A Security Analysis of WPA3-PK: Implementation and Precomputation Attacks

Mathy Vanhoef[0000−0002−8971−9470] and Jeroen Robben[0000−0001−7566−1194]

DistriNet, KU Leuven, 3001 Leuven, Belgium
first.last@kuleuven.be

Abstract. Creating secure Wi-Fi hotspots has historically been challenging: when using an open network it is trivial for an adversary to eavesdrop traffic. Alternatively, when using a password-protected network and sharing the password publicly, anyone who knows the password can create a rogue clone of the network to intercept traffic. To overcome this problem, the Wi-Fi Alliance released SAE-PK as part of an update to WPA3, which we will call WPA3-PK. In this protocol, a public key is used to verify the hotspot's authenticity, and the password of the network encodes a fingerprint of this public key. As a result, someone who knows the password can no longer clone the network, because they do not know the corresponding private key.
In this paper, we systematically analyze the security of WPA3-PK. We first study implementations, where we show that the private WPA3-PK password gets leaked when using a flawed random number generator, and confirm that this may indeed happen in practice. We then study network aspects, where we show how a malicious insider can intercept the traffic of others. Our third focus is cryptographic attacks, where we perform an evaluation of time-memory trade-off attacks against WPA3-PK, and we optimize these attacks by combining the technique of rainbow tables with distinguished points. Additionally, we construct multi-network password collisions that allow an adversary to build a single rainbow table that can be used to attack multiple networks. Finally, we discuss defenses against our attacks and propose updates to the WPA3-PK standard.

Keywords: WPA3-PK · Rainbow table · Time-memory trade-off.

1 Introduction

Securing Wi-Fi hotspots has historically been a daunting task. Using an open, unsecured, Wi-Fi network makes it trivial for an adversary to read and intercept any user's traffic. A password-protected network, where the password is shared publicly, is not much better: anyone who knows the password can create a rogue clone of the network to intercept all traffic. Previous works tried to improve this situation by creating a new enterprise authentication method, where the public key of the network is pinned and used to authenticate the hotspot, and where the client is not authenticated [11,14]. Unfortunately, these proposals never gained widespread adoption, and could still be attacked when a client connects to the

network for the first time. To remedy this situation, and better protect Wi-Fi hotspots, the Wi-Fi Alliance released the Simultaneous Authentication of Equals Public Key (SAE-PK) protocol in December 2020 as part of an update to the WPA3 specification. We will refer to this protocol as WPA3-PK.

The goal of WPA3-PK is to strengthen the security of password-protected Wi-Fi hotspots by preventing an adversary from creating a rogue clone of the hotspot, even when that adversary possesses the pre-shared password. This is achieved by authenticating the hotspot with a public key and by verifying the authenticity of this public key using a password. The idea is that the password is derived from the hotspot's public key, meaning the password effectively contains a trusted fingerprint of the public key. When a user connects to the hotspot, the fingerprint encoded in the password can then be used to verify the hotspot's public key. An adversary cannot create a rogue clone of the hotspot as long as it is infeasible to generate a private and public key that results in the same fingerprint and WPA3-PK password.

In this paper, we systematically analyze the security of WPA3-PK. We first investigate existing deployments of WPA3-PK, where we analyze implementation and network-related aspects. Doing so, we discover that using a bad random number generator will cause the hotspot's password to be leaked. Additionally, because WPA3-PK does not mandate client isolation, we found that network-layer attacks can still be abused to intercept the traffic of other users.

Our second focus is time-memory trade-off attacks against WPA3-PK. In these attacks, the goal is to find a private and public key that result in a given WPA3-PK password, i.e., to perform a second preimage attack. We first evaluate the technique of distinguished points, and confirm that a precomputation attack reduces the time to find a second preimage of a WPA3-PK password from 48 CPU years to an amortized time of fewer than 12 days. We then combine this approach with rainbow tables to increase the success rate of the attack. To evaluate our rainbow table attack, we create a proof-of-concept tool that can generate the precomputed rainbow tables. These experiments confirm that using rainbow tables improves the performance of time-memory trade-off attacks against WPA3-PK. We also show how to construct multi-network WPA3-PK password collisions. These allow an adversary to attack multiple networks using a single precomputed table.

Finally, we propose improvements to the design of WPA3-PK that prevent our newly discovered attacks. We also discuss backward-compatible mitigations that either prevent or reduce the impact of our attacks.

To summarize, our contributions are:

- We analyze implementation and network aspects of WPA3-PK, such as random number generation, client isolation, and shared group keys (Section 3).
- We empirically evaluate time-memory trade-off attacks against WPA3-PK. We also show how to construct rainbow tables to more efficiently invert a WPA3-PK password into a public and private key pair (Section 4).
- We construct multi-network WPA3-PK password collisions that allow an adversary to attack multiple networks using a single rainbow table (Section 5).

– We discuss defenses against the identified design and implementation issues and suggest updates to the WPA3-PK standard (Section 6).

Disclosure. We reported our security analysis of WPA3-PK to the Wi-Fi Alliance. Our code to construct and analyze rainbow tables, and our multi-network WPA3-PK password collisions code, are both available online [1].

2 Background

In this section, we introduce the SAE handshake, how WPA3-PK extends SAE, and explain the generation of WPA3-PK passwords and their security properties.

2.1 Simultaneous Authentication of Equals (SAE)

The Simultaneous Authentication of Equals (SAE) handshake, also called Dragonfly, lies at the basis of WPA3 and provides forward secrecy and resistance against offline dictionary attacks. This handshake was first introduced in 2008 by Harkins [12] and in 2018 became mandatory in home WPA3 networks [29].

A client can discover nearby Wi-Fi networks that support SAE by sending a broadcast probe request (see Figure 1). Nearby Access Points (APs) will reply with probe responses. These responses contain various properties of the network, including the name of the network, which is commonly also called the SSID (Service Set Identifier), and whether the network supports SAE.

Once the client finds a network to connect to, it can initiate the SAE handshake. This handshake consists of two phases, called the commit and confirm phase, and these are illustrated in Figure 1. The first phase can be viewed as a variation of a Diffie-Hellman key exchange, except that the generator used for exponentiation is derived from a pre-shared password instead of being a fixed value [19]. In other words, the first phase negotiates a shared key between the client and AP using Auth-Commit frames. The second phase is used to confirm that the Access Point (AP) and client derived the same keys in the commit phase. More precisely, the confirm element in the Auth-Confirm frame is used to verify that the other party negotiated the same keys. After the SAE handshake, the client associates to the AP, and finally performs a 4-way handshake to derive pairwise transient keys that can be used to protect data frames.

The SAE password can only be shared with trusted individuals. This is because anyone that possesses the password can create a rogue clone of the network with the same SSID and password, and can then trick victims into connecting to this rogue clone. This makes SAE unsuitable for hotspots, since in that case the password is shared publicly, meaning adversaries will also possess the password.

2.2 WPA3 Public Key (WPA3-PK)

In December 2020, the Wi-Fi Alliance released the SAE Public Key protocol as part of an update to the WPA3 certification [29]. We will use WPA3-PK to refer to this protocol. This protocol has as goal to improve the security of password-protected Wi-Fi hotspots and makes it infeasible for a malicious insider to create

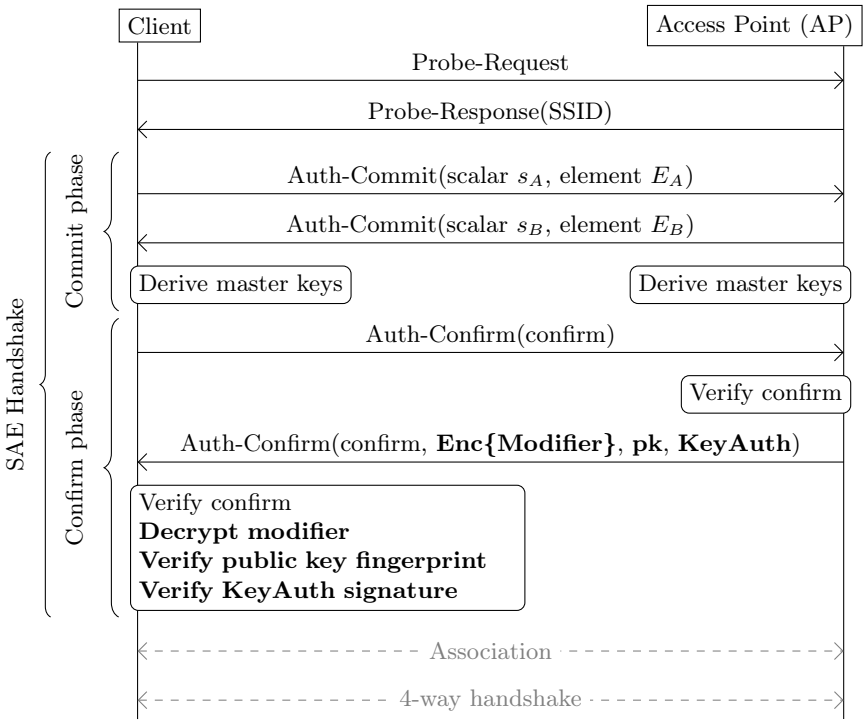

Fig. 1. Diagram of the SAE handshake and the extensions added by WPA3-PK. The parameters and actions shown in bold are unique to SAE-PK: these additions allow the client to verify the authenticity of the AP and assure the AP is not a rogue clone.

a rogue clone of a hotspot. In other words, even if an attacker has the hotspot's password, it would be infeasible to create a rogue clone of the network. The WPA3-PK protocol accomplishes this by authenticating the network using a public key and by making the password encode a fingerprint of this public key. When a client connects to the network, the AP transmits its public key pk to the client in the confirm frame (see Figure 1), and the authenticity of this public key is verified using the pre-shared WPA3-PK password. Once the public key is verified, it is used to authenticate the AP. Concretely, the AP will authenticate itself by signing the following data using its private key:

$$\text{KeyAuth} = \text{Sig}_{sk}(E_B \parallel E_A \parallel s_B \parallel s_A \parallel M \parallel pk \parallel \text{AP}_{\text{MAC}} \parallel \text{STA}_{\text{MAC}})$$

Here sk represents the hotspot's private key. For our purposes, the parameters E_A, E_B, s_A, and s_B, can be treated as random values in the commit phase of the handshake. Including these parameters in the signed data ensures that the KeyAuth signature is unique for each execution of the handshake. The signature is also computed over the public key pk, over the MAC addresses of the AP and client, and over a modifier value M that is used to generate the WPA3-PK password (see Section 2.3). The Elliptic Curve Digital Signature Algorithm (ECDSA)

Algorithm 1: Create a fingerprint of an SSID and public key pk. We use $\text{PKHash}_{\theta,\ell}(pk, \text{SSID}, M)$ to denote the algorithm. The parameters θ and ℓ are dropped when clear from context.

Input: pk: Public key of the hotspot to calculate a fingerprint for.
SSID: Network name to calculate a fingerprint for.
M: Starting modifier, this is by default a random value.
θ : Number of internal digest bits that must be zero.
ℓ: Fingerprint length in bits (excluding removed leading zeros).
Returns: Fingerprint of the given SSID and public key.

for $i = 0$ **to** $2^{128} - 1$ **do**
  $m_i \leftarrow (M + i) \mod 2^{128}$ ▷ The modifier consists of 128 bits
  $h \leftarrow \text{Hash}(\text{SSID} \parallel m_i \parallel pk)$ ▷ m is encoded in big-endian
  if $\log_2(h) < 128 - \theta$ **then** ▷ Check whether the first θ bits are zero
    return $L(h, \theta, \theta + \ell)$ ▷ Remove first θ zero bits and return result
  $i \leftarrow i + 1$
return *false* ▷ We should never get here in practice

algorithm is used to create the signature, which implies that the public key pk must be based on elliptic curves. The client can verify the KeyAuth value using the public key pk, and in the next section, we describe how in turn the authenticity of the public key pk can be checked based on the WPA3-PK password.

2.3 Generation of the WPA3-PK password

The WPA3-PK password is generated so that it can act as a secure and user-friendly fingerprint of the hotspot's public key. Note that simply using the normal fingerprint of a public key as the password would not be user-friendly, since a traditional fingerprint is the hash of the public key and this is too long. Instead, to balance security and usability, WPA3-PK generates a fingerprint of a public key as shown in Algorithm 1: The public key, along with the SSID of the hotspot, is combined with a random modifier M such that the first θ bits of the following hash are zero:

$$\text{Hash}(\text{SSID} \parallel M \parallel pk) \tag{1}$$

Here $\parallel$ denotes concatenation of binary strings. Variable M denotes a 32-byte integer which is incremented until this hash function returns an output whose first θ bits are zero. The modifier M is encoded in big-endian, and SSID represents the binary encoding of the network name. Including the SSID in the hash input ensures that each network has a different fingerprint even when they use the same public key. The argument pk represents the public key of the network and is encoded according to RFC 5480 [23]. Once a modifier has been found that results in θ leading zero bits, these first θ bits are dropped, and the next ℓ bits are returned. In other words, the function $L(h, \theta, N)$ in Algorithm 1 extracts bits θ to N of the binary string h starting from the left. The length of the public key influences the hash function that is used [29]. At the time of

writing, this is either SHA2-256, SHA2-384, or SHA2-512 [3, §Table 12-1]. Algorithm 1 shows the resulting algorithm and we will represent it using the function $\text{PKHash}_{\theta,\ell}(pk, SSID, M)$.

The output from PKHash acts as a fingerprint of the public key, where the parameters θ and ℓ control the fingerprint's security. The values for both these parameters are derived from the parameters *Sec* and λ as follows [29]:

$$\theta = 8 \cdot Sec \tag{2}$$

$$\ell = 19 \cdot \frac{\lambda}{4} - 5 \tag{3}$$

Allowed values for *Sec* are 3 or 5, and allowed values for λ are 12, 16, 20, and so on [29]. For instance, when picking $Sec = 3$, the output of the hash operation in equation (1) must start with 24 zero bits, while with $Sec = 5$, the hash output must start with 40 zero bits. This fingerprint is then encoded into a human-readable password, where the parameter λ corresponds to the number of characters that are required to encode the resulting password. The human-readable password also encodes the security parameter *Sec*. For the remainder of the paper, we will use the terms fingerprint and password as synonyms.

The AP will transmit the modifier M and public key pk to any client that connects using WPA3-PK (see Figure 1). This allows the client to recompute the fingerprint of the given public key, SSID, and modifier, and compare the resulting fingerprint with the one encoded by the WPA3-PK password. In case these fingerprints do not match, the handshake is aborted. Note that the modifier value is encrypted with the negotiated key to ensure that the modifier remains unknown to outsiders. This is important, because if the modifier gets leaked to outsiders, then the WPA3-PK password of the network will leak (see Section 3.1).

2.4 Security guarantees provided by WPA3-PK

It is important that WPA3-PK is sufficiently resistant to second preimage attacks. That is, given a WPA3-PK password, it must be infeasible to find a modifier M and public key pk (for which the private key is known) that results in the given WPA3-PK password. To estimate the resistance against such attacks, the WPA3 specification calculates the cost of a brute-force preimage search for various security parameters λ and *Sec*. This analysis indicates that, when targeting a network that uses the lowest allowed security setting of $\lambda = 12$ and $Sec = 3$, and when using a single hash miner capable of 50 TeraHashes per second, it would take roughly 48 CPU years to attack a WPA3-PK password [29].

We also remark that WPA3-PK can be used to secure private Wi-Fi networks. Compared to plain WPA3, using WPA3-PK has the advantage that, if an internal device is compromised, this compromised device cannot abuse the pre-shared password to create a rogue clone of the network to attack other devices.

3 Implementation and Network-Based Attacks

In this section, we investigate implementation risks of WPA3-PK. That is, we analyze the impact of bad randomness and study network-layer security aspects.

Table 1. Different implementations of PKHash and whether they let the modifier M start from a random value, and if so, which source is used to generate random numbers.

Password generation tool	Start value of M	Source of randomness
Hostap's sae-pk-gen	Random	Linux's `/dev/urandom/`
OpenSSL-based tool	Random	Router's MAC address
Python3 implementation	Zero	—

3.1 Bad randomness leaks the password

Threat model. In this subsection, we will assume that the adversary does not know the password of the WPA3-PK network, but wants to obtain it. This threat model corresponds to one of the design goals of WPA3-PK, namely, that the password should remain secret if it is not shared publicly. This means that in this subsection, WPA3-PK is not used to secure a hotspot, but we instead assume that WPA3-PK is used to secure a private home network.

Flawed randomness risk. To ensure that the WPA3-PK password stays secret, the modifier M should start from an unpredictable value when generating the password (see Algorithm 1). In fact, the WPA3 specification states: "the Modifier is generated using a random number generator with high entropy" [29]. Using high entropy is essential because, due to the design of WPA3-PK, using low entropy randomness risks leaking the WPA3-PK password. In particular, when an adversary does not know the password of a WPA3-PK network, they can monitor the network until a legitimate client tries to connect. The adversary can then capture the public key pk that is sent in plaintext in the last Auth-Confirm frame (see Figure 1). Once the adversary has obtained the public key, the initial value of the modifier M can be guessed, and the PKHash algorithm can be executed to find the WPA3-PK password of the network. It is therefore essential that a cryptographically strong random number generator is used to initialize the modifier M in the PKHash algorithm, since that will prevent an adversary from guessing the (initial) value of the modifier.

Implementation analysis. To investigate whether WPA3-PK implementations securely initialize the modifier M, we searched for open-source implementations of PKHash and studied those. More precisely, we analyzed: (1) the sae-pk-gen password generation tool included in Linux's hostap daemon; (2) an implementation of PKHash based on OpenSSL that is part of a modified dd-wrt release; and (3) a Python3 implementation of PKHash. The analyzed source code snapshots of these three tools are available on our repository [1]. Table 1 gives an overview of these three implementations and their properties.

We found that hostap's sae-pk-gen uses `/dev/urandom` to generate an initial value of M. Although researchers have previously identified weaknesses in older Linux implementations of `/dev/urandom` [15], it is believed to be a secure source of randomness in newer kernels. In contrast, we found that the PKHash implementation based on OpenSSL, and used in a dd-wrt fork, was using an insecure

method to initialize the modifier. In particular, it used the MAC address of the router as the argument to `srand`, and then used libc's `rand` function to initialize the modifier. This means that the initial value of M can be inferred by an adversary and that the resulting WPA3-PK password can be derived from the hotspot's public key. Finally, the Python3 implementation of PKHash always initialized the modifier M to zero and incremented it until a valid modifier was found. This makes it trivial for an adversary to derive the WPA3-PK password when only knowing the public key of the network.

Evaluation. To evaluate our attack, and confirm that an adversary can infer the WPA3-PK password generated by vulnerable implementations, we generated WPA3-PK passwords using the three implementations in Table 1. The generated passwords and private keys were used to create a WPA3-PK hotspot using Linux's `hostapd` daemon. To then perform the attack and infer the network's password, we created a Python script that uses the `Scapy` library to monitor Wi-Fi frames sent by the AP. When a legitimate client connects, and the Auth-Confirm frame sent by the AP is detected, our script will extract the AP's public key from this frame (recall Figure 1).

Once the AP's public key has been intercepted, our script runs the PKHash algorithm locally with the captured public key as input. In a first run of PKHash, the initial value of the modifier value M is set to zero, to simulate the Python3 implementation. In a second run, the initial value is set based on the MAC address of the AP. All combined, this results in two potential WPA3-PK passwords. To determine whether one of these passwords is correct, we use `wpa_supplicant` to try to connect to the AP using these passwords. If one of the connections is successful, we know that the password is correct. We repeated this experiment 10 times, where each time new WPA3-PK passwords and public keys were generated, and each time our script was able to derive the password generated by the OpenSSL-based and Python3 implementation of PKHash.

3.2 Network-based attacks

Client-to-client attacks. With WPA3-PK, an adversary cannot set up a rogue AP to intercept the traffic of clients. However, by default, it remains possible to intercept a victim's traffic using network-based attacks. In particular, an attacker can connect as a client and then use ARP poisoning to redirect and intercept the traffic of other users that are connected to the hotspot. To perform an ARP poisoning attack, the adversary must know the IP address of the victim, but that info can be determined by scanning the network using tools such as `nmap`.

We confirmed this attack in practice against a Linux AP running hostapd 2.10 that was configured as a WPA3-PK network, connecting to the AP using two Linux laptops, and using Scapy to perform an ARP poisoning attack. This successfully poisoned the ARP cache of both the victim client and the AP, and caused the attacker to intercept all traffic to and from the victim.

Abusing group keys. When using WPA3-PK, the group key that is used to protect broadcast and multicast Wi-Fi traffic is shared between all clients.

This means that an adversary can connect to an WPA3-PK hotspot, learn the group key, and abuse this key to spoof broadcast and multicast traffic to all clients. More worrisome, previous work has shown that against many devices, it is possible to inject unicast traffic using the group key [27], worsening the impact of such an attack. Overall, we found that an adversary can abuse the group key in a WPA3-PK network to send arbitrary traffic to other clients even if client-to-client traffic was blocked by the network.

We confirmed the above attack against a Linux client that was using version 2.10 of `wpa_supplicant`. In our attack, we connected ourselves to the WPA3-PK network using a modified `wpa_supplicant` that outputs the group key. This group key was then used to inject both broadcast and unicast frames towards the victim, even though client-to-client traffic was disabled by the AP.

4 Precomputation Attacks and Rainbow Tables

In this section, we study improved time-memory trade-off attacks against WPA3's SAE-PK protocol, i.e., against WPA3-PK. We analyze the expected performance of a baseline attack, improve this attack using rainbow tables, and evaluate a proof-of-concept implementation of the rainbow table attack.

4.1 Background on time-memory trade-off attacks

Precomputation attacks. Our goal is to find a modifier M and public key pk that results in a given password, i.e., to perform a preimage attack. That is, we want to invert $\text{PKHash}_{\theta,\ell}(pk, \text{SSID}, M)$ when given a SSID, public key pk, and security parameters θ and ℓ. One option is doing a brute-force search for a value M that results in the desired output, but that requires either a large amount of computational power or takes a huge amount of time. When performing a preimage attack multiple times, it is typically possible to precompute information so that subsequent attacks can be carried out faster. Such time-memory trade-off attacks were first introduced by Hellman in 1980: he proposed a probabilistic method to break a block cipher that supports 2^n possible keys by precomputing a lookup table of $2^{2/3n}$ elements, after which recovering the key from a known plaintext takes $2^{2/3n}$ operations [13]. Another common use case for time-memory trade-off attacks is to invert a hash function.

In a time-memory trade-off attack, intermediate results are saved so that subsequent attacks are more efficient. For instance, assume we want to invert a hash function H, i.e., given a hash output C we want to find an input P such that $C = H(P)$. A naive idea is to iterate over all inputs and save *all* input and hash output pairs. However, this requires a large amount of storage. Instead, in a time-memory trade-off attack, the inputs and hash outputs are organized in chains, and only the first and last elements of each chain are saved. The chains are created by defining a reduction function R that transforms a hash output into a new candidate hash input. We then define the function $f(p) = R(H(p))$ that maps an input p to another input, and use this to construct a chain of inputs:

$$p_1 \xrightarrow{f(p_1)} p_2 \xrightarrow{f(p_2)} \dots \xrightarrow{f(p_{t-1})} p_t \tag{4}$$

For every chain only the first input p_1 and last input p_t are stored. These two points are commonly called the starting point and endpoint, respectively. By changing the length t of chains we will be able to trade lookup time with memory.

To find an input that results in a given hash output C, we first create a new (temporary) chain of inputs of length t starting with $R(C)$. For every output in this temporary chain, we look up whether this input occurs as an endpoint in the precomputed table. Once an endpoint has been found, the entire chain in the precomputed table is reconstructed, which is possible since the precomputed table contains the starting input of each chain. If the recreated chain contains an input that results in the given hash output C, then the lookup was successful, since that means we found an input that results in the given hash output C. If the recreated chain does not contain the hash output C, then we say that a false alarm has occurred, and we continue the search until t applications of f are applied to $R(C)$.

Chain collisions. Time-memory trade-off attacks are probabilistic: there is no guarantee that all hash inputs of a given length are contained in the table. The success probability of a preimage attack will therefore depend on the size of the precomputed table and how the table is constructed. Additionally, chains may collide with each other, meaning at some point they both generate the same (partial) chain of inputs. We call this a chain collision. To increase the success rate, and reduce the number of collisions, a common strategy is to create multiple smaller subtables that each use a (slightly) different reduction function R.

Rainbow tables and distinguished points. Various improvements to time-memory trade-off attacks have been proposed over time. Two important ones are Distinguished Points (DP) and rainbow tables. The idea behind distinguished points was first mentioned by Rivest [9, p.100] and was later investigated in detail by Borst et al [7]. When using distinguished points, the number of table lookups is reduced, which is important when working with slow storage mediums or large lookup tables. Arguably the most well-known improvement is the technique of rainbow tables, which was proposed by Oechslin in 2003 [22]. The advantage of this technique is that the number of chain collisions is reduced, and that there is a reduction in the expected number of table lookups compared to the classical method of Hellman.

4.2 Motivation: SSID reuse

The network's SSID influences the WPA3-PK password and thereby acts as a salt to mitigate precomputation and time-memory trade-off attacks. However, SSIDs are frequently reused by different networks, meaning it still is beneficial to perform precomputation attacks against WPA3-PK. To investigate how common SSID reuse is, we analyzed the SSID statistics provided by WiGLE [2], which at the time of our analysis contained 884 396 925 Access Points (APs). Based on this data, Figure 2 shows how many Wi-Fi networks are represented by the most common 100 SSIDs. We can see that the most common 100 SSIDs represent almost 10% of all Wi-Fi networks worldwide. The most common SSID,

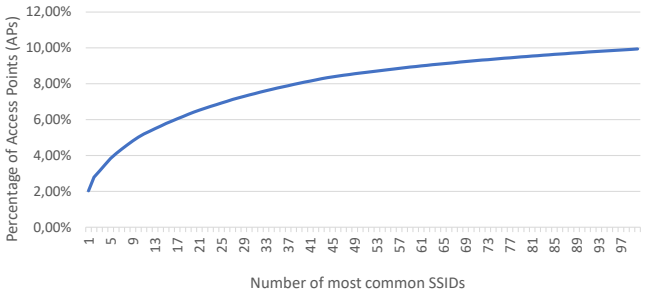

Fig. 2. Percentage of Access Points (APs) on WiGLE that are represented by the top most common SSIDs [2]. This excludes the empty SSID that is used to hide networks.

`xfinitywifi`, represents 2,03% of all APs, and the top 10 SSIDs represent 5,02% of all APs. This shows that SSIDs are frequently reused and motivates our research into precomputation and time-memory trade-off attacks, since the reuse of SSIDs enables the use of a single precomputed table to attack multiple networks.

4.3 Baseline precomputation attack against WPA3-PK

Our rainbow attack extends the time-memory trade-off attack of [26]. We therefore first introduce this baseline attack and perform a more extensive evaluation of its performance. Both attacks have as input a public key pk for which we know the private key and a WPA3-PK password with parameters ℓ and θ, i.e., a fingerprint, and then find a modifier M such that $\text{PKHash}_{\theta,\ell}(pk, \text{SSID}, M)$ has as output the given fingerprint. The baseline time-memory trade-off attack works as follows [26]:

Reduction function. The reduction function takes a fingerprint, i.e., an output of PKHash, and converts it to a modifier value M. The function takes the given fingerprint of ℓ bits, and appends it with zero bits until it has a total length of 16 bytes. With this construction, the chance of chain collisions is reduced, because the counter m_i inside PKHash will then be unlikely to ever equal another fingerprint.

Constructing chains and tables. The baseline attack uses distinguished points to construct chains, which makes handling large tables more efficient [6,22]. That is, it keeps applying the reduction and PKHash function until a fingerprint with a given number of leading zero bits is encountered. This fingerprint is called a distinguished endpoint or distinguished fingerprint. The number of leading zeros of a distinguished fingerprint is represented by d. This implies that the internal hash output in Algorithm 1 must start with $\theta + d$ zero bits.

To construct one table, m random fingerprints are picked as starting points, denoted by p_1 to p_m. For each starting point, the reduction and PKHash function, which corresponds to function f in Section 4.1, is executed until we get a distinguished fingerprint. Function f can now be written as follows:

$$p_{i,j+1} = \text{PKHash}_{\theta,\ell}\left(pk, \text{SSID}, p_{i,j} \ll (32 \cdot 8 - \ell)\right) \tag{5}$$

Table 2. Symbols used in this paper and their meaning.

Symbol	Description
λ	Length of the SAE-PK password (defined by WPA3)
Sec	Security level of the fingerprint (defined by WPA3)
pk	Public key
sk	Private key
M	Modifier value used to calculate a fingerprint
m	Number of starting points in one table
r	Table index
T	Number of tables
B	Number of colors used in a table
c	Current color
θ	Number of SHA2 output bits that must be zero
ℓ	Length in bits of the desired fingerprint
d	Number of leading zeros in distinguished fingerprints
t	Represents the (average) length of a chain

Here $p_{i,j}$ is the j-th point in chain i. Each chain i starts with fingerprint $p_{i,1} = p_i$. The operator $\ll$ denotes a binary left shift, and the constant $32 \cdot 8$ corresponds to the length of the modifier M. To detect loops in a chain, the length of a chain is limited to t_{max} elements. If no distinguished fingerprint was found after t_{max} applications of the function f, the chain is discarded.

Storage. For every chain i we store the starting fingerprint p_i, the distinguished endpoint $p_{i,t}$, and the chain length t. By storing the chain length we can merge chain collisions and only keep the longest chain. To allow lookups of an endpoint in logarithmic time, the table is sorted based on the distinguished endpoints.

Multiple tables. Using multiple smaller subtables, where each table uses a unique reduction function, reduces chain collisions which improves the success rate of table lookups and makes it easier to parallelize lookups [6,25]. One can construct a unique reduction function per table by encoding the index of the subtable into the high-order bits of the modifier M. After the bits that encode the table's index, the output of the previous PKHash call is placed. In other words, for subtable r out of T in total, the combination of the reduction and PKHash function becomes:

$$s_{table} = 32 \cdot 8 - \lceil \log_2(T) \rceil \tag{6}$$

$$s_{mod} = s_{table} - \ell \tag{7}$$

$$p_{i,j+1} = \text{PKHash}_{\theta,\ell}\left(pk, \text{SSID}, (r \ll s_{table}) \mid (p_{i,j} \ll s_{mod})\right) \tag{8}$$

Operator $\mid$ denotes the binary OR operation and T is the number of subtables.

4.4 Improved analysis of the baseline precomputation attack

To determine the performance of the above baseline time-memory trade-off attack, we improve its proof-of-concept implementation, evaluate its resulting per-

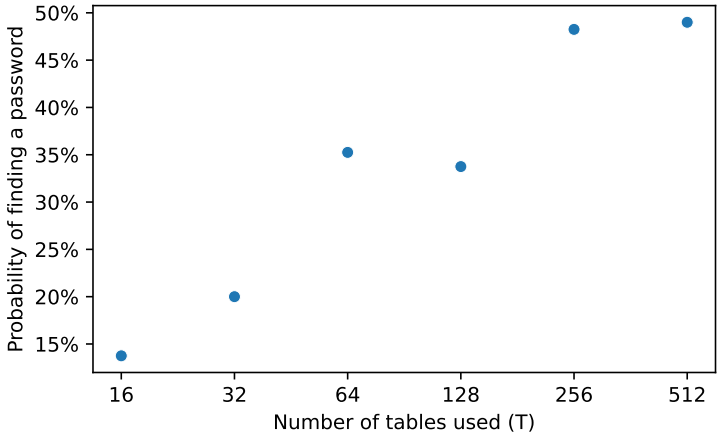

Fig. 3. Performance of lookup tables for the baseline time-memory trade-off attack. The x-axis denotes the number of subtables T. The total number of starting points across all subtables is identical in each experiment. In other words, when employing fewer subtables, each one contains a higher number of chains, i.e., starting points.

formance, and compare this more thorough experimental evaluation with the theoretical estimates of [26]. Note that in the theoretical analysis, chain merges are ignored to simplify the analysis at the cost of some reduction in precision [25, §8].

Experiments. We started with the proof-of-concept implementation of [26] and carried out a more extensive evaluation over more parameters. While doing so, we fixed a bug in the lookup function in the proof-of-concept implementation that caused the endpoint in a chain to be overwritten, which caused the success rate of table lookups to be underestimated. When then ran simulations with the WPA3-PK security parameters $\theta = 8$ and $\ell = 24$, used $d = 8$ for distinguished fingerprints, $t_{max} = 2^{11}$ as the maximum chain length, used $m = 2^8$ starting points per table, and $T = 2^8$ individual tables. Based on a simulation of 400 password lookups, on average one lookup required $2^{16.42}$ calls to PKHash, and the success rate of a password lookup was 46.5%. This is a substantially higher success rate compared to the analysis in [26], which we attribute to the bug fix in the proof-of-concept implementation.

We also further improved the proof-of-concept code to support arbitrary bit lengths for the parameters θ, ℓ, and d. This enabled a more thorough performance evaluation while still ensuring that simulations terminate within practical time. In particular, in our second set of experiments, we set $\theta = 0$, $\ell = 24$, and $d = 8$. The number of subtables T was set to 16, 32, ..., 512. The number of chains m was chosen so the table generation covered on average 2^ℓ hash inputs. That

is, $T = \frac{2^\ell}{2^d \cdot m}$, which ensures the generation time of the table is equal under all parameter combinations, and ensures that all subtables combined have the same size, resulting in a fair comparison between the different tables. For every generated table, we looked up 400 random passwords, and measured how many lookups were successful. The results of this experiment are shown in Figure 3. We can clearly see that using different smaller subtables, each with their own unique reduction function, improves the performance of the time-memory trade-off attack.

Precomputation complexity. We can compare our observed performance with the predicted theoretic performance calculated in [25,26]:

1. **Success rate.** The success rate of finding an WPA3-PK passphrase in a single table equals $SR \approx \frac{s(\gamma m)}{2^\ell}$ [25,26]. In case we use T different tables, where each table has a different reduction function, the probability of a successful lookup is $PS(T) = 1 - (1 - SR)^T$ [25].

   For example, when using $T = 2^8$ tables, the expected success rate is 50%. In practice, we saw a success rate of 46.5% in the first experiment and 48.25% in the second experiment. As another example, for $T = 2^4$ tables the predicted success rate is 16%, and the observed success rate is 13.75% (see Figure 3). Overall, the observed success rates are in line with the predicted rates.

2. **Lookup cost.** The processing complexity, i.e., the expected number of calls to PKHash when looking up an element over all T subtables, can be estimated using $T \cdot \beta$. Here β is the average length of a chain, which can be approximated by 2^d. This assumes that there are no false alarms when looking up a password [25].

   For example, when using $T = 2^8$ tables, the expected lookup cost consists of 2^{16} calls to PKHash. In practice, we observe $2^{16.42}$ calls to PKHash in the first experiment, and $2^{16.44}$ in the second experiment. When using $T = 2^4$ tables, we would expect 2^{12} calls to PKHash, and in practice we observe $2^{13.28}$ calls per lookup. We conjecture that this higher time complexity is due to false alarms during a password lookup, e.g., with $T = 2^4$, on average there were 12 false alarms per lookup.

All combined, the results of our experiments are in line with the predicted success rates. The above also confirms the prediction of [26] that breaking WPA3-PK under its lowest security setting, namely when $Sec = 3$ and $\lambda = 12$, would require an amortized computational cost of less than 12 days, where a lookup in the precomputed table would have a success rate of close to 50%.

4.5 Rainbow tables for WPA3-PK

To increase the success rate of a password lookup, we will combine the above table construction with rainbow tables. In a traditional rainbow table, the reduction function R is (slightly) changed at every point in the chain to reduce

chain collisions [22,16]. In our approach, we will change the reduction function once a distinguished fingerprint is encountered:

$$p_{1,1} \xrightarrow{f_1(p_{1,1})} \dots \xrightarrow{f_1(p_{1,t-1})} p_{1,t} \xrightarrow{f_2(p_{1,t})} p_{2,1} \xrightarrow{f_2(p_{2,1})} \dots \xrightarrow{f_2(p_{2,t-1})} p_{2,t} \tag{9}$$

Here function $f_1(p) = R_1(H(p))$ is applied until we get a fingerprint that starts with d zero bits, which is the same as a distinguished fingerprint in our previous table. Once a distinguished fingerprint $p_{1,t}$ is found, we switch to a different reduction function R_2 meaning we apply the function $f_2 = R_2(H(p))$, until we obtain another fingerprint with d leading zero bits, and so on. We say that each reduction function R_c uses a different color c. The total number of colors B used in a chain is a parameter of the table. Analogous to changing the reduction function for every table r, we create a new reduction function per color by encoding the color c in the modifier M. All combined, the function $f_{c,r}(p_{i,j})$ for color c in subtable r becomes:

$$s_{color} = 32 \cdot 8 - \lceil \log_2(B) \rceil \tag{10}$$

$$s_{table} = s_{color} - \lceil \log_2(T) \rceil \tag{11}$$

$$s_{mod} = s_{table} - \ell \tag{12}$$

$$p_i^{j+1} = \mathrm{PKHash}_{\theta,\ell}\left(pk, \mathrm{SSID}, (c \ll s_{color}) \mid (r \ll s_{table}) \mid (p_{i,j} \ll s_{mod})\right) \tag{13}$$

Here c represents the current color used in the reduction function, B the number of colors used, r the table index, and T the number of tables. To detect possible loops in a chain, we discard a chain if no distinguished point was found after 2^{d+3} applications of $f_{c,r}$.

4.6 Rainbow table: performance experiments

We implemented a proof-of-concept of our rainbow table technique to estimate the success probability of password lookups. To ensure simulations finish within practical time, we set $\theta = 0$ and $\ell = 24$. Our tool has as parameters the number of leading bits d of a distinguished point, the number of colors B, the number of subtables T, and the number of chains m in a subtable. Note that when using $B = 1$, meaning only one color is used, the resulting table is equivalent to the tables constructed in the previous two sections.

We did simulations with $B \in \{1, 2, 4, 8, 2^4, 2^5\}$, d ranging from 0 to 8, and $m \in \{2^7, \dots, 2^{12}, 2^{13}\}$. The number of subtables T was chosen so the table generation covered on average 2^ℓ hashes. That is, $T = \frac{2^\ell}{2^d \cdot B \cdot m}$, which ensures the generation time is equal under all parameter combinations, ensuring a fair comparison between the created tables. Note that depending on the values for T and m the resulting tables may be of different size. For every combination of parameters, our tool created the rainbow table, then performed 400 random lookups in the table, and finally wrote the resulting statistics to a JSON file. Figure 4 shows the performance of the resulting lookup tables for tables that store 2^{16} chains or less. This limit for the number of chains over all subtables

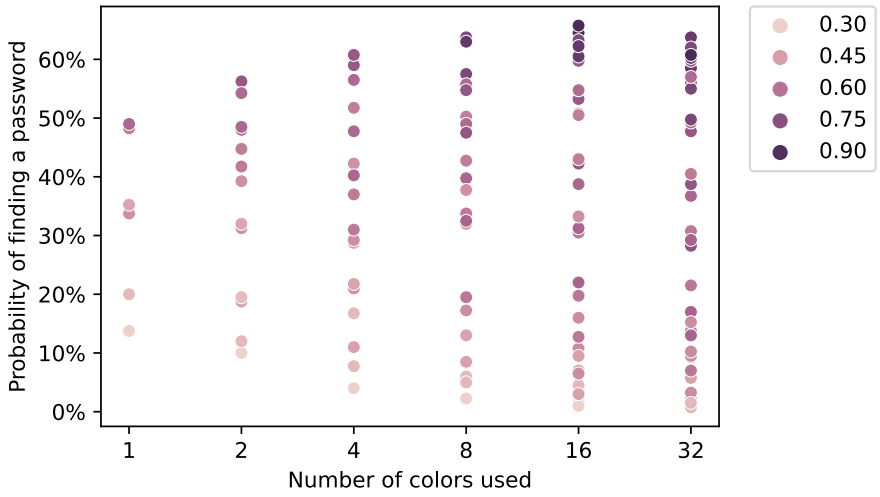

Fig. 4. Performance of lookup tables in function of the number of colors used, number of subtables, and the number of chains in a subtable (see Section 4.6 for details). Each point represents a lookup table containing at most 2^{16} chains. The x-axis denotes the number of colors in a table and the y-axis the resulting password lookup success. The hue of the point represents the normalized, in log scale, average number of table accesses when looking up a password, where the maximum number of table lookups was 10659.

effectively puts a limit on the size of the lookup table, further ensuring a fair comparison. We observe that for tables of similar size, and with a fixed table generation time, the usage of colors increases the attack success probability. For instance, the highest success probability over all parameter combinations with one color is 49%, with two colors this increases to 56%, and with 16 colors it reaches its maximum of 65%.

The increased success probability of using different colors comes at the cost of more table accesses during the lookup of a password. For instance, for the highest success probability for each color, the number of table accesses equal 370 for 1 color, 706 for 2 colors, and 5091 for 16 colors.

5 Multi-Network Password Collisions

In this section, we propose a new method to create password collisions, that is, we create networks with different SSIDs that have the same WPA3-PK password. This is non-trivial because the SSID acts as a salt when calculating the password. We also create multi-network password collisions, where multiple SSIDs have the same WPA3-PK password. These password collisions allow an attacker to create a single precomputed table that can be used to attack multiple networks.

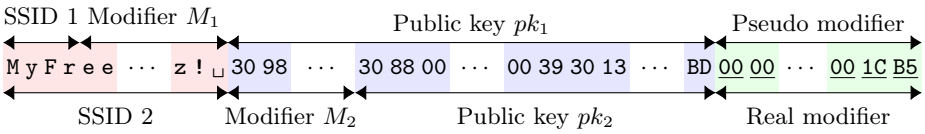

Fig. 5. Input given to PKHash such that the SAE-PK password for both SSIDs is identical. The top shows how this input is split into the SSID, modifier, and public key for the first network. The bottom shows how this input is mapped to the SSID, modifier, and public key for the second network.

5.1 Constructing password collisions

To create WPA3-PK password collisions, we ensure that the input to the underlying hash function of PKHash, i.e., equation 1 in Algorithm 1, is identical for different SSIDs [26]. The core idea to achieve this is that an attacker can still change the length of the SSID *after* the password has already been generated [26]. This idea is illustrated in Figure 5, where the input to PKHash is given two interpretations. In the first interpretation, the SSID equals `MyFr`, the modifier equals the binary encoding of `eeWifi␣2.4␣GHz!␣`, and the public key starts with the bytes `30 90` and ends with the byte `BD`. In the second interpretation, the SSID equals `MyFreeWifi␣2.4␣GHz!␣`, the modifier equals the first 16 bytes of the public key pk_1, and the public key starts in the middle of pk_1 with the bytes `30 88` and ends with the byte `BD` (more on this later).

To create a valid WPA3-PK password, we need to be able to freely modify certain bytes to ensure that the internal hash operation in PKHash starts with enough zeros. However, as shown in Figure 5, the modifier M cannot be freely changed anymore because it now overlaps with the SSID or public key of the other network. To still be able to freely change bytes in the input of PKHash, we will not change the encoding of the public key as in [26], but we will instead include a pseudo modifier after the public key. This pseudo modifier can be changed until the hash output starts with sufficiently many zeros.

Our password collision construction only works if the client does not remove the pseudo modifier that is appended to the public key. Fortunately, when the public key pk is sent to the client in the Auth-Confirm frame, the encoding of the public key is treated as an opaque data blob [3, §9.4.2.180]. This means we can add trailing data after the public key without the client noticing this.

We tested whether `wpa_supplicant`, which is the only open-source Wi-Fi client that supports WPA3-PK, accepts trailing data after the public key. This client supports two crypto libraries when using WPA3-PK, namely OpenSSL and WolfSSL, and in both cases trailing data *after* the public key was accepted and included in the input given to PKHash. This confirms that we can use the structure in Figure 5 to build password collisions, where the pseudo modifier can be changed until the hash output starts with sufficiently many zero bits.

All combined, we can now build a precomputed table where the construction in Figure 5 is used as the input to the internal hash function in PKHash. Here

Table 3. Behavior of crypto libraries regarding the parsing of public keys. The second column contains the tested function, the third whether it returns the number of bytes read, and the fourth column whether trailing data is allowed in the ASN.1 sequence.

Library	SubjectPublicKeyInfo parser	Bytes read	Extra data
OpenSSL	`d2i_PUBKEY`	Yes	Rejected
WolfSSL	`wc_EccPublicKeyDecode`	Yes	Accepted
GnuTLS	`gnutls_pubkey_import`	No	Rejected
MatrixSSL	`psParseSubjectPublicKeyInfo`	No	Accepted

the pseudo modifier contains the argument M of PKHash. An adversary can then use the resulting table to attack both SSIDs.

5.2 Public key embedding and trailing data

One obstacle when creating a password collision is that public key pk_1 must be constructed so that public key pk_2 starts in the middle of it, i.e., we must be able to embed one public key into another. To achieve this, we exploit a similar parsing vulnerability as the one in [18,26], namely that arbitrary data can be encoded in variable length fields. In particular, when encoding a length field, if the length is smaller or equal to 127, the length is directly encoded as a byte. For example, the byte `0x10` encodes the length 16. Otherwise, if the length is 128 or higher, the high-order bit of the first byte is set, and the other low-order bits denote how many subsequent bytes encode the actual length. For example, the two bytes `0x81 0xFF` represent the length 255. We can embed arbitrary data inside this variable length encoding by using the following construction:

```
8C XX XX XX XX 00 00 00 00 00 00 00 39
```

The first byte denotes that the next 12 bytes will be used to encode the length field. However, most implementations can handle at most 64-bit integers. As a result, only the last 8 bytes of the length field matter, and the 4 bytes represented by `XX` are effectively ignored due to an integer overflow.

We can now encode the *start* of a public key into the variable length field of another public key. In particular, we can put the bytes `30 88` in place of the last two `XX` bytes in our example. Here `30` encodes the start of the public key and `88` denotes a variable length field where the length is represented using the next 8 bytes. After this length field in the byte sequence, both public keys are aligned, meaning that the remaining bytes will encode the same public key.

5.3 Accepting trailing data inside the public key

An alternative to putting the pseudo modifier as trailing data *after* the public key, is to put it in the *end* of the public key itself. More precisely, the encoding of the public key is defined using ASN.1 as follows:

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
   algorithm         AlgorithmIdentifier,
   subjectPublicKey  BIT STRING
   // Add trailing data, i.e., the pseudo modifier M, here  }
```

The idea is that we add trailing data after the subjectPublicKey, but inside the SubjectPublicKeyInfo sequence. Out of the 4 TLS libraries we tested, WolfSSL and MatrixSSL accepted trailing data in this location, and this did not interfere with the parsing of the public key. This was tested by calling the public key parsing functions shown in Table 3, where for MatrixSSL the functions `getEcPubKey` and `PsParseSubjectPublicKeyInfo` were combined to parse the public key.

5.4 Multi-network password collisions

Apart from creating a WPA3-PK password collision for two SSIDs, we can also create a collision for multiple SSIDs. To accomplish this, we assume that every SSID is a prefix or extension of another SSID, and that each SSID differs in length from all other SSIDs by at least two characters. This, for instance, allows us to construct collisions for the following sets of SSIDs:

$$\{\ \texttt{MyFreeWifi␣2.4␣GHz!␣},\ \texttt{MyFreeWifi␣2.4␣GHz},\ \ldots,\ \texttt{MyFree},\ \texttt{MyFr}\ \}$$

The maximum difference in length between the longest and shortest SSID is 16 characters. This limitation is a result of having to use the 16-byte modifier M to ensure that different SSIDs still result in the same hash input (recall Figure 5).

When constructing the collision, we use the same format as in Figure 5 where a pseudo modifier M is placed after the public key. However, the variable length fields of the public keys are now constructed as shown in Figure 6. The idea is that, as long as the public key starts on one of the underlined bytes, then all the remaining bytes will be ignored until the actual public key starts. This means a valid public key can start at multiple locations, which in turn means multiple SSID lengths will result in a valid starting position of the public key.

We created a script to create multi-network password collisions [1]. It takes as input a private key, the longest SSID that we want to be part of the collision, and the security parameter *Sec* of the resulting WPA3-PK password. The tool will then create collisions for all shorter SSIDs in steps of two characters.

We also created a modified AP that advertises our constructed public key along with the given SSID. When tested against a client that is vulnerable to the same parsing flexibility as in [18,26], the public key was accepted, and the client could use the same WPA3-PK password to connect to all the different SSIDs.

6 Defenses and Discussion

In this section, we discuss possible defenses against all our attacks and propose updates to the WPA3-PK standard.

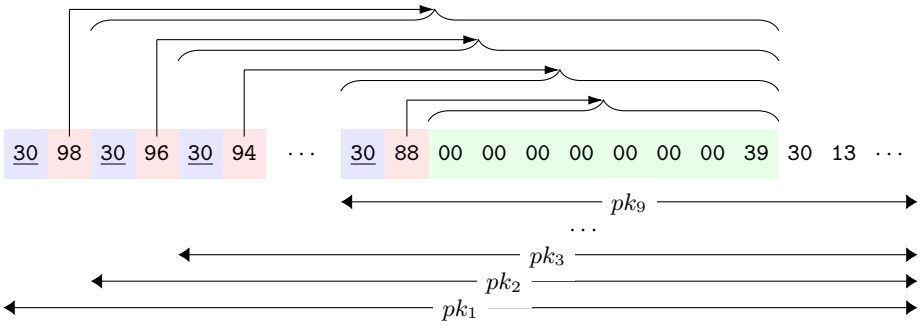

Fig. 6. Structure of the variable length fields in a multi-network password collision. The underlined bytes represent the start of a public key. The red bytes encode the size of the variable length field. The green bytes encode the actual length of the remaining public key bytes, and all preceding length bytes are ignored due to integer overflows.

6.1 Handling bad randomness: encrypting the public key

To prevent bad randomness from leaking the password, the AP should send the public key to the client in an encrypted manner. This can easily be done: the AP already encrypts the modifier M when sending it to the client (recall Figure 1). The AP can use the same encryption operation to also encrypt the public key.

When the client uses the correct password to connect, they can decrypt and obtain the public key, and then verify the authenticity of the public key. In case the client is not using the correct key, the confirm message in the Auth-Confirm frame is invalid, and the client will drop the frame before trying to decrypt the public key. All combined, a legitimate client will still be able to obtain the public key, while it will remain hidden from an adversary. When adopting this approach, it is no longer possible for an adversary to derive the WPA3-PK password, even when the modifier M was initialized in a predictable manner.

6.2 Preventing network-layer attacks

Our network-based attacks can be prevented by: (1) blocking all types of client-to-client communication [4, §5.1]; and by (2) using the Downstream Group-Addressed Forwarding (DGAF) Disable feature of Passpoint [4, §5.2], which effectively disables the use of the group key in Wi-Fi networks.

6.3 Mitigating time-memory trade-off attacks

To mitigate time-memory trade-off attacks in a backwards-compatible manner, networks can use a WPA3-PK password of at least $\lambda = 16$ characters or use a security parameter of $Sec = 5$. This makes it too costly to construct a precomputed table. Alternatively, to prevent time-memory trade-off attacks, the user can scan a QR code to learn the precise public key instead of only its fingerprint. Additionally, network administrators can decide to use a unique SSID, since adversaries are less likely to create a precomputed table for unique SSIDs.

6.4 Preventing password collisions: committing to an SSID length

To prevent an adversary from constructing password collisions, the input given to the hash function inside PKHash should be updated to start with a single byte that represents the length of the SSID. This forces an attacker to commit to a specific SSID length when constructing the precomputed time-memory trade-off table, or the rainbow table, and thereby prevents our WPA3-PK password collision attacks. Unfortunately, this change requires a modification to the protocol. A backwards-compatible mitigation is to more strictly parse the public key and to only allow a single possible encoding of the public key.

7 Related Work

This paper builds upon our previous preliminary analysis of WPA3-PK [26], and also investigates network-based attacks, studies practical implementation risks of the password generation algorithm, more accurately evaluates the baseline time-memory trade-off attack, proposes rainbow table attacks, and explores new techniques to construct multi-network password collisions.

The predecessor of WPA3, namely WPA2, was quickly shown to be susceptible to offline dictionary attacks [20]. Internally, WPA2 employs the PBKDF2 algorithm to derive a Pairwise Master Key (PMK) from the combination of the password and SSID. This resulting PMK serves as the input for the WPA2 4-way handshake, where the PMK, along with two random nonces, are mixed to generate a fresh session key. An adversary can intercept the 4-way handshake and then perform a brute-force attack to determine the password that results in the correct session key. However, due to the inclusion of two randomly generated 32-byte nonces in the session key derivation, performing a time-memory trade-off attack against the 4-way handshake is not possible. Nonetheless, given the slow nature of offline brute-forcing attempts on personal computers, lookup tables were generated to speed up the brute-force attacks against WPA2's 4-way handshake [24]. By using these precomputed tables, one can avoid executing the computationally intensive PBKDF2 hash that maps an SSID and password to the corresponding PMK. Initially, these tables covered the top 1 000 SSIDs using a dictionary of 172 000 possible passwords, which was subsequently expanded to a dictionary containing one million words. The resulting lookup tables occupied 7 GB and 33 GB of storage space. Due to the increased adoption of GPU-based password-cracking methods, the demand for such lookup tables has waned.

Regarding the security of WPA3, researchers quickly showed that it was vulnerable to timing attacks [28]. Although backwards-compatible defenses were proposed, not all implementations properly implemented these defenses [5].

Flexibilities in parsing public keys were previously abused to attack RSA [10]. Related to this, it was also discovered that some libraries accept arbitrary parameters in the algorithm identifier [8].

Hellman introduced time-memory trade-off attacks [13] and Oechslin improved them using the rainbow construction [22]. Using distinguished points to perform time-memory trade-off attacks was first proposed by Rivest [9] and later

worked out by Borst et al. [7]. Nohl combined the technique of rainbow tables with distinguished points to break the GSM A5/1 cipher, where this combination was important to handle large lookup tables in practice [21], and the generation of these tables was later improved by using FPGAs [17].

8 Conclusion

Our security analysis of WPA3-PK revealed that, when using this protocol in practice, it is important to also consider implementation and network-layer attacks. In particular, implementations must use a secure random number generator and clients must be properly isolated from each other at the network layer.

When using the weakest allowed WPA3-PK password, we demonstrated that time-memory trade-off attacks, including the creation of rainbow tables, are on the verge of practicality. These attacks enable an adversary to perform a preimage attack, i.e, to find a public and private key that result in a given WPA3-PK password. To mitigate these attacks, we recommend setting the parameter *Sec* to 5, or using a WPA3-PK password of at least $\lambda = 16$ characters. This is especially important when using a common SSID name for the network.

Acknowledgements This research is partially funded by the Research Fund KU Leuven and by the Flemish Research Programme Cybersecurity.

References

1. `https://github.com/vanhoefm/acns-wpa3-pk-sae`
2. WiGLE: Statistics: SSID/manufacturer. Retrieved 20 May 2022 from `https://web.archive.org/web/20220520222830/https://wigle.net/csv/ssid.csv`
3. 802.11, I.S.: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec (2020)
4. Alliance, W.F.: Hotspot 2.0 Specification Ver. 3.1 (2019)
5. de Almeida Braga, D., Fouque, P.A., Sabt, M.: Dragonblood is still leaking: Practical cache-based side-channel in the wild. In: ACSAC (2020)
6. Borst, J.: Block ciphers: Design, analysis and side-channel analysis. Ph.D. thesis, KU Leuven, Leuven, Belgium (2001)
7. Borst, J., Preneel, B., Vandewalle, J.: On the time-memory tradeoff between exhaustive key search and table precomputation. In: Symposium on Information Theory in the Benelux. pp. 111–118. Technische Universiteit Delft (1998)
8. Chau, S.Y., Yahyazadeh, M., Chowdhury, O., Kate, A., Li, N.: Analyzing semantic correctness with symbolic execution: A case study on PKCS# 1 v1.5 signature verification. In: NDSS (2019)
9. Denning, D.E., Denning, P.J.: Data security. ACM Computing Surveys (CSUR) **11**(3), 227–249 (1979)
10. Finney, H.: Bleichenbacher's RSA signature forgery based on implementation error. Retrieved 21 November 2021 from `https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/` (2006)
11. Gonzales, H., Bauer, K., Lindqvist, J., McCoy, D., Sicker, D.: Practical defenses for evil twin attacks in 802.11. In: GLOBECOM (2010)

12. Harkins, D.: Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks. In: The Second International Conference on Sensor Technologies and Applications (SENSORCOMM). pp. 839–844 (Aug 2008)
13. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE transactions on Information Theory **26**(4), 401–406 (1980)
14. Hendershot, T.S.: Towards Using Certificate-Based Authentication as a Defense Against Evil Twins in 802.11 Networks. Master's thesis, Brigham Young University (2016)
15. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: USENIX Security. pp. 205–220 (2012)
16. Hong, J., Jeong, K.C., Kwon, E.Y., Lee, I.S., Ma, D.: Variants of the distinguished point method for cryptanalytic time memory trade-offs. In: International Conference on Information Security Practice and Experience (2008)
17. Kalenderi, M., Pnevmatikatos, D., Papaefstathiou, I., Manifavas, C.: Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS. In: FPL (2012)
18. Langley, A.: PKCS# 1 signature validation. Retrieved 21 November 2021 from `https://www.imperialviolet.org/2014/09/26/pkcs1.html` (Sep 2014)
19. Lee, H., Won, D.: Prevention of exponential equivalence in simple password exponential key exchange (speke). Symmetry **7**(3), 1587–1594 (2015)
20. Moskowitz, R.: Weakness in passphrase choice in WPA interface. Retrieved 26 December 2021 form `https://wifinetnews.com/archives/2003/11/weakness_in_passphrase_choice_in_wpa_interface.html` (2003)
21. Nohl, K.: Attacking phone privacy. Black Hat USA pp. 1–6 (2010)
22. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: Annual International Cryptology Conference. pp. 617–630. Springer (2003)
23. Polk, T., Housley, R., Turner, S., Brown, D.R.L., Yiu, K.: Elliptic Curve Cryptography Subject Public Key Information. RFC 5480 (Mar 2009). `https://doi.org/10.17487/RFC5480`, `https://rfc-editor.org/rfc/rfc5480.txt`
24. RenderMan, Wright, J., Thorn, Dragorn, H1kari, Twitchy: Church of wifi WPA-PSK lookup tables. Retrieved 26 December 2021 from `https://www.renderlab.net/projects/WPA-tables/` (2006)
25. Standaert, F.X., Rouvroy, G., Quisquater, J.J., Legat, J.D.: A time-memory trade-off using distinguished points: New analysis & FPGA results. In: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems. p. 593–609. CHES '02, Springer-Verlag, Berlin, Heidelberg (2002)
26. Vanhoef, M.: A time-memory trade-off attack on WPA3's SAE-PK. In: Proceedings of the 9th ACM ASIA Public-Key Cryptography Workshop. ACM (2022)
27. Vanhoef, M., Piessens, F.: Predicting, decrypting, and abusing WPA2/802.11 group keys. In: USENIX Security. pp. 673–688 (2016)
28. Vanhoef, M., Ronen, E.: Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In: IEEE Security & Privacy (SP). IEEE (2020)
29. Wi-Fi Alliance: WPA3 specification version 3.0. Retrieved 20 November 2021 from `https://www.wi-fi.org/file/wpa3-specification` (Dec 2020)