A Time-Memory Trade-Off Attack on WPA3's SAE-PK

Mathy Vanhoef
Mathy.Vanhoef@kuleuven.be
imec-DistriNet, KU Leuven
Belgium

ABSTRACT

To increase the security of Wi-Fi hotspots, the Wi-Fi Alliance released the SAE-PK (Simultaneous Authentication of Equals - Public Key) protocol in December 2020. In this protocol, the password of the network is derived from the public key and the name of the network. The idea is that an adversary cannot, within practical time, generate their own private and corresponding public key that leads to the same password. This prevents an adversary from setting up a rogue Wi-Fi network with the same name and password because they do not have a private and public key that results in the given password.

In this paper, we analyze the impact of time-memory trade-off attacks on the security of the SAE-PK protocol. By utilizing distinguished points to construct precomputed tables, we estimate that the amortized cost of attacks can be significantly reused when the name of the Wi-Fi network, called the SSID (Service Sets Identifier), is reused. For instance, we estimate that when an adversary is targeting an SSID used by a large number of networks, the amortized computational costs of attacking the weakest allowed SAE-PK configuration is lowered from 48 CPU years to roughly two weeks. Finally, we also demonstrate how to construct SAE-PK password collisions, where two networks with different SSIDs end up using the same password. Such a password collision enables the creation of a single precomputed table that can attack multiple SSIDs.

CCS CONCEPTS

• **Security and privacy** → **Cryptanalysis and other attacks**; *Public key (asymmetric) techniques*; **Security protocols**; • **Networks** → Network protocol design.

KEYWORDS

WPA3, SAE-PK, 802.11, Wi-Fi, time-memory trade-off, collision

ACM Reference Format:
Mathy Vanhoef. 2022. A Time-Memory Trade-Off Attack on WPA3's SAE-PK. In *Proceedings of the 9th ACM ASIA Public-Key Cryptography Workshop (APKC '22), May 30, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3494105.3526235

1 INTRODUCTION

The Wi-Fi protocol remains one of the most frequently used protocols to access the internet. Apart from being used in home and company networks, it is also commonly used by public venues and shops to offer free and convenient access the internet. However, the security of such public Wi-Fi networks, often also called hotspots, is typically lacking. For instance, an adversary can trivially eavesdrop on the traffic of open Wi-Fi hotspots and can create rogue hotspots to manipulate traffic. Fortunately, in December 2020, the Wi-Fi Alliance updated the Wi-Fi Protected Access 3 (WPA3) certification with features to strengthen the security of hotspots.

The initial version of the WPA3 certification was released in April 2018 and one of its contributions is a new handshake to authenticate to a network based on a shared password. An advantage compared to WPA2 is that this new handshake, called Simultaneous Authentication of Equals (SAE), provides forward secrecy and prevents offline dictionary attacks. Although some security issues were discovered in the SAE handshake [25], these were rectified by the Wi-Fi Alliance [26]. However, a remaining challenge was how to protect Wi-Fi hotspots where everyone, including possibly malicious users, are in possession of the password.

Version three of WPA3, which was released in December 2020, added a feature to secure hotspots [26]. The goal of this update is to prevent an adversary from creating a rogue clone of the network with the same name and password as the legitimate network. To achieve this property, the password is tied to a public key, and only those who own the corresponding private key can set up a legitimate Access Point (AP). Since the adversary only knows the name and password of the hotspot, but not the private key, they are unable to set up a rogue clone of the hotspot. This new authentication mechanism is based on the SAE handshake and is called SAE Public Key (SAE-PK). In this paper we will explore the security guarantees that SAE-PK provides.

We first investigate the feasibility of time-memory trade-off attacks against SAE-PK. This is of importance because the WPA3 specification, when discussing appropriate security parameters for SAE-PK, currently assumes that the most efficient attack is a brute-force attack. However, because the SAE-PK password is essentially a truncated hash of the name and public key of the network, it becomes possible to perform a time-memory trade-off attack against it. In particular, after a large precomputation effort with as goal to attack a particular SSID, it becomes possible to more efficient to map a password back to a (different) public key for which the corresponding private key is known. In other words, when targeting a common SSID, or an SSID that remains the same over an extended period of time, it becomes possible to break a password using a lower amortized cost than previously assumed.

To further optimize our time-memory trade-off attack, we also show how it is possible to precompute a table that can be used to target multiple SSIDs. This is accomplished by showing how an adversary can construct multiple public keys and SSIDs that all result in the same SAE-PK password, i.e., we show how to create SAE-PK password collisions. We test the feasibility of constructing password collisions against various TLS libraries.

APKC '22, May 30, 2022, Nagasaki, Japan

Our attacks can be mitigated by using unique SSIDs. Alternatively, the discovered weaknesses can also be mitigated by using an SAE-PK password of at least 16 characters, or by using a password with a higher security level, i.e., by setting the *Sec* parameter of the SAE-PK password to 5.

To summarize, our contributions are:

- We highlight the reuse of SSIDs between different networks, and show how to construct time-memory lookup tables for SAE-PK when SSIDs are reused (Section 3).
- We show how to crease password collisions, allowing one to build a single table to attack multiple SSIDs (Section 4).
- We propose and discuss defenses against the weaknesses that we identified (Section 5).

To explain our contributions we first introduce the necessary background information in Section 2. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

In this section we first introduce WPA3 and the SAE handshake. We then explain the SAE-PK extension that secures hotspots by tying a public key to a password. Finally, we will give an introduction to time-memory trade-off attacks.

2.1 Simultaneous authentication of equals (SAE)

The first version of WPA3 mandates support of the Simultaneous Authentication of Equals (SAE) handshake which is also known as the Dragonfly handshake [26]. Unlike the 4-way handshake of WPA2, the SAE handshake provides forward secrecy and defends against offline dictionary attacks. When using SAE, the administrator picks a password of at least 5 characters, and distributes this password to all users of the network. The password can only be given to trusted users, since anyone that possess the password can set up a rogue network with the same SSID and password.

Figure 1 illustrates that messages that are exchanged when a client uses SAE to connect to an Access Point (AP). First, the client transmits probe requests, and nearby APs will reply using probe responses. These probe responses contain various properties of the network, including the SSID of the network. After this network discovery phase, the SAE handshake starts, where in the first phase of the handshake the client and AP use Auth-Commit messages to exchange a random scalar and element. The exchanged scalar and element are combined with the password to derive a fresh master key. In the second phase of the SAE handshake, the client and AP use Auth-Confirm messages to exchange confirm elements that are used to assure that both endpoints generated the same master keys. Finally, after the SAE handshake, the client associates and performs a 4-way handshake to derive fresh data protection keys.

2.2 SAE public key (SAE-PK)

To secure hotspots, version three of the WPA3 certification added the SAE Public Key (SAE-PK) protocol [26]. The idea behind this protocol is that the password of the network is tied to a specific public key. This means that the network administrator no longer picks a password manually, but instead a password is derived from the public key's fingerprint. When a client now connects to the network, the AP will transmit the public key *pk* to the client, and the client will compare the fingerprint of the received public key to the fingerprint encoded in the password. Only when these fingerprints match, and a given value is properly signed by the AP using the corresponding private key, will the client continue connecting. In the case of SAE-PK, the data that must be signed by the AP is called the KeyAuth field, which among other things contains the public key, the client's and AP's random scalar and element (see the Auth-Commit messages in Fig 1), and the MAC addresses of the client and AP. This assures that the KeyAuth field and its signature is unique for each SAE-PK handshake. The signature itself is made using the Elliptic Curve Digital Signature Algorithm (ECDSA), meaning that only public keys based on elliptic curves are supported by SAE-PK.

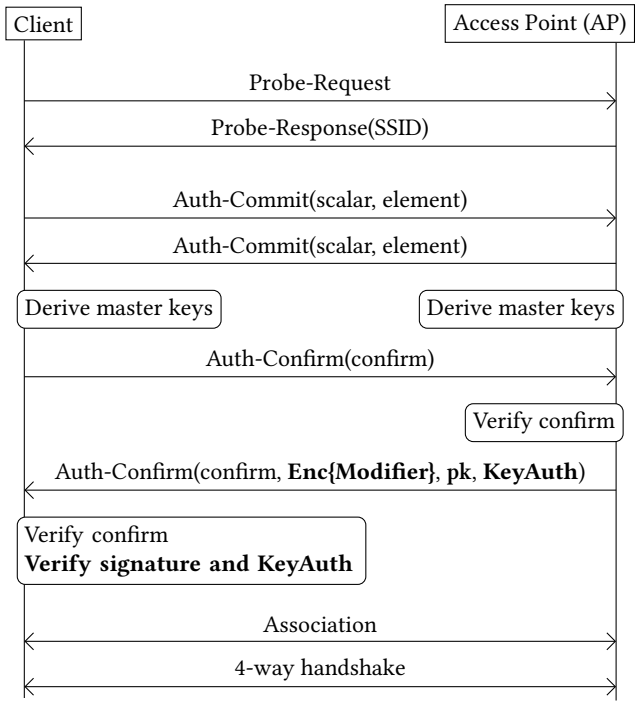

Figure 1: Illustration of the SAE-PK handshake. It adds the parameters and steps in bold to the SAE handshake, so clients can verify the authenticity of hotspots using public keys.

2.2.1 SAE-PK fingerprint. To generate the SAE-PK password, a fingerprint is first derived from the public key, SSID, and from a random modifier. This fingerprint is then encoded into a human-readable SAE-PK password (see Section 2.2.2). Algorithm 1 illustrates how the fingerprint is calculated, and we will represent this algorithm using the function $\text{PKHash}_{\theta,\ell}(pk, SSID, M)$. The first argument *pk* is the public key of the network. The second argument is the SSID of the network, which acts as a salt to assure that each network has a different fingerprint even when they use the same public key. The third argument is a 32-byte integer which is initialized to a random value and called the modifier [26]. This modifier is incremented until the hash function

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

Algorithm 1: Create a fingerprint of an SSID and public key. The algorithm is denoted by $\text{PKHash}_{\theta,\ell}(pk, \text{SSID}, M)$. Parameters θ and ℓ are dropped when clear from context.

Input: pk: Public key to calculate a fingerprint for
SSID: Network to calculate a fingerprint for
M: Starting modifier (by default a random value)
θ : Number of internal digest bits that must be zero
ℓ: Length in bits of the returned fingerprint

Returns: Fingerprint of the SSID and public key which can be easily converted into an SAE-PK password.

for $i = 0$ **to** $2^{128} - 1$ **do**
  $d \leftarrow \text{Hash}(\text{SSID} \parallel M + i \parallel pk)$ ▷ $M + i$ is in big-endian
  if $\log_2(d) < 128 - \theta$ **then** ▷ Are first θ bits zero?
    return $L(d, \theta, \theta + \ell)$ ▷ Remove the first θ bits
  $i \leftarrow i + 1$
return *false* ▷ Should never get here in practice

returns an output whose first θ bits are zero. When constructing the input to the above hash function, the modifier M is encoded in big-endian, the public key pk is encoded according to RFC 5480 [21], and $\parallel$ denotes the concatenation of binary strings. Once a modifier has been found that results in an output where the first θ bits are zero, these first θ bits are dropped, and the next ℓ bits are returned. That is, in Algorithm 1, the function $L(d, \theta, N)$ extracts bits θ to N of the bit string d starting from the left. The hash function that is used depends on the length of the public key [26]. In the current version of the 802.11 standard this is either SHA2-256, SHA2-384, or SHA2-512 [2, §Table 12-1]. The parameters θ and ℓ control the security of the fingerprint and are further discussed below.

When a client connects to a network using SAE-PK, the AP will transmit the modifier M to the client in the Auth-Confirm message (see Figure 1). To prevent an outsider from learning the modifier value, it is encrypted using the negotiated master keys. This means that an adversary who observes a SAE-PK handshake will only learn the public key of the network, which, on its own, is insufficient to derive the SAE-PK password. This assures that the SAE-PK password remains unknown to outsiders.

2.2.2 Human-readable SAE-PK password. To use Algorithm 1 to generate a fingerprint, the network administrator first has to pick a value for two security parameters called *Sec* and λ [26]. Valid values for *Sec* are either 3 or 5, and valid values for λ are 12, 16, 20, and so on. These two parameters are converted to the arguments θ and ℓ of PKHash as follows:

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

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

In other words, when using $Sec = 3$, the output of the hash operation in (1) must start with 24 zero bits, while with $Sec = 5$, the output must start with 40 zero bits.

The resulting output d of $\text{PKHash}_{\theta,\ell}$ is then encoded into alphanumeric characters as follows. First, let Sec_{1b} be equal to one when $Sec = 3$, and otherwise, when $Sec = 5$, then Sec_{1b} equals zero. The bit Sec_{1b} is prepended to the fingerprint d and also inserted between every 19 bits of the fingerprint. For example, assume that $\lambda = 8$, $Sec = 3$, and that the output d of PKHash in binary is:

$$d = 10000000011111111000000011111111$$

Since we picked $Sec = 3$, we know that $\text{sec}_{1b} = 1$, and inserting this bit in the appropriate places gives:

$$\underline{1}10000000011111111001\underline{0}0000011111111$$

By inserting sec_{1b} in this manner, the receiver of the password reliably learns the parameter *Sec* that was used to generate the fingerprint. This intermediate bitstring, with the inserted sec_{1b} bits, is then appended with 5 bits that function as a checksum, resulting in the bitstring d'. In our example, this leads to the following output:

$$d' = 11000000001111111100100000011111111\underline{00000}$$

The 5-bit checksum is calculated using the Verhoeff algorithm [26]. Finally, d' is encoded into alphanumeric characters using the base32 encoding function as defined in RFC 4648, and every group of four characters is split using a hyphen character [13, 26]. In our example, the resulting alphanumeric password for the SAE-PK network is:

ya74-qh7a

Notice that the parameter λ corresponds to the number of characters that are required to encode the resulting SAE-PK password. In practice, an SAE-PK password must be at least 12 characters, since λ must be equal or higher than 12 [26].

2.2.3 Security of SAE-PK. The WPA3 certification contains a security analysis, where the effort needed to attack various SAE-PK passwords is estimated. An attack is considered successful when the adversary can find a modifier M and public key (for which the private key is known) that results in a given password. When targeting a network that uses $\lambda = 12$ and $Sec = 3$, which is the lowest allowed security setting, it is estimated that attacking a single SAE-PK password takes roughly 48 CPU years when using a hash miner capable of 50 TeraHashes per second. In this analysis, it is assumed that the adversary carries out a brute-force search for a modifier M and public key that results in a given SAE-PK password.

2.2.4 Packet formats. Note that when the public key pk is sent to the client in the Auth-Confirm frame (see Figure 1), the public key is transported using the FILS Public Key element [26]. This element is defined in the 802.11 standard [2, §9.4.2.180] and places no restrictions on how the public key is encoded. This will become important in Section 4 when constructing password collisions.

2.3 Time-memory trade-off attacks

Brute-force search attacks require either a large amount of (parallel) computational power or take a substantial amount of time to complete. In case the attack is performed 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 [12]. A common use-case for time-memory trade-off attacks is to find the key used to encrypt a given plaintext or to invert a hash function.

The idea behind a time-memory trade-off attack is to save intermediate results so that subsequent attacks are more efficient. For

instance, assume we want to build a table to recover the key that was used by a cipher E to encrypt a plaintext P_0 when given the resulting ciphertext C_0. A naive idea is to iterate over all keys and save *all* resulting key and ciphertext pairs. However, this requires a large amount of storage. Instead, in a time-memory trade-off attack, the ciphertexts and keys are organized in chains, and only the first and last element of each chain is saved. The chains are created by defining a reduction function R that transforms a ciphertext into a key. We then define the function $f(k) = R(E_k(P_0))$ that maps a key k to another key, and use this to construct a chain of keys:

$$k_1 \xrightarrow{f(k_1)} k_2 \xrightarrow{f(k_2)} \dots \xrightarrow{f(k_{t-1})} k_t \tag{4}$$

For every chain, only the first key k_1 and last key k_t is stored. By changing the length t of the chain we can trade lookup time with memory. The first element of a chain is called the starting point, and the last element is commonly called the endpoint.

We can use the precomputed table, which consists of the starting points and endpoints of all chains, to find the key that was used to generate a ciphertext C. This is done by generating a new chain of keys of length t starting with key $R(C)$, where C is the captured ciphertext for which we want to find the key. For every generated key in this chain, we look whether this key is an endpoint of a chain stored in the table. Once such an endpoint is found, we can recreate the chain that was stored in the table. If the chain contains a key resulting in ciphertext C then we have successfully found the key that was used to encrypt the given plaintext. Otherwise, if no such key was found in the chain, we say that a false alarm has occurred, and we continue our search until t applications of f have been applied to $R(C)$ where C is the captured plaintext.

It is important to note that time-memory trade-off attacks are probabilistic: there is no guarantee that all keys can be found in a table. The success probability depends on how the table is constructed and how large it is. Additionally, chains may collide with each other, meaning at some point they both generate the same (partial) chain of keys. To increase the success rate, and reduce the number of collisions, a common strategy is to create multiple smaller tables that each use a (slightly) different reduction function R.

After the introduction of time-memory trade-off attacks by Hellman, various improvements have been proposed. Two notable ones are Distinguished Points (DP) and rainbow tables. The idea behind distinguished points was first mentioned by Rivest [10, p.100] and later worked out by Borst et al. [5]. The advantage of distinguished points is that it reduces the number of table lookups, which can be important when working with large lookup tables. We elaborate on the usage of distinguished points in Section 3. In 2003, Oechslin proposed the usage of rainbow tables [20], which among other things reduces the number of chain collisions while simultaneously reducing the number of expected table lookups compared to the classical method of Hellman.

3 TIME-MEMORY TRADE-OFF ATTACK

In this section we explain how to perform a time-memory trade-off attack against SAE-PK. This consists of precomputing a table that an adversary can use to more easily convert an SAE-PK password into a valid modifier and public key for which the private key is known. We assume that the attacker repeatedly wants to attack the same SSID, which can occur when the administrator periodically changes the password of a network, or when the same SSID is used in multiple independent networks.

3.1 Motivation: reused and common SSIDs

Performing a time-memory trade-off attack is only meaningful when the precomputed table can be used repeatedly. In the case of SAE-PK, the precomputed table will (by default) be specific to a given SSID, since the SSID is used as a salt in the password derivation algorithm (recall Section 2.2). This means a time-memory trade-off attack is only relevant if the same SSID is reused.

In practice, there are two scenario in which it is meaningful to attack the same SSID multiple times. First, a network may use the same SSID for an extended period of time. For instance, a company may use its name for the SSID and this name will likely remain the same for several years. However, the public key of the SAE-PK protocol may be periodically refreshed, for instance when using new Wi-Fi hardware, which in turn changes the SAE-PK password. This implies the adversary may wish to attack the same SSID multiple times over a given time-frame.

Second, many different networks use the same SSID. For instance, based on statistics gathered by WiGLE, the most common SSID is `xfinitewifi` which, at the time of writing, is used by 17 289 341 networks [1]. Another example is `linksys` which is used by 3 137 128 networks. In general, network names based on a company name are common, where another example is an SSID such as `stackbucks`. This illustrates that in practice the same SSID might be attacked multiple times, making it beneficial to perform a time-memory trade-off attack where a precomputed table is constructed to more easily attack one SSID multiple times.

3.2 Defining the hash and reduction function

Given a specific SSID, our goal is to be able to efficiently map an SAE-PK password into a modifier value M and a public key pk. The adversary can then use this information to create a rogue AP with the resulting public key pk, matching private key sk, and modifier M, such that these values result in the same SAE-PK password as the target network. In other words, we want to invert the function $\text{PKHash}_{\theta,\ell}(pk, \text{SSID}, M)$ for a given SSID, public key pk, and for given security parameters θ and ℓ. Note that the parameters ℓ and θ determine the type and maximum length of the SAE-PK passwords that we can invert (see Section 2.2). To reduce storage requirements, we will directly work with the output of PKHash without converting it to the alphanumeric SAE-PK password representation. Inverting the function PKHash then implies finding a modifier value M such that, for all other given parameters of PKHash, the function will result in the desired output.

Before constructing the table, we need to define a reduction function that takes the output of PKHash and converts it to a modifier value M that we can give as input to the next PKHash call of the chain (recall Section 2.3). For this reduction function we will use the output of PKHash and append it with zero bits until it has a total length of 16 bytes. With this construction, the counter inside PKHash will unlikely result in an overlap with a different output of the reduction function. This is because the reduction function pushes the output bits of PKHash to the left, while the counter in

Table 1: Symbols used in this paper and their meaning.

Symbol	Description
λ	Length of the SAE-PK password (defined in WPA3)
Sec	Security level of the fingerprint (defined in WPA3)
pk	Public key
sk	Private key
M	Modifier value to calculate a fingerprint
m	Number of starting points in one table
r	Number of tables
θ	Number of SHA2 output bits that must be zero
ℓ	Length in bits of the desired fingerprint
d	Number of leading zeros in disting. fingerprints
t	Represents the (average) length of a chain
t_{min}	Minimum length of a chain
t_{max}	Maximum length of a chain
L	Number of chains that are stored in the table
α	Expected number of chains in a table
β	Expected length of a chain

Algorithm 1 increments the right-most bits. This will reduce the number of chains in the table that might otherwise collide.

3.3 Constructing chains and tables

In order to construct chains we will make use of distinguished points. We opted for this approach because the generated tables are large, and we want to reduce the number of required lookups when using the table. This choice was also inspired by work of Nohl on the A5/1 cipher, where distinguished points were used for similar purposes [19]. When constructing a chain using distinguished points, we keep applying the reduction and PKHash function until we encounter a fingerprint with a given number of leading zero bits. This fingerprint is commonly called a distinguished endpoint and we will also call it a 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. The advantage of using distinguished points is that fewer table lookups have to be performed when inverting a fingerprint compared to using rainbow tables [4, 20].

To construct a single table, we pick m random fingerprints as starting points, denoted by k_1 to k_m. For each starting point, we execute the reduction and PKHash function until we get a fingerprint that starts with d bits. The combination of the reduction and PKHash function, which corresponds to function f in the background section, can be written as follows:

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

where $k_{i,j}$ is the j-th point in chain i. Each chain i starts with fingerprint $k_{i,1} = k_i$. The operator $\ll$ denotes a binary left shift, and the constant $32 \cdot 8$ corresponds to the length of the modifier. To detect possible loops, we limit the length of a chain to at most t_{max} elements. In case no distinguished fingerprint was found after t_{max} applications of the reduction and PKHash function then the chain is discarded. Optionally, it is also possible to set a minimum chain length t_{min}, to prevent the inclusion of short chains.

3.3.1 Storage. For every chain i we store the starting fingerprint k_i, the distinguished endpoint $k_{i,t}$, and the length of the chain t. By storing the length of the chain we can merge chain collisions by keeping only the longest chain. That is, when a new distinguished endpoint is already part of the table, we only store the longest chain. As is typical for time-memory trade-off attacks, the table is sorted based on the distinguished endpoints. This allows one to perform lookups of an endpoint in logarithmic time.

3.3.2 Multiple tables. When using multiple tables, each table uses a unique reduction function to reduce the number of chain collisions. This improves the success rate of table lookups [4, 23]. We can easily construct a unique reduction function per table by encoding the index of the table into the high order bits of the modifier M. After the bits that encode the index of the table, the output of the previous PKHash call is placed. In other words, for table n out of r, the combination of the reduction and PKHash function can be written as follows:

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

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

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

Here the operator | denotes the binary OR operation.

3.4 Precomputed table properties

To determine the efficacy of the constructed precomputed tables, we apply the analysis of Standeart et al. to our use case [23]. In our analysis we do not take into account chain collisions and corresponding merges. This simplifies the analysis at the cost of some reduction in precision [23, §8]. To also obtain precise estimates, we perform experiments using a proof-of-concept implementation.

3.4.1 Precomputation complexity. The cost of precomputing the table can be estimated by $C = r \cdot m \cdot \delta$ where δ can be approximated by 2^d (its exact value is in Appendix A). Recall that r denotes the number of individual tables being computed and m denotes the number of chains, i.e., starting points, in each table (see also Table 1). The result C equals the number of calls to PKHash. Since PKHash executes on average 2^θ SHA2 operations internally, the total number of expected SHA2 operations can be approximated by the formula $r \cdot m \cdot 2^{\theta+d}$.

3.4.2 Success rate. The expected success rate of finding an SAE-PK passphrase in a single table equals [23]:

$$SR \approx \frac{s(\gamma m)}{2^\ell} \tag{9}$$

See Appendix A for Standeart's et al. formula to calculate the function s. In case we use r different tables, where each table has a different reduction function to reduce the number of chain collisions, the probability of a successful lookup is [23]:

$$PS(r) = 1 - (1 - SR)^r \tag{10}$$

3.4.3 Storage requirements. Given that c bytes are needed to store one chain in the table, then the analysis in [23] shows that the number of bytes needed to store the table is bounded by:

$$c \cdot \frac{s(\gamma \cdot m)}{\beta} \cdot r \tag{11}$$

The value for β can be approximated by 2^d and its precise value based on [23] is listed in Appendix A.

3.4.4 Lookup complexity. The processing complexity, i.e., the expected number of calls to PKHash when looking up an element over all r tables, can be estimated using $r \cdot \beta$. 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 [23].

3.5 Estimations and experiments

3.5.1 Analytical analysis. Based on the previously derived properties, we can estimate the cost of breaking SAE-PK under its lowest security setting, namely when $Sec = 3$ and $\lambda = 12$. In this configuration, the derived parameters for PKHash are $\theta = 24$ and $\ell = 52$. Assuming we want a success rate of at least 50% when looking up an SAE-PK password in the precomputed table, we can construct $r = 2^{13}$ tables, where each table is constructed using $m = 2^{26}$ starting fingerprints. Additionally, we can give chains a maximum length of $t_{max} = 2^{16}$, and let a distinguished fingerprint start with $d = 13$ bits. Given that storing one chain requires 15 bytes, the storage requirement for all tables is less than 6 terabytes. To look up an SAE-PK passphrase under this configuration would require an expected 2^{26} calls to PKHash, which results in an expected 2^{50} SHA2 operations to look up a single SAE-PK password.

3.5.2 Time estimations. To estimate the computational time requirements for building the table and looking up elements, we first need to determine which SHA2 function is used in PKHash. Rather conveniently, the specific SHA2 hash function depends on the length of the public key, meaning the attacker can determine which SHA2 hash function will be used [26]. As a result, we can chose the most efficient SHA2 function supported by SAE-PK, and in our case we opted for SHA2-256.

We can now analyze the time required to build the table, assuming the weakest allowed SAE-PK security settings are used. Similar to the security analysis in the WPA3 certification, we take as reference a Bitcoin "hash miner" that is capable of performing 50 TeraHashes per second [26]. A Bitcoin miner internally performs a double SHA2-256 hash, meaning we would be able to perform at least 2^{46} SHA2-256 hashes per second. This implies that constructing the table would take at least 2^{30} seconds, which equals slightly more than 30 CPU years.

To look up an SAE-PK password in the precomputed table, we need to perform 2^{50} SHA2-256 operations. We assume that the attacker rents an Amazon `p4d.24xlarge` instance to look up passwords, which has 8 terabytes of storage and contains 8 NVIDIA A100 Tensor Cores, and hence has sufficient disk space to store the tables. This instance is able to calculate 2^{35} SHA2-256 hashes per second, meaning looking up a password would take at least 9 hours. At the time of writing, it costs less than $10 to rent a spot instance of this type for one hour, meaning the estimated cost of a password lookup would be roughly $90 under our assumptions.

All combined, if we construct a precomputed table for an SSID that is used by, for instance, one thousand networks, we can attack all those networks using this table. Based on our analysis, constructing the table would take 30 CPU years, and this process can be parallelized by using multiple devices. This implies that the amortized computational cost of attacking a single network becomes less than 12 days. In contrast, the current WPA3 specification assumes that attacking any network, independent of how common the SSID is, will cost roughly 48 CPU years.

3.5.3 Experiments. We implemented a proof-of-concept in C to confirm the principles behind the attack. Our current code only supports values for θ, ℓ, and d that are multiples of eight, which was done to simplify the implementation. To also allow fast simulations, we used parameters $\theta = 8$ and $\ell = 24$ in our experiments. For the construction of the precomputed table we used $d = 8$ for distinguished fingerprints, $t_{max} = 2^{11}$ as the maximum chain length, $m = 2^8$ starting points per table, and $r = 2^8$ individual tables (with each a different reduction function).

With the above parameters, we performed an experiment where we looked up 400 passwords in the constructed table. On average one lookup required $2^{15.67}$ calls to PKHash, while the expected number from our analysis is 2^{15}. We conjecture that this higher time complexity is due to false alarms during a password lookup. In particular, on average there were 59 false alarms during a password lookup, and every false alarm required roughly $2^\beta \approx 2^8$ extra PKHash calls. Under the above table configuration, the success rate of a password lookup was 34%, which is lower than our analytical prediction of 42%. We conjecture that this discrepancy is due to not taking into account chain collisions during the derivation of the analytical table properties: on average we observed that 38% of chains in a table were removed because they collided with other chains. Nevertheless, even if the success rate is lower than predicted, it shows the attack is feasible and must be taken into account when picking security parameters for a new SAE-PK network. We can also compensate for the lower practical success rate by generating bigger precomputed tables. Overall, these experiments confirm the correctness of our table construction.

3.6 Discussion

Our results illustrate the feasibility of time-memory trade-off attacks against SAE-PK. When using a common SSID, we strongly encourage users to use a parameter of $Sec = 5$, or a longer password by setting λ to 16 or higher, in order to mitigate time-memory trade-off attacks.

Our experiments showed that, when solely relying on distinguished points to construct the tables, there is a high amount of chain collisions while constructing the table, and that there is a non-negligible number of false alarms when looking up a password in the table. Both these factors decrease the efficacy of the time-memory trade-off. Inspired by [19], we consider it interesting future work to combine the distinguished point technique with rainbow tables to reduce these two observed downsides.

One limitation of our proof-of-concept is that it only supports parameters that are a multiple of eight. As a result, important future work is to perform simulations for longer SAE-PK passwords and for different table parameters.

4 CREATING PASSWORD COLLISIONS

In this section, we demonstrate how to create password collisions. That is, we create two networks with different SSIDs, but that both have the same SAE-PK password. This non-trivial to achieve

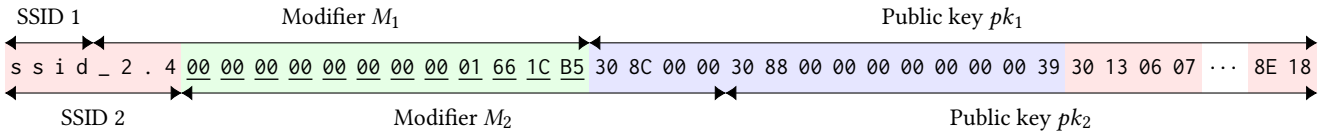

Figure 2: Input given to PKHash such that the SAE-PK password of two SSIDs are identical. The top shows how this input is split into the SSID, modifier, and public key for the first network. The bottom shows how this input is split for the second network. The green underlined bytes can be modified so that the hash output starts with sufficiently many zeros.

because the SSID is used as a salt in the PKHash function. Nevertheless, we accomplish this by manipulating the input given to the SHA2 function in PKHash, and by abusing flexibilities in how libraries parse public keys. All combined, this implies that a single precomputed time-memory trade-off table can be used against different SSIDs. Concretely, we will first recap the encoding of public keys, then abuse known issues how certain libraries parse public keys, and finally we explore new parsing issues.

4.1 Constructing identical PKHash inputs

In our attack, one SSID must be the prefix of the other SSID, and we will construct a modifier and public key so that both networks will have the same password. For instance, we will have two networks "`ssid`" and "`ssid_2.4`" for which we will construct a modifier M and public key pk such that the resulting SAE-PK password is identical. The core idea is to pick M and pk such that in both cases the PKHash algorithm gives exactly the same input to its internal hash function, meaning the resulting password is the same. The main challenge is to assure that the public key will be properly parsed in both cases.

Our construction for the modifier M and public key pk is shown in Figure 2. We let network 1 denote the network with the shorter SSID, and let network 2 denote the network with the longer SSID:

4.1.1 Modifier construction. The 16-byte modifier is always placed immediately after the SSID when constructing the input to the SHA2 function in PKHash. Therefore, to assure that the input is equal for both networks, the 16-byte modifier M_1 of network 1 must start with the remaining characters of the longer SSID. In our example, this means that M_1 must start with the ASCII encoding of "`_2.4`" (see Figure 2). For modifier M_2 of network 2, the last bytes of the modifier must equal the first few bytes of the public key of network 1, i.e., it must equal the first bytes of pk_1. In our example in Figure 2, this means that M_2 must end with the four bytes `30 8C 00 00`.

Assuming that the longer SSID has x more characters compared to the shorter SSID, then $16 - x$ bytes remain unconstrained in both M_1 and M_2. These unconstrained bytes will be picked such that the resulting output of PKHash starts with sufficiently many zeros as required by the security parameter *Sec*.

4.1.2 Public key length and placement. The public key is placed immediately after the modifier when constructing the input to the SHA2 function in PKHash. To assure that the resulting input has the same length for both networks, the public key of network 2 must be shorter than the public key of network 1. This can be achieved because public keys do not have a fixed length, for instance, it depends on the elliptic curve being used and how they are encoded in binary. The challenge is that both public keys start at different positions in the input to PKHash. To keep this input equal for both networks, this means that a valid public key for network 2 (pk_2) must start in the middle of the public key of network 1 (pk_1), as illustrated in Figure 2. To accomplish this we abuse flexibilities in the way that implementations parse public keys, which is described next.

4.2 Public key encoding and length fields

4.2.1 Public key encoding. The WPA3 specification states that the public key must be encoded according to RFC 5480 [21]. This means it must use the same format as the Subject Public Key Information field of X.509 certificates. The elliptic curve point inside this public key is encoded according to ANSI X9.63, but we first focus on the overall structure of the SubjectPublicKeyInfo and its (known) flexibilities.

The structure of the SubjectPublicKeyInfo field of X.509 certificates, i.e., the format of pk_1 and pk_2, is described using ASN.1 notation. In particular, the public key consists of two main components, which are defined as follows using ASN.1:

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
   algorithm         AlgorithmIdentifier,
   subjectPublicKey  BIT STRING
}
```

Remaining details of the encoding in ASN.1 are shown in Appendix B. Summarized, the above means that there is first some metadata that describes the type of elliptic curve, and the second part is the elliptic curve point encoded according to section 4.3.6 of the X9.62 [3] standard. The full ASN.1 structure is encoded to binary using DER (Distinguished Encoding Rules). Although in theory DER defines only one way to encode data, in practice some flexibility is possible when encoding the public key.

4.2.2 Variable length fields. The DER encoding uses a tag, length, value format. This is typically called a TLV format. The idea is that each element start with one or more bytes that define the type of the next element, then one or more bytes that define the length of the element, and finally the content of the element itself. The SubjectPublicKeyInfo field consists of a single TLV element, namely a sequence element. This in turn contains the two elements as shown previously. This outer sequence is encoded using a TLV element with as type the byte `30`, as is also shown in the start of both public keys in Figure 2.

The length in a TLV element can be encoded using one or more bytes. When the length of the element is smaller or equal to 127

Table 2: Behaviour of crypto libraries regarding the parsing of ASN.1 length fields inside TLV elements. The second column contains the function used to parse length fields, the third column contains the maximum number of bytes in a variable length field, the fourth whether an arbitrary number of leading zeros are allowed, and the fifth column contains whether overflows in the length are detected.

Library	ASN.1 length function	Max bytes	Leading zeros	Overflow
OpenSSL	`asn1_get_length`	`sizeof(long)`	Allowed	Detected
WolfSSL	`GetLength_ex`	`sizeof(int)`	Disallowed	Detected
GnuTLS (libtasn1)	`asn1_get_length_der`	`sizeof(unsigned int)`	Allowed	Detected
MatrixSSL	`getAsnLength`	4	Disallowed	Detected

Table 3: Behaviour of crypto libraries regarding the parsing of X9.62 encoded elliptic curve points and the X.509 SubjectPublicKeyInfo structure. The second column contains the function that we tested, the third column whether the length of the elliptic curve point is restricted, and the fourth whether the library accepts or rejects an x or y coordinate that is bigger than the prime of the curve. The fifth column contains the function used to parse the whole public key as included in the PKHash function.

Library	X9.62 point decoding function	Len. check	Added prime	SubjectPublicKeyInfo parser
OpenSSL	`ec_GFp_simple_oct2point`	Yes	Rejected	`d2i_PUBKEY`
WolfSSL	`wc_ecc_import_x963`	No	Accepted	`wc_EccPublicKeyDecode`
GnuTLS	`gnutls_pubkey_import_ecc_x962`	No	Accepted	`gnutls_pubkey_import`
MatrixSSL	`psEccX963ImportKey`	No	Accepted	`psParseSubjectPublicKeyInfo`

bytes, then the length can be directly encoded as a single byte. When the length is bigger than 127, it can be encoded using a variable number of bytes: in that case the first byte has its high-order bit set and the remaining bits define how many bytes encode the length. For instance, if the first byte is `0x84`, this means that the following 4 bytes encode the length of the element.

Against older libraries and implementation, it has been shown that variable length encoding can be abused to hide arbitrary data in the length field of an element [15]. For instance, we can specify the following element and length field:

```
30 8C 00 00 30 88 00 00 00 00 00 00 00 39
```

Here we start a sequence element (byte `30`) and then specify that the length is encoded using 12 bytes (byte `8C`). Assuming the target can handle at most 64-bit integers, this causes the first 4 length bytes (namely `00 00 30 88`) to be ignored when reading the length due to an overflow. Put differently, some implementations will think this encodes a length of 56 (`39` in hexadecimal). We can abuse this to put arbitrary data within a public key, without affecting how the public key is parsed. In particular, in our password collision attack, we can abuse this to hide a valid start of public key pk_2 within the first length field of another public key pk_1 (see Figure 2).

4.2.3 Implementation analysis. A notable implementation that was vulnerable to this was the TLS library of Mozilla, called Network Security Services (NSS). This bug was assigned CVE-2014-1569 and allowed an adversary to encode arbitrary data within the length field of an element, which was abused to attack the RSA cipher [15].

To test for the feasibility of this construction in practice, we focus on Linux's hostap daemon. This is, to the best of our knowledge, the only open source implementation that supports SAE-PK at the time of writing. This daemon can be used with various TLS libraries, including OpenSSL, WolfSSL, and GnuTLS. Additionally, old versions of hostap were also ported to work with MatrixSSL [16]. We therefore focus on these four TLS libraries.

The result of our experiments is shown in Table 2. It shows the internal function that is used to parse the length of TLV elements, the maximum number of bytes that can be used to encode the length, whether an arbitrary number of leading zeros in the length encoding are allowed, and whether overflows are detected. Interestingly, *modern* versions of these libraries are no longer vulnerable since they all detect overflows. Nevertheless, we conjecture that new implementations of SAE-PK may repeat this mistake.

4.2.4 Creating a valid SAE-PK password. We now know one technique to create two identical inputs to PKHash for two different SSIDs. The last step is to assure that the resulting output of the PKHash starts with sufficiently many zeros (recall Section 2.2). To achieve this, we can vary the value of modifier M_1 excluding the bytes that overlap with the longer SSID. These bytes correspond to the green underlined bytes in Figure 2. This implies that the two SSIDs must only differ by a certain amount in length, otherwise there is no space left to assure that the output of PKHash starts with sufficiently many zeros. In the next section we discuss a method to overcome this limitation.

4.2.5 Experiments. To confirm that the same SAE-PK password is valid for two different SSIDs after applying the construction shown in Figure 2, we performed experiments using Linux's hostap daemon. In particular, we the compiled client with an OpenSSL library that was modified to accept overflows in the `asn1_get_length` function. For the access point we used hostapd, which was modified so that it can be configured to advertise and distribute the SAE-PK public key using a non-standard encoding that is provided by the attacker. We then created two networks, where the SSID of the first network is the prefix of the other, and configured both with

the constructed public key encodings and modifiers as shown in Figure 2. This confirmed that the hostap client can successfully connect to the two different SSIDs using the same SAE-PK password.

4.3 Variations in point encoding

In order to assure that the output of PKHash starts with sufficiently many zeros, our previous approach was to change the modifier M. However, in our password collision attack, this limits the maximum difference in length between the two SSIDs. To overcome this problem, one method is to instead generate new public keys until we have found one that results in a PKHash output with enough zeros. The downside of this approach is that generating or modifying a public key can result in a significant overhead.

An alternative option is to change how the elliptic curve point of the public key is encoded. Inside the SubjectPublicKeyInfo structure, the subjectPublicKey field contains the point as an `ECPoint` type:

```
ECPoint ::= OCTET STRING
```

The value of the `ECPoint` element is the binary encoding of an elliptic curve point according to the rules of section 4.3.6 in the X9.62 [3] standard. The WPA3 certification specifies that this point must be encoded in compressed format when constructing the PKHash input [26]. An example encoding of a compressed elliptic curve point is:

```
03 // Type of this element is a BIT STRING
22 // Length of the BIT STRING element
00 // Number of unused bits in the final byte
03 // We are encoding a compressed point
24 40 58 8D A5 02 95 5C B4 C8 49 85 FD 1F 7D A7
1A 17 6B 72 9B 06 EB 72 20 70 00 A8 35 45 8E 18
```

The first byte (here `03`) defines that this element is an `BIT STRING`, the second byte (here 22) denotes the length of the element, and the third byte (here `00`) defines that all bits in the last byte of the octet string are used. Note that a `BIT STRING` is used to encode the point: the idea is that, although the `ECPoint` itself is an `OCTET STRING`, it is directly mapped to a `BIT STRING` when placed inside the subjectPublicKey field [21]. After these three bytes (`03 22 00`), the X9.62 encoding of the elliptic curve point starts.

The X9.62 and X9.63 standard both define an identical method to encode the elliptic curve point, but unfortunately these standards not freely accessible. As a result, in practice the open SECG standard [6, §2.3.3] is often referenced instead, because it contains a compatible description of how to encode a (compressed) elliptic curve point. Fortunately, all these standard encode points in the same way, with the exception that SECG does not define the hybrid point encoding method. In practice, the libraries that we inspected mainly support compressed and uncompressed point encoding, with only few libraries supporting hybrid point encoding.

An uncompressed elliptic curve point starts with the byte `04` and is followed by the x and y coordinate of the point. A compressed points starts with either the byte `02` or `03` and is followed by the x coordinate. The value of the y coordinate can then be derived from the x coordinate and from the first byte. That is, the first byte `02` or `03` of an compressed point defines which of the two possible y coordinates should be taken.

Instead of constantly generating new public keys until the PKHash output contains enough zeros, we can instead change how a point is encoded. In particular, we can add the prime p of the elliptic curve field to the encoded coordinates. This results in an equivalent representation of the point since all operations are done modulo p when processing the point. Although the X9.62, X9.63, and SECG standard state that implementations must not accept such encodings, in practice we did observe that WolfSSL, GnuTLS, and MatrixSSL accepts such encodings (see Table 3). This provides an efficient method to change how the public key is encoded. For instance, the following encodes the same point as above:

```
03 24 00 // Type and length header
03       // Compressed point
00 01    // Prime p was added to x coordinate
24 40 58 8C A5 02 95 5D B4 C8 49 85 FD 1F 7D A7
1A 17 6B 73 9B 06 EB 72 20 70 00 A8 35 45 8E 17
```

Some implementations do check that a coordinate consists of an even number of bytes, but that can easily be satisfied by pretended an extra zero byte.

Note that when the AP transports the key to the client inside a FILS element, it is not specified whether the elliptic curve public key must be encoded in compressed or uncompressed form, and in general no checks are performed on the transported public key. As a result, when an adversary creates a rogue AP, arbitrary encodings of the public key can be sent to the client, and it will depend on the crypto library of the client which encodings are accepted (see Table 3).

4.4 Practical applications

By constructing SAE-PK password collisions for two SSIDs, we can build a precomputed time-memory trade-off table that can be used to attack two SSIDs. This further increases the motivation for an attacker to invest the cost of performing this precomputation, since it can subsequently be used to attack a larger number of networks.

5 DISCUSSION

To prevent our discovered attacks, we recommend the following backwards-compatible defenses, and we also propose a design change to SAE-PK:

Backwards-compatible defenses. To prevent SAE-PK password collisions, a client should only allow one possible encoding of the public key. To mitigate time-memory trade-off attacks, the network should use a security parameter of $Sec = 5$, or should use passwords of length $\lambda = 16$ or longer. Alternatively, the network administrator can decide to use a unique SSID, since adversaries are less likely to create a computed table for unique SSIDs.

Design changes. The input given to the SHA2 function inside PKHash should be updated to start with a single byte that represents the length of the SSID. This effectively forces an attacker to commit to a specific SSID length when constructing the precomputed time-memory trade-off table, and prevents our SAE-PK password collision attacks. Unfortunately, this change is not backwards-compatible.

6 RELATED WORK

The older WPA2 protocol was quickly found to be vulnerable to offline dictionary attacks [17]. It uses PBKDF2 to derive a Pairwise Master Key (PMK) from the password and SSID, and then uses the 4-way handshake to combine the PMK with two random nonces to derive a unique session key. An adversary can capture the 4-way handshake and then brute-force the password until a correct session key is found. Because of the inclusion of two random 32-byte nonces to derive the session key, it is infeasible to perform a time-memory trade-off attack against the 4-way handshake. Nevertheless, because offline brute-forcing of the passphrase was slow on personal computers, lookup tables were created to speedup this process [22]. In particular, the tables were used to avoid the computationally costly PBKDF2 hash by mapping an SSID and password to the corresponding Pairwise Master Key (PMK). These tables covered the top 1 000 SSIDs using a dictionary of 172 000 possible passwords, which was later extended to a dictionary of one million words. Their resulting lookup tables where 7 GB and 33 GB large. Due to the rise of GPU-based password cracking, the need for such tables has decreased.

Regarding the security of WPA3, Vanhoef and Ronen discovered side-channel vulnerabilities in the first version of WPA3 [25], which have meanwhile been addressed in the latest version of WPA3 [26]. De Almeida Braga et al. later discovered that some implementations did not defend against these side-channel flaws even after the public disclosure of these flaws [9].

Hellman introduced the time-memory trade-off attack [12]. These were later improved by Oechslin to what are called rainbow tables [20]. A combination of distinguished points and rainbow tables were used by Nohl to carry out a time-memory trade-off attack against the A5/1 cipher [19]. This attack was later made more efficient by making use of FPGAs [14].

Flexibilities in parsing public keys were previously abused to attack RSA ciphers [11, 24]. In this line of work it was also discovered that some libraries accept arbitrary parameters in the algorithm identifier, which was assigned CVE-2018-16151, and they also discovered other parsing flexibilities in cryptographic libraries [7, 8].

7 CONCLUSION

By performing a time-memory trade-off attack against WPA3's SAE-PK protocol, it becomes possible to break a password using a lower amortized computational cost than previously assumed. In order to mitigate such attacks, we recommend that networks with a common SSID set the security parameter *Sec* to 5. Alternatively, such networks can decide to use a longer SAE-PK password at the cost of decreased usability, or they can decide to change the SSID to a more unique one. Finally, we consider it interesting future work to combine the technique of distinguished points with rainbow tables to further optimize our time-memory trade-off attack against SAE-PK in practice.

ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. Mathy Vanhoef holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

REFERENCES

[1] [n. d.]. WiGLE Stats. Retrieved 26 December 2021 from https://wigle.net/stats.
[2] IEEE Std 802.11. 2020. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.*
[3] ANSI. 1999. X9. 62-1998: Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA). *American National Standards Institute (ANSI)* (1999).
[4] Johan Borst. 2001. *Block ciphers: Design, analysis and side-channel analysis.* Ph. D. Dissertation. KU Leuven, Leuven, Belgium.
[5] Johan Borst, Bart Preneel, and Joos Vandewalle. 1998. On the time-memory tradeoff between exhaustive key search and table precomputation. In *Symposium on Information Theory in the Benelux*. TECHNISCHE UNIVERSITEIT DELFT, 111–118.
[6] D Brown. 2009. Standards for efficient cryptography, SEC 1: elliptic curve cryptography. *Released Standard Version* 1 (2009).
[7] Sze Yiu Chau. 2019. A Decade After Bleichenbacher'06, RSA Signature Forgery Still Works. *Black Hat USA* (2019).
[8] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing semantic correctness with symbolic execution: A case study on PKCS# 1 v1.5 signature verification. In *Network and Distributed Systems Security (NDSS) Symposium 2019*.
[9] Daniel de Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. 2020. Dragonblood is still leaking: Practical cache-based side-channel in the wild. In *Annual Computer Security Applications Conference*. 291–303.
[10] Dorothy E Denning and Peter J Denning. 1979. Data security. *ACM Computing Surveys (CSUR)* 11, 3 (1979), 227–249.
[11] Hal Finney. 2006. Bleichenbacher's RSA signature forgery based on implementation error. Retrieved 21 November 2021 from https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/.
[12] Martin Hellman. 1980. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory* 26, 4 (1980), 401–406.
[13] Simon Josefsson. 2006. The Base16, Base32, and Base64 Data Encodings. RFC 4648. https://doi.org/10.17487/RFC4648
[14] Maria Kalenderi, Dionisios Pnevmatikatos, Ioannis Papaefstathiou, and Charalampos Manifavas. 2012. Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS. In *22nd International conference on field programmable logic and applications (FPL)*. IEEE, 747–753.
[15] Adam Langley. 2014. PKCS# 1 Signature Validation. Retrieved 21 November 2021 from https://www.imperialviolet.org/2014/09/26/pkcs1.html.
[16] Jouni Malinen. 2007. oepnssl vs. matrixssl. Retrieved 5 January 2022 from http://lists.shmoo.com/pipermail/hostap/2007-October/016357.html.
[17] Robert Moskowitz. 2003. 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.
[18] Yoav Nir, Simon Josefsson, and Manuel Pégourié-Gonnard. 2018. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. RFC 8422. https://doi.org/10.17487/RFC8422
[19] Karsten Nohl. 2010. Attacking phone privacy. *Black Hat USA* (2010), 1–6.
[20] Philippe Oechslin. 2003. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*. Springer, 617–630.
[21] Tim Polk, Russ Housley, Sean Turner, Daniel R. L. Brown, and Kelvin Yiu. 2009. Elliptic Curve Cryptography Subject Public Key Information. RFC 5480. https://doi.org/10.17487/RFC5480
[22] RenderMan, Joshua Wright, Thorn, Dragorn, H1kari, and Twitchy. 2006. Church of Wifi WPA-PSK Lookup Tables. Retrieved 26 December 2021 from https://www.renderlab.net/projects/WPA-tables/.
[23] Francois-Xavier Standaert, Gael Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. 2002. A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*. Springer-Verlag, Berlin, Heidelberg, 593–609.
[24] Filippo Valsorda. 2016. Bleichenbacher'06 signature forgery in Python-RSA. Retrieved 21 November 2021 from https://blog.filippo.io/bleichenbacher-06-signature-forgery-in-python-rsa/.
[25] Mathy Vanhoef and Eyal Ronen. 2020. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In *IEEE Security & Privacy (SP)*. IEEE.
[26] Wi-Fi Alliance. 2020. WPA3 Specification Version 3.0. Retrieved 20 November 2021 from https://www.wi-fi.org/file/wpa3-specification.

A USED FORMULAS

We used the following formulas from [23] to analyze the impact of the parameters that are used to construct the precomputed table:

$$P_2(l) \approx \left(1 - \frac{2^{k-d}}{2^k - \frac{l-1}{2}}\right)^l \approx e^{-\frac{l}{2^d}} \tag{12}$$

$$P_1(l) = 1 - P_2(l) \tag{13}$$

$$\gamma = P_1(t_{max}) - P_1(t_{min} - 1) \approx P_1(t_{max}) \tag{14}$$

$$t = \frac{t_{max} + t_{min}}{2} \tag{15}$$

$$x = \frac{2^{k-d}}{2^k - \frac{t}{2}} \tag{16}$$

$$= 1 - e^{-\frac{t_{max}}{2^d}} \tag{17}$$

$$\beta = \frac{(1-x)^{t_{min}-2} \cdot (t_{min} + \frac{1-x}{x}) - (1-x)^{t_{max}-1} \cdot (t_{max} + \frac{1}{x})}{\gamma} \tag{18}$$

$$s(j) = 2^k \cdot \left(1 - \left(\frac{2^k}{-\beta j + \beta^2 j + K}\right)^{\frac{1}{\beta-1}}\right) \tag{19}$$

$$K = 2^k \cdot \left(1 - \frac{s(0)}{2^k}\right)^{1-\beta} \tag{20}$$

$$s(0) = 0 \tag{21}$$

$$\delta = t_{max} \cdot (1 - P_1(t_{max})) + \beta \cdot P_1(t_{max}) \tag{22}$$

For the parameter k, which in the work of Standaert et al. represented the key length of the block cipher, we used the value ℓ.

B PUBLIC KEY ENCODING

The AlgorithmIdentifier field is defined as follows:

```
AlgorithmIdentifier  ::=  SEQUENCE  {
   algorithm   OBJECT IDENTIFIER,
   parameters  ANY DEFINED BY algorithm OPTIONAL
}
```

The possible values for the algorithm identifier are id-ecPublicKey, id-ecDH, or id-ecMQV. The identifier value and parameters for id-ecPublicKey are:

```
id-ecPublicKey OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) ansi-X9-62(10045)
     keyType(2) 1 }

ECParameters ::= CHOICE {
   namedCurve         OBJECT IDENTIFIER
   -- implicitCurve   NULL
   -- specifiedCurve  SpecifiedECDomain
}
```

Note that under RFC 5480 it is not allowed to use an implicit or specified curve, hence why these two options are commented out using two hyphens. The namedCurve can have various values, one common example being curve secp256r1, which is also known as NIST P-256 or prime256v1 [18]. This curve is represented using the following object identifier:

```
secp256r1 OBJECT IDENTIFIER ::= {
  iso(1) member-body(2) us(840) ansi-X9-62(10045)
    curves(3) prime(1) 7
}
```

There's also the options id-ecDH and id-ecMQV that restrict the usage of the given public key to the ECDH or the Elliptic Curve Menezes-Qu-Vanstone key agreement algorithm. The object identifier varies based on the algorithm, but the parameters are always ECParameters similar to with id-ecPublicKey.

The subjectPublicKey from SubjectPublicKeyInfo is the ECC public key. ECC public keys have the following syntax:

```
ECPoint ::= OCTET STRING
```

Here ECPoint is the octet representation of an elliptic curve point as defined in the X9.62 [3] standard. According to RFC5480, implementations must support uncompressed forms and may support compressed forms [21]. Note that the WPA3 standard requires support of the compressed form [26].