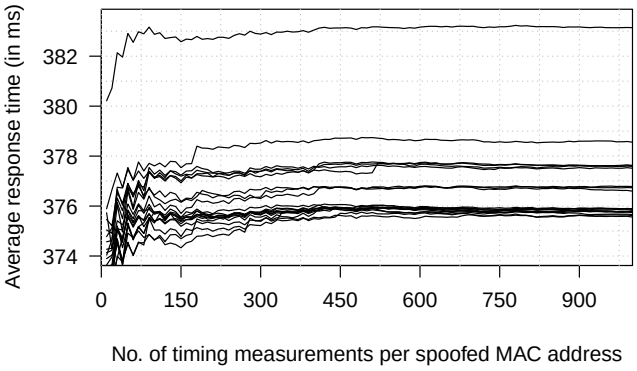
This means that the average number of iterations required to derive the password element equals $E[X] = (1 - \Pr[value \geq p])^{-1}$. For MODP group 22, this equals 1.45, and for group 24 this equals 1.89 iterations. In other words, on average one timing measurement allows the adversary to learn the result of multiple iterations. Finally, observe that the MAC address of the client also influences the output of the KDF, and hence also influences the number of executed iterations (line 6 in Listing 2). This means that an adversary can spoof MAC addresses, and for each address measure the number of executed iterations. We will show in Section 8 how this information can be used to perform a password partitioning attack, allowing an adversary to recover the target's password.

6.3 Experiments

To determine the feasibility of measuring the number of execution iterations, we performed the attack in a realistic setting. For the



(a) Measurements of a timing attack using MODP group 22.



(b) Measurements of a timing attack using MODP group 24.

Figure 6: Recovering the number of iterations needed to generate the password element for MODP group 22 and 24. Each line denotes a spoofed MAC address. The lowest cluster of lines corresponds to a single iteration, the second cluster to two iterations, and so on. The victim device is a Raspberry Pi 1 model B+ running hostapd version 2.6.

victim we used a Raspberry Pi 1 model B+ that was running hostapd version 2.6. We used a Raspberry Pi because its 700 MHz CPU matches the one in commodity home routers [91]. The Raspberry Pi was equipped with a WNDA3200 Wi-Fi dongle. Picking hostapd to run the AP was an obvious choice, since it is the most widely used wireless daemon in both professional and home routers, and is the only one that supports MODP groups at the time of writing.

The adversary used a MSI GE60 Laptop with a WNDA3200 Wi-Fi dongle. To perform timing measurements, we wrote a tool on top of the aircrack-ng tool suite. It spoofs commit frames, and measures how long it takes to receive the corresponding commit reply. After each individual measurement, a deauthentication packet is injected towards the AP. This causes hostapd to clear all state related to the spoofed MAC address, and enables us to rapidly perform a new timing measurement with the same spoofed address.

Two optimizations were required to make the attack practical. First, similar to our clogging attack of Section 4.3, we had to use virtual interface support of Atheros chips to acknowledge all frames sent to a spoofed MAC address. This prevents the AP from retransmitting frames, making the attack faster and more reliable. More importantly, background Wi-Fi traffic influences the timing measurements. Additionally, periodic background tasks on the AP also influence the timing measurements. Both sources of noise are problematic because they are not constant throughout the attack. To handle this, we interleave the time measurements of all spoofed MAC addresses, instead of performing all measurements for each MAC address one by one. As a result, temporary noise equally influences the measurements of all MAC addresses, instead of only affecting the measurements of one address.

With the above setup and optimizations, we carried out an attack using MODP group 22, and another attack using group 24. We spoofed 20 addresses in each experiment, and performed 1000 measurements for each spoofed address. The attack against group 22 took 228 minutes, and the attack against group 24 took 607 minutes. Figure 6 shows the results of these experiments. Each line represents the average timing measurements of one spoofed MAC address. From these timings, it is straightforward to derive the number of executed iterations. For example, the cluster of lines (i.e. spoofed MAC addresses) at the bottom corresponds to one iteration. The cluster above that corresponds to two iterations, and so on. For the highest line in the MODP group 24 attack, careful inspection reveals that this corresponds to 9 iterations. The correctness of these results was confirmed by inspecting the debug output of hostapd. We conclude that timing attacks can accurately determine the number of executed iterations.

6.4 Countermeasures and Discussion

Ideally, groups 22, 23, and 24 should be disabled. Doing this is in line with RFC 8247, which recommends that implementations no longer use these groups due to, among other things, their small subgroups [64, 82]. Implementations also should not use MODP groups 1, 2, or 5 [64]. The other MODP groups use primes that are slightly smaller than a power of two, meaning it is extremely unlikely that the output of the KDF in line 7 of Listing 2 is bigger or equal to the prime p. Therefore, with these groups the password element is practically always found in the first iteration.

Another option is to exclude the MAC addresses from the hash-to-group method used by the SAE handshake. Similar to our defense from Section 4.5 against clogging attacks, this would allow implementations to calculate the password element offline. Although the algorithm would still be insecure, an adversary would no longer be able to easily trigger (and measure) executions of it.

In principle, extra iterations can also be performed after finding the MODP password element, so a fixed number of iterations are always executed. This is similar to the countermeasure for the elliptic curve case. However, we do not recommend this defense, because implementations may still be vulnerable to cache-based attacks, and because groups 22 to 24 should be avoided in general [64].

6.5 Applicability to Elliptic Curves

In practice, we discovered that version 2.1 to 2.4 of wpa_supplicant and hostapd use $k = 4$ in the hash-to-curve algorithm, while only newer versions use $k = 40$. This means that against these older versions, timing attacks are also possible when elliptic curves are

used. In particular, a random MAC address will have a probability of $2^{-4} = 6.25\%$ of requiring more than 4 iterations, in which case information about the password is leaked. Although most Linux distributions that used these older versions do not enable SAE in those builds, dedicated builds for networking devices (e.g. OpenWRT 15.05.1) did use these older versions with SAE enabled.

We also conjecture that resource-constrained devices may not implement the fixed amount of 40 iterations, due to the overhead of this countermeasure. Against such implementations, timing attacks would also be possible against the hash-to-curve algorithm.

7 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. Similar to the timing attack against MODP groups, this will later on enable an adversary to recover a target's password.

7.1 Background and Attack Goal

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 partitioning attack of Section 8 to recover the target's password. Unlike the case of the MODP groups described in Section 6, 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 [45, §12.4.4.3.2], and blinding of the underlying cryptographic calculation of the quadratic residue test [36]. 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, in many cases such pseudo-constant time implementations are still vulnerable to different types micro-architectural side-channel attacks [2, 51, 70].

7.1.1 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 [26]. Cache-based side-channel attacks exploit the state of the memory case (either instructions or data) and have been widely used to break cryptographic primitives [5, 14, 27, 67, 95].

In the different variants of the Flush+Reload attack [29, 30, 95, 97] the attacker 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. 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.

7.2 Attack Scenario

Our attack requires the ability to monitor cache access patterns on the victim machine. However, unlike many cache attacks against TLS implementations [2, 51, 70], we can also target the client side. We can run our attack from any unprivileged user-mode process (or application on android). Oren et al. [66] even showed how to perform such attacks from the browser using JavaScript code (although browsers are now implementing mitigation for these types of attacks).

For our password partitioning attack, we need to record several handshakes with different MAC addresses. We can get handshakes with different MAC addresses by targeting multiple clients in the same network (e.g. convince multiple users to download the same malicious application). If we are only able to attack one client, we can set up rogue APs with the same SSID but a spoofed MAC address. We can force victims into connecting to our rogue AP by using a higher signal strength, or jamming the legitimate AP [84].

7.3 Attacking the hostap Implementation

Our target implementation is the `sae_derive_pwe_ecc` function in the latest hostap code (commit 0eb34f8f2 from Sat Jan 26) 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. To monitor access to the instruction cache we use the Flush+Reload attack [95], as implemented in the Mastik toolkit [94].

To learn the result of the first QR test, we can either attack the blinded QR test implementation, or the branch in the iteration loop that checks the result of the test. A simple cache attack against the blinded QR test is infeasible as the two possible code paths (see Listing 4 line 19) are compiled into a single cache line.[1]

The two code paths of the branch inside the iteration loop (see Listing 5 line 28) are compiled into two separate cache lines. Therefore we can monitor cache access to nQR cache line which is the target of the conditional jump (see Listing 6 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 what is done in [96]).

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 if the monitored cache lines are close. Consequently, for our "clock" we choose to monitor a cache line far away from the nQR cache line (in our case the function `sha256_prf_bits`).

7.3.1 Classification of Cache Access Patterns. We want to classify our cache traces as one of two cases depending on the results of the QR test in the first iteration (non-QR or QR). The measured cache access patterns to the two monitored cache lines show a high variance between different traces of the same case. This might be due to OS related noise, speculative execution, or the way that the random dummy iterations affect the branch predictor behavior in the next run of the function. To overcome this we perform a simplified variant of a cache template attack [16, 30]. 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 50000 clock cycles (each iteration takes $\approx$ 200000 clock cycles on our test machine). We encode each trace into two bits that correspond to the two memory locations. In each trace a bit is set

[1] More advanced micro architectural attacks targeting the branch predictor [2, 6, 21] will fail due to the extra random iterations.

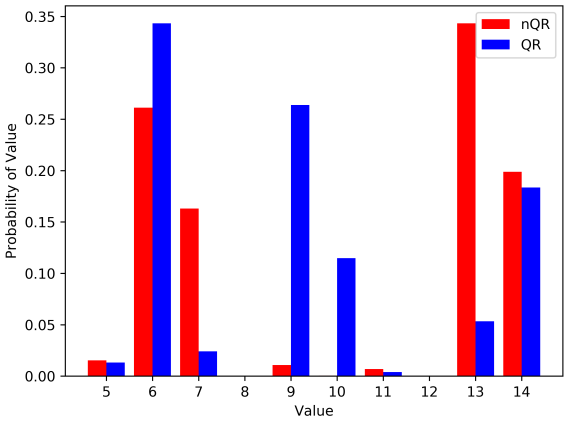


Figure 7: Probability distribution for attack results

to one if its corresponding memory location was accessed, and to zero otherwise.

In our attack we only keep active measurements with at least one non-zero bit. Our attack returns the value of the first two active measurements, meaning the return value consists of four bits (resulting in 9 possible return values). Figure 7 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).

For our classification we repeat the attack 20 times for each MAC address. We created a training set for the non-QR and the QR cases using $100 \cdot 20$ traces each. We used this training set to build a simple linear classifier that receives 20 traces as an input, and returns the input classification, namely either non-QR or QR. We then tested our attack and linear classifier on a larger test set of $400 \cdot 20$ traces for each case. For the non-QR case we have achieved a 100% success rate (400 out of 400), and for the QR case we have achieved a 99.5% success rate (398 out of 400). We can conclude that an adversary can reliably abuse cache-based side-channels to determine whether the password element was found in the first iteration or not.

7.4 Countermeasures and Discussion

As in the MODP case, the ideal solution is to modify the SAE handshake such that the password element is independent of MAC addresses, and use a constant-time hash-to-curve algorithm from the new standard [73]. Even if the attacker can attack the one-time offline calculation, and exploit some residual side-channel leakage, the expected number of password bits leaked is only two. A backwards-compatible countermeasure is to replace the two vulnerable branches with a constant-time select utility, and use constant time Legendre symbol computation as defined in [73].

8 PASSWORD PARTITIONING

In this section we show how to perform password partition attacks, using the information obtained from our timing and cache attacks. This enables an adversary to recover the password of a target.

8.1 Partitioning a Dictionary

In the first attack variant, our goal is to recover the password from a given dictionary. We accomplish this by repeatedly partitioning the dictionary into correct and incorrect password candidates.

Listing 3: Password partitioning algorithm in Python-like pseudo code. It eliminates incorrect passwords based on the result of specific element tests for each spoofed MAC address. It returns a list of remaining candidate passwords.

```
def recover_password(dictionary, testdata, mactarget):
  # dictionary: Set of possible passwords.
  # testdata: Element test results for each spoofed MAC address and
  #           counter value used in a specific element test.
  # mactarget: MAC address of the target (e.g. a client or AP).

  for macspoof, counter, result in testdata
    for p in dictionary:
      if simulate_test(p, macspoof, mactarget, counter) != result:
        dictionary.remove(p)
    if len(dictionary) <= 1: break
  return dictionary
```

Practically, this is implemented by removing incorrect passwords from the dictionary during each partitioning step. If the dictionary becomes empty, this means the target's password was not in it. However, if after the partitioning steps only one password remains, then with high probability this is the target's password.

The result of every element test that is performed in a (password-dependent) iteration of Listing 1 or 2 can be used to partition the dictionary. We use the term *element test* to refer to both the quadratic residue test for elliptic curves, and the if-test that checks whether the prime of the MODP group is bigger than the hash output. Recall that with one timing measurement against the MODP algorithm, we learn on average the result of multiple (failed) element tests. Considering element tests separately also has the advantage that, if for example we are unsure whether a spoofed address resulted in 4 or 5 iterations, this info still enables us to determine that the first three element tests failed. By representing our timing and cache attack measurements as a set of element tests, we can now use the same partitioning attack algorithm in both attack scenarios.

The algorithm illustrated in Listing 3 implements the password partitioning algorithm. As input it receives a dictionary, the set of element tests and their result, and the MAC address of the target. The algorithm uses this information to partition the dictionary by removing passwords that lead to a different result for the element test compared to the result that we measuring this the timing or cache-based attack. More importantly, this algorithm can be run offline, i.e., without requiring any interactions with the target device.

8.2 Prerequisites and Success Analysis

To determine the performance of the password partitioning algorithm, we first calculate how many element tests are required to uniquely recover the password with high probability. Note that every element test is independent, because in each iteration the hash inputs are different, resulting in independent hash outputs. Let p_e denote the probability that the group element is not found, meaning another iteration and element test has to be performed. For the elliptic curve algorithm, p_e is close to 50%, and for MODP groups the values for p_e are listed in in Table 3 under $\Pr[value \geq p]$.

We want to know the probability of eliminating d incorrect passwords, when given the result of n element tests. Let Z denote a random variable that equals the number of element tests that are

required to eliminate d incorrect passwords. This means that if a dictionary of size $d+1$ contains the correct password, and we use n element tests, the probability of uniquely recovering the password is $\Pr[Z \leq n]$. To derive this probability, we first introduce random variable Y as being the number of element tests where the password element was found. We have:

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

This is because the result of an element test does not eliminate an incorrect password when the incorrect password has the same result under the given MAC addresses and iteration count. For example, if the real password in a given iteration did not have a quadratic residue, and the incorrect password also did not, then the results of this element test does not eliminate the password. Given that in k out of n measured element tests the password element was found, the probability that all element tests do *not* eliminate a random password equals $(1 - p_e)^k \cdot p_e^{n-k}$. Now let random variable E denote the number of eliminated passwords. The probability that all d incorrect passwords *are* eliminated equals:

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

Finally, given the result of n random element tests, the probability that all d incorrect password are eliminated equals:

$$\Pr[Z \leq n] = \sum_{k=0}^{n} \Pr[Y = k] \cdot \Pr[E = d \mid Y = k] \tag{4}$$

We tested the above formula by running 100 000 runs of the partitioning algorithm on 1 000 passwords. Each run used random simulated element test results (i.e. simulated timing measurements). Our experimental results closely matched that of formula 4.

By trying various values for n with the RockYou password dump, we find that for MODP group 22, having $n \geq 35$ element tests gives us a probability above 95% of uniquely recovering the correct password. On average, we need to perform $35/1.44 = 24.3$ timing measurements to obtain 35 element tests (recall Table 3). For elliptic curve P-256, an adversary needs to obtain 29 element test results to uniquely recover the password with a probability above 95%. Given that our cache-based side-channel attack can detect a QR with 100% accuracy, and a non-QR with 99.5% accuracy, the probability of that on average all used non-QR measurement are correct equals $0.995^{12.5} = 0.939$. The probability of uniquely recovering the password then becomes at least $0.95 \cdot 0.939 = 0.892$. In other words, using 25 cache-based element test results, the probability of recovering the password from the RockYou dump is close to 90%.

Using formula 4, we can also determine the *average* number of element tests that are needed to eliminate all d incorrect passwords:

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

We tested the above formula by running 100 000 runs of the partitioning algorithm with each time 1 000 random passwords, assuming p_e equals 0.3084. Results closely matched the predicted ones.

Assuming the RockYou dump is used for the dictionary, and MODP group 22 as the target, formula 5 teaches us that on average an adversary needs to obtain 28.28 element tests to uniquely recover the password. For elliptic curve P-256, the adversary would need on average 25.11 element tests (i.e. quadratic residue test results).

8.3 Computational Requirements

To estimate the computational costs of running the partitioning algorithm in function of the dictionary size d, we derive the expected number of element tests that have to be simulated (see line 9 in Listing 3). From formula 5 we already know the expected number of element tests that are needed to eliminate all d incorrect passwords. In other words, the partitioning algorithm will execute on average ℓ iterations. During each of these iterations, a percentage of passwords are eliminated from the dictionary. More precisely, when taking a random element test as reference, and comparing it with an incorrect password, the chance of *not* being able to remove the incorrect password as a candidate is $p_f = p_e^2 + (1 - p_e)^2$. Hence, the amount of element tests that are performed on average is:

$$d + p_f^1 d + p_f^2 d + \ldots + p_f^{-\lceil \ell \rceil} d = d \frac{1 - p_f^{\lceil \ell \rceil}}{1 - p_f} \tag{6}$$

We again tested the above formula by running 100 000 runs of the partitioning algorithm, with each time 1 000 random passwords, assuming $p_e = 0.3084$. Results closely matched the predictions. For MODP group 22 and the RockYou dictionary, this would mean that on average we have to perform 33 627 714 element tests. With elliptic curve P-256 this results in 28 689 748 element tests on average.

On a laptop with an Intel i7-8650U CPU running at 1.90GHz, performing an attack using the RockYou password list takes on average around 11 minutes. This means ordinary users can perform this attack on their existing off-the-shelf hardware.

We can further optimize the partitioning algorithm if p_e differs from 0.5. That is, when attacking a MODP group, we can first process the dictionary using an element test result where the target did not found the password in the given iteration. Recall that this happens with probability p_e. A random incorrect password then has a probability $1 - p_e$ of being eliminated. Since on average $\ell \cdot p_e$ element tests can be used to eliminate an incorrect password with a probability of $1 - p_e$, formula 6 can be modified in the obvious way to take our new strategy into account. When using the RockYou dictionary under this new strategy, we need to perform only 20 742 225 element tests using MODP group 22, a reduction by 38%.

8.4 Brute-Force Attacks in the Cloud

In our second variant of the partitioning attack, we essentially perform an offline brute-force attack on the password. More concretely, our goal is to test all possible 8-character lowercase passwords. Using formula 5 we know that on average this requires 38.36 element tests for MODP group 22, and 38.92 for elliptic curve P-256. Recall that for the MODP case, this means we need to make on average $38.36/1.44 = 26.64$ timing measurements. These are relatively modest requirements. For example, in our demonstration of the timing attack we already performed 20 timing measurements. As a result, we must assume an adversary can obtain the required number of element test results.

We now calculate the costs of running the offline partitioning phase on Amazon EC2 instances. For both the MODP and elliptic curve cases, we first performed repeated microbenchmarks where we simulated one million element tests on a single EC2 vCPU. On average, the MODP test took 3.04 microseconds, and the quadratic residue test took 23.25 microseconds. In macrobenchmarks of the partitioning algorithm on the RockYou dictionary, close to identical running times were observed. We now multiply these timings with the result from formula 6. In particular, for MODP group 22 we need to perform on average 301 947 836 620 element tests, and for curve P-256 we need to perform 417 654 129 151 tests. Fortunately, we can parallelize the code by splitting the brute-force search out over several workers. Every worker gets access to all the element test results, meaning if sufficient element tests were obtained, every worker will discard all incorrect passwords. Only the real password will be detected as potentially valid.

Assuming we rent c5.18xlarge instances having 72 vCPUs, which costs $3.06 an hour, we can perform the brute-force search against the MODP case for $10.63 on average (within e.g. an hour). The brute-force attack against elliptic curves would cost $125 on average, which although more costly, is still a worryingly low amount.

9 RELATED WORK

After the introduction of WPA, it was quickly found to be vulnerable to dictionary attacks [62]. Later, He and Mitchell formally analyzed WPA's 4-way handshake, and discovered a DoS vulnerability [40, 61]. This resulted in the standardization of a slightly improved 4-way handshake [45]. He et al. continued to analyze the 4-way handshake, and proved its correctness [41]. However, implementations of the 4-way handshake were nevertheless vulnerable to downgrade attacks [85]. Recently, Vanhoef and Piessens discovered that WPA2 was vulnerable to key reinstallation attacks [86, 87]. Finally, Kohlios and Hayajneh provide an overview of WPA2 and the differences with WPA3 [56].

Researchers also discovered several DoS attack against Wi-Fi networks. The most well-known is the deauthentication attack [11]. Other DoS attacks exploit weaknesses in TKIP [28]. Additionally, Könings et al. found several DoS vulnerabilities in the physical and MAC layer of 802.11 [57], and other researchers constructed jammers using commodity hardware [72, 84]. A detailed survey of DoS attacks at the physical and MAC layer is given by Bicakci and Tavli [15]. Aiello et al. show how susceptibility to denial-of-service attacks can be balanced with the need for perfect forward secrecy [7]. 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 [22]. A modified variant was then specified in 2008 [33]. Several close variants of it have been defined over the years, and are commonly referred to as Dragonfly-type handshakes [35, 37–39]. Trevor Perrin did a review of an improved draft of the handshake [69], and later provided an overview of other people's comments on the handshake [68]. Struik reviewed a draft of the handshake [79]. Clarke and Hao discovered a small subgroup attack against a draft of the handshake, which was mitigated in a new draft [19]. Lancrenon and Skrobot provided a security proof of a close variant of the Dragonfly handshake [59]. Finally, Alharbi et al. designed a variant of Dragonfly that attempts to keep computational costs low [8].

Other types of PAKEs have also been proposed by researchers over the years [3, 4, 10, 12, 13, 52, 53, 75, 77, 78, 93], of which some have been submitted as RFCs [31, 58, 58, 74, 76, 81, 92]. Finally, there is also research into post-quantum PAKEs [20, 25].

10 CONCLUSION AND RECOMMENDATIONS

In light of our presented attacks, we believe that WPA3 does not meet the standards of a modern security protocol. Moreover, we believe that our attacks could have been avoided if the Wi-Fi Alliance created the WPA3 certification in a more open manner. Notable is also that nearly all of our attacks are against SAE's password encoding method, i.e., against its hash-to-group and hash-to-curve algorithm. Interestingly, a simple change to this algorithm would have prevented most of our attacks. In particular, the peer's MAC addresses can be excluded from SAE's password encoding algorithm, and instead included later on in the handshake itself. This allows the password element to be computed offline, meaning an adversary can no longer actively trigger executions of the password encoding method. Moreover, this would mean that for a given password, the execution time of the password encoding method would always be identical, limiting the amount of information being leaked. Surprisingly, when the CFRG was reviewing a minor variant of Dragonfly, they actually discussed these type of modifications [48, 49, 69, 80]. However, to our surprise, this change was not incorporated into any of the Dragonfly variants.

We also conjecture that resource-constrained devices may not implement all the side-channel countermeasures, as these may be too costly on lightweight processors. Additionally, correctly implementing our suggested backwards-compatible side-channel countermeasures is non-trivial. This is worrisome, because security protocols are normally designed to reduce the change of implementation vulnerabilities.

Finally, we believe that a more open process would have prevented (or clarified) the possibility of downgrade attacks against WPA3-Transition mode. Nevertheless, although WPA3 has its flaws, we still consider it an improvement over WPA2.

ACKNOWLEDGMENTS

We thank Yuval Yarom for his helpful comments and 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).

REFERENCES

[1] 2013. FIPS PUB 186-4: Digital Signature Standard (DSS). *National Institute of Standards and Technology (NIST)* (2013).
[2] 2019. 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.
[3] Michel Abdalla and David Pointcheval. 2005. Simple password-based encrypted key exchange protocols. In *Cryptographers' track at the RSA conference*. Springer, 191–208.

[4] Michel Abdalla and David Pointcheval. 2005. Simple Password-Based Encrypted Key Exchange Protocols. In *Topics in Cryptology – CT-RSA 2005*. Springer Berlin Heidelberg, 191–208.

[5] Onur Acıiçmez. 2007. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *CSAW*.

[6] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *IMA Int. Conf.*

[7] William Aiello, Steven M. Bellovin, Matt Blaze, John Ioannidis, Omer Reingold, Ran Canetti, and Angelos D. Keromytis. 2002. Efficient, DoS-resistant, Secure Key Exchange for Internet Protocols. In *ACM CCS*.

[8] Eman Alharbi, Noha Alsulami, and Omar Batarfi. 2015. An Enhanced Dragonfly Key Exchange Protocol against Offline Dictionary Attack. *Journal of Information Security* 6, 02 (2015), 69.

[9] Gabor Bajko. 2017. 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.

[10] José Becerra, Dimiter Ostrev, and Marjan Škrobot. 2018. Forward Secrecy of SPAKE2. In *International Conference on Provable Security*. Springer, 366–384.

[11] John Bellardo and Stefan Savage. 2003. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. In *USENIX Security*.

[12] Steven M Bellovin and Michael Merritt. 1992. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*. IEEE, 72–84.

[13] Steven M Bellovin and Michael Merritt. 1993. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*. ACM, 244–250.

[14] Daniel J. Bernstein. 2005. Cache-timing attacks on AES.

[15] Kemal Bicakci and Bulent Tavli. 2009. Denial-of-Service attacks and countermeasures in IEEE 802.11 wireless networks. *Comput. Stand. Interfaces* 31, 5 (2009).

[16] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *ASIACRYPT (Lecture Notes in Computer Science)*, Vol. 5912. Springer, 667–684.

[17] CERT/CC. 2019. Vulnerability Note VU#871675: Security issues with WPA3. http://www.kb.cert.org/vuls/id/871675

[18] Hemant Chaskar. 2019. WLAN Security Enhancements: WPA3, OWE, DPP. Retrieved 9 April 2019 from https://d2cpnw0u24fjm4.cloudfront.net/wp-content/uploads/WLPC_2019_WPA3-OWE-and-DDP_Hemant-Chaskar.pdf.

[19] D. Clarke and F. Hao. 2014. Cryptanalysis of the dragonfly key exchange protocol. *IET Information Security* 8, 6 (2014), 283–289.

[20] Jintai Ding, Saed Alsayigh, Jean Lancrenon, RV Saraswathy, and Michael Snook. 2017. Provably secure password authenticated key exchange based on RLWE for the post-quantum world. In *Crypto Track at the RSA Conference*. Springer, 183–204.

[21] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*.

[22] Scott Fluhrer. 2008. 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.

[23] Scott Fluhrer. 2012. Re: [Cfrg] Status of DragonFly. Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03265.html.

[24] Scott Fluhrer. 2014. Re: [Cfrg] Requesting removal of CFRG co-chair. Retrieved 7 April 2019 from https://mailarchive.ietf.org/arch/msg/cfrg/WXyM6pHDjGRZXZzSc_HlERnp0Iw.

[25] Xinwei Gao, Jintai Ding, Lin Li, Saraswathy RV, and Jiqiang Liu. 2017. Efficient Implementation of Password-Based Authenticated Key Exchange from RLWE and Post-Quantum TLS. Cryptology ePrint Archive, Report 2017/1192. https://eprint.iacr.org/2017/1192.

[26] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptographic Engineering* 8, 1 (2018).

[27] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *ACNS*.

[28] Stephen Mark Glass and Vallipuram Muthukkumarasamy. 2007. A Study of the TKIP Cryptographic DoS Attack. In *International Conf. on Networks*. IEEE.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*. USENIX Association, 897–912.

[31] F. Hao. 2017. J-PAKE: Password-Authenticated Key Exchange by Juggling. RFC 8236.

[32] Dan Harkins. [n. d.]. simultaneous authentication of equals. Retrieved 14 November 2018 from https://sourceforge.net/p/authsae/wiki/Home/.

[33] Dan Harkins. 2008. Simultaneous Authentication of Equals: A Secure, Password-Based Key Exchange for Mesh Networks. In *The Second International Conference on Sensor Technologies and Applications (SENSORCOMM)*. 839–844.

[34] Dan Harkins. 2011. Thwarting Side Channel Attacks. Retrieved 9 September 2018 from https://mentor.ieee.org/802.11/dcn/11-11-1411-01-000m-thwarting-side-channel-attacks.doc.

[35] Dan Harkins. 2012. Secure Pre-Shared Key (PSK) Authentication for the Internet Key Exchange Protocol (IKE). RFC 6617.

[36] Dan Harkins. 2014. 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.

[37] Dan Harkins. 2015. Dragonfly Key Exchange. RFC 7664.

[38] Dan Harkins. 2019. Secure Password Ciphersuites for Transport Layer Security (TLS). RFC 8492. https://doi.org/10.17487/RFC8492

[39] Dan Harkins and G. Zorn. 2010. Extensible Authentication Protocol (EAP) Authentication Using Only a Password. RFC 5931.

[40] Changhua He and John C Mitchell. 2004. Analysis of the 802.1 i 4-Way Handshake. In *WiSe*. ACM.

[41] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C Mitchell. 2005. A modular correctness proof of IEEE 802.11i and TLS. In *CCS*.

[42] Masashi Honma. 2015. [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.

[43] IANA. 2018. Internet Key Exchange (IKE) Attributes. Last retrieved 12 November 2018 form https://www.iana.org/assignments/ipsec-registry/ipsec-registry.xml#ipsec-registry-10.

[44] Thomas Icart. 2009. How to Hash into Elliptic Curves. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*.

[45] IEEE Std 802.11. 2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.*

[46] IEEE Std 802.11. 2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.*

[47] IEEE Std 802.11s. 2011. *Amendment 10: Mesh Networking*.

[48] Kevin M. Igoe. 2012. [Cfrg] Status of DragonFly. Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03258.html.

[49] Kevin M. Igoe. 2012. [Cfrg] Status of DragonFly. Retrieved 8 November 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03261.html.

[50] Kevin M. Igoe. 2012. Re: [Cfrg] Status of DragonFly. Retrieved 9 September 2018 from https://www.ietf.org/mail-archive/web/cfrg/current/msg03264.html.

[51] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *ASIA CCS*.

[52] David P Jablon. 1996. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review* 26, 5 (1996), 5–26.

[53] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks. Cryptology ePrint Archive, Report 2018/163. https://eprint.iacr.org/2018/163.

[54] P. Karn and W. Simpson. 1999. Photuris: Session-Key Management Protocol. RFC 2522.

[55] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. 2014. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296.

[56] Christopher P Kohlios and Thaier Hayajneh. 2018. A Comprehensive Attack Flow Model and Security Analysis for Wi-Fi and WPA3. (2018).

[57] Bastian Könings, Florian Schaub, Frank Kargl, and Stefan Dietzel. 2009. Channel switch and quiet attack: New DoS attacks exploiting the 802.11 standard. In *LCN*.

[58] Watson Ladd and Benjamin Kaduk. 2018. *SPAKE2, a PAKE*. Internet-Draft draft-irtf-cfrg-spake2-07. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-spake2-07 Work in Progress.

[59] Jean Lancrenon and Marjan Škrobot. 2015. On the Provable Security of the Dragonfly Protocol. In *Information Security*. Springer International Publishing.

[60] Jouni Malinen. 2015. SAE: Increase security parameter k to 40 based on Dragonfly recommendation. Hostap commit `4584b66eaecd`.

[61] John Mitchell and Changhua He. 2005. Security Analysis and Improvements for IEEE 802.11i. In *NDSS*.

[62] Robert Moskowitz. 2003. 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.

[63] Mozilla. 2019. Strict-Transport-Security - HTTP. Retrieved 3 February 2019 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[64] Yoav Nir, Tero Kivinen, Paul Wouters, and Daniel Migault. 2017. Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2). RFC 8247. https://doi.org/10.17487/RFC8247

[65] Rolf Oppliger. 1999. Protecting key exchange and management protocols against resource clogging attacks. In *Secure Information Networks*. Springer, 163–175.

[66] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*. ACM, 1406–1418.

[67] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*.

[68] Trevor Perrin. 2013. [TLS] Question regarding CFRG process. Retrieved 29 October 2018 from https://www.ietf.org/mail-archive/web/tls/current/msg10962.html.

[69] Trevor Perrin. 2013. [TLS] Review of Dragonfly PAKE. Retrieved 9 September 2018 from https://www.ietf.org/mail-archive/web/tls/current/msg10922.html.

[70] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *CCS*.

[71] Joseph Salowey. 2013. [TLS] Conclusion of WGLC draft-ietf-tls-pwd. Retrieved 7 April from https://mailarchive.ietf.org/arch/msg/tls/Fep2-E7xQX7OQKzfxOoFInVFtm4.

[72] Matthias Schulz, Francesco Gringoli, Daniel Steinmetzer, Michael Koch, and Matthias Hollick. 2017. Massive reactive smartphone-based jamming using arbitrary waveforms and adaptive power control. In *WiSec*. ACM, 111–121.

[73] Sam Scott, Nick Sullivan, and Christopher A. Wood. 2019. *Hashing to Elliptic Curves*. Internet-Draft draft-irtf-cfrg-hash-to-curve-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-03 Work in Progress.

[74] S. Shin and K. Kobara. 2012. Efficient Augmented Password-Only Authentication and Key Exchange for IKEv2. RFC 6628.

[75] SeongHan Shin, Kazukuni Kobara, and Hideki Imai. 2010. Security Proof of AugPAKE. *IACR Cryptology ePrint Archive* 2010 (2010), 334.

[76] S. Smyshlyaev, E. Alekseev, I. Oshkin, and V. Popov. 2017. The Security Evaluated Standardized Password-Authenticated Key Exchange (SESPAKE) Protocol. RFC 8133.

[77] Stanislav V. Smyshlyaev, Igor B. Oshkin, Evgeniy K. Alekseev, and Liliya R. Ahmetzyanova. 2015. On the Security of One Password Authenticated Key Exchange Protocol. Cryptology ePrint Archive, Report 2015/1237. https://eprint.iacr.org/2015/1237.

[78] Michael Steiner, Gene Tsudik, and Michael Waidner. 1995. Refinement and extension of encrypted key exchange. *ACM SIGOPS Operating Systems Review* 29, 3 (1995), 22–30.

[79] Rene Struik. 2013. [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.

[80] Rene Struik. 2013. 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.

[81] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. 2007. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054.

[82] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. 2017. Measuring small subgroup attacks against Diffie-Hellman. In *24th Annual Network and Distributed System Security Symposium NDSS*.

[83] Mathy Vanhoef, Nehru Bhandaru, Thomas Derham, Ido Ouzieli, and Frank Piessens. 2018. Operating Channel Validation: Preventing Multi-Channel Man-in-the-Middle Attacks Against Protected Wi-Fi Networks. In *WiSec*.

[84] Mathy Vanhoef and Frank Piessens. 2014. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*.

[85] Mathy Vanhoef and Frank Piessens. 2016. Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys. In *USENIX Security*.

[86] Mathy Vanhoef and Frank Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *CCS*.

[87] Mathy Vanhoef and Frank Piessens. 2018. Release the Kraken: new KRACKs in the 802.11 Standard. In *CCS*.

[88] Wi-Fi Alliance. 2018. Wi-Fi Alliance introduces security enhancements. Retrieved 6 April 2019 from https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-security-enhancements.

[89] Wi-Fi Alliance. 2018. Wi-Fi Alliance introduces Wi-Fi certified WPA3 security. Retrieved 6 April 2019 from https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-wpa3-security.

[90] Wi-Fi Alliance. 2018. WPA3 Specification Version 1.0. Retrieved 6 April 2019 from https://www.wi-fi.org/file/wpa3-specification-v10.

[91] WikiDevi. 2018. Semantic search: wireless routers. Last retrieved 14 November 2018 form https://wikidevi.com/.

[92] T. Wu. 2000. The SRP Authentication and Key Exchange System. RFC 2945.

[93] Thomas D Wu et al. 1998. The Secure Remote Password Protocol.. In *NDSS*, Vol. 98. Citeseer, 97–111.

[94] Yuval Yarom. 2017. Mastik: A Micro-Architectural Side-Channel Toolkit. cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf.

[95] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Sec*.

[96] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *CHES (Lecture Notes in Computer Science)*, Vol. 9813. Springer, 346–367.

[97] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS*.

A EXPERIMENTS

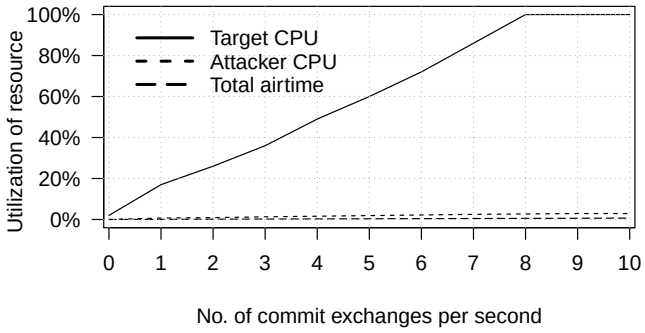


Figure 8: Clogging attack against a professional AP from vendor A using curve P-521. The attacker uses a Raspberry Pi 1 model B+, and its CPU usage is shown in the small dashed line. The total amount of airtime consumed by all SAE frames is shown in the long dashed line.

B SOURCE CODE

Listing 4: Side channel protected quadratic residue test.

```
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 is a quadratic residue modulo p to avoid leaking
    * timing information 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: SAE password derivation using hash-to-curve.

```
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) modulo 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 6: Assembly output of SAE's hash-to-curve method.

```
000000000002efe0 <sae_derive_pwe_ecc>:
...
2f2c8:  e8 f3 17 05 00        callq  80ac0 <sha256_prf_bits>
...

2f719:  e8 f2 fa 04 00        callq  7f210 <crypto_bignum_legendre>
...
2f751:  e8 1a f7 04 00        callq  7ee70 <crypto_bignum_deinit>
2f75d:  0f 85 59 01 00 00     jne    2f8bc <sae_derive_pwe_ecc+0x8dc>
... /* handle qr case code range */
2f7d2:  0f 86 60 fa ff ff     jbe    2f238 <sae_derive_pwe_ecc+0x258>
...
/* start nqr case code */
2f8bc:  48 8b 7c 24 40        mov    0x40(%rsp),%rdi
2f8c1:  be 01 00 00 00        mov    $0x1,%esi
2f8c6:  e8 a5 f5 04 00        callq  7ee70 <crypto_bignum_deinit>
2f8cb:  e9 94 fa ff ff        jmpq   2f364 <sae_derive_pwe_ecc+0x384>
/* end nqr case code */
...
```