KEY RECOVERY IN PASSWORD-BASED AES-GCM WITH PARTITIONING ORACLE ATTACKS

Christian James Tan[1], Xu Jingxin[2], Ruth Ng[3], Choo Jia Guang[3]
[1] NUS High School of Mathematics and Science, 20 Clementi Avenue 1, Singapore 129957
[2] Anglo-Chinese School (Independent), 121 Dover Road, Singapore 139650
[3] DSO National Laboratories, 12 Science Park Drive, Singapore 118225

Abstract
While many cryptographic primitives are well tried and well tested, new attacks against them still pop up from time to time. This report shines a light on a recent attack on non-key-committing authenticated encryption schemes called the partitioning oracle attack. We describe the conditions required for the attack to be feasible, and illustrate the theory in practice by using this attack on a toy messaging application which uses AES-GCM with MAC-based authentication. We hope this paper can act as a resource for others to better understand the details of this attack, and to provide an environment for researchers to play with the attack themselves.

1. Introduction
Authenticated encryption (AE) schemes have long been favoured in many applications for their guarantee of privacy (encrypted data reveals no information of the plaintext data) and authenticity (encrypted data cannot be modified in transit or forged without the receiving party noticing, resulting in an invalid decryption).

However, in the case of non-key-committing AE schemes; that is, schemes where there exist ciphertexts that can be decrypted validly by the receiving party under several distinct keys under the same conditions, these become vulnerable to a chosen ciphertext attack that recovers the secret key. This attack was termed the **partitioning oracle attack** by Len et al. (2021) in their seminal paper on the topic.

Thus, in this paper, we attempt to define precisely the conditions required for the partitioning oracle attack to be feasible, including both abstract requirements of the AE scheme itself (e.g. the ability to create colliding ciphertexts) as well as those pertaining to a concrete implementation of the attack (e.g. the distribution of the key should be non-uniform, such as that of a passphrase). We additionally illustrate the significant speedup of such an attack mathematically against AE schemes with polynomial-based message authentication codes (MACs) as well as through a toy example utilising AES-GCM, to better demonstrate the inner workings of the attack.

2. Background & Theory

2.1. Authenticated Encryption

Symmetric-key encryption (SE) schemes are cryptographic schemes that utilise the same secret key for both encryption & decryption. These schemes aim to achieve two main properties:

1. **Correctness**: the scheme works correctly; in the sense that encrypted messages can be decrypted back to their original plaintext messages.
2. **Confidentiality**: the encrypted messages grant no information of the original plaintext without knowing the secret key.

Authenticated encryption (AE) schemes fulfill both of these criteria, and one more:

3. **Authenticity**: encrypted messages cannot be forged or tampered with without being detected.

Most schemes (tag-based encrypt-then-MAC) achieve this by combining an already existing SE scheme (to achieve correctness & confidentiality) with a **Message Authentication Code** (MAC) function: a function that takes in the secret key & some message to authenticate and generates a MAC for it; acting as a "checksum" to ensure that the message could not have been tampered with in transit (data integrity) and that it was sent from a verifiable source (as the sender must know the secret key). It does so by calculating the MAC on encryption, sending it with the encrypted data, and recalculating the MAC on decryption before comparing with the original MAC. If they fail to match, that would imply that the encrypted message/MAC had been tampered with in transit or was not sent from a verifiable source.

More formally, for an AE scheme AE with

- secret key K,
- underlying nonce-based symmetric encryption scheme $\mathsf{AE}.n = (\mathrm{Enc}(K, N, M), \mathrm{Dec}(K, N, C))$, with nonce N, plaintext message M & encrypted ciphertext C.
  - In order for the resultant AE scheme to achieve confidentiality and correctness, the underlying scheme $\mathsf{AE}.n$ must achieve both as well (e.g. $M = n.\mathrm{Dec}(K, N, n.\mathrm{Enc}(K, N, M))$)
- MAC function $\mathsf{AE}.\mathrm{genMAC}(K, N, C)$,

the following pseudocode describes its encryption ($\mathsf{AE}.\mathrm{AuthEnc}$) and decryption ($\mathsf{AE}.\mathrm{AuthDec}$) routines, with plaintext message M, encrypted ciphertext C and MAC T.

$\mathsf{AE}.\mathrm{AuthEnc}(K, N, M)$
$C \leftarrow \mathsf{AE}.n.\mathrm{Enc}(K, N, M)$
$T \leftarrow \mathsf{AE}.\mathrm{genMAC}(K, N, C)$
Return C, T

$\mathsf{AE}.\mathrm{AuthDec}(K, N, C, T)$
$M \leftarrow \mathsf{AE}.n.\mathrm{Dec}(K, N, C)$
$T' \leftarrow \mathsf{AE}.\mathrm{genMAC}(K, N, C)$
If $T \neq T'$ then **Return** $\perp$
Return M

Table 1: Pseudocode for $\mathsf{AE}.\mathrm{AuthEnc}$ & $\mathsf{AE}.\mathrm{AuthDec}$. note that $\perp$ represents an error state.

As such, $\mathsf{AE}.\mathrm{AuthDec}$ "fails" (by returning $\perp$) if the sent MAC T and recalculated MAC T' do not match.

2.1.1. Additional Data

Most AE schemes support authenticating some additional unencrypted data A alongside the ciphertext when encrypting and decrypting; such schemes are known as **Authenticated Encryption with Additional Data** (AEAD) schemes:

$$\mathsf{AE}.\mathrm{genMAC}(K, N, C) \Rightarrow \mathsf{AEAD}.\mathrm{genMAC}(K, N, A, C)$$
$$\mathsf{AE}.\mathrm{AuthEnc}(K, N, M) \Rightarrow \mathsf{AEAD}.\mathrm{AuthEnc}(K, N, A, M)$$
$$\mathsf{AE}.\mathrm{AuthDec}(K, N, C) \Rightarrow \mathsf{AEAD}.\mathrm{AuthDec}(K, N, A, C)$$

For the purposes of this paper, all AE schemes are assumed to be AEAD schemes, as the presence of additional data has no effect on the partitioning oracle attack.

2.1.2. Key-Commitment

Although the MAC function may guarantee authenticity under honest conditions, the concept may break down under dishonest ones (such as the recipient party decrypting under a different key). We characterise this specific situation using the notation & definitions from Menda et al. (2023):

An AE scheme AE_k^* is considered **key-committing** (or CMT_k^* secure) if all ciphertexts can only be decrypted correctly with their respective keys. That is, notating the space of ciphertexts as $\mathcal{C}$ & the keyspace as $\mathcal{K}$:

$$\forall C \in \mathcal{C}\ \exists! K \in \mathcal{K} \text{ such that } \mathsf{AE}_k^*.\mathrm{AuthDec}(K, N, C) \neq \perp$$

for some fixed nonce N. If, instead, there exists one or more ciphertexts that decrypt validly under several keys, that is:

$$\exists C \in \mathcal{C}, \exists k_1, k_2 \in \mathcal{K} \text{ such that } \mathsf{AE}.\mathrm{AuthDec}(k_1, N, C) \neq \perp,$$
$$\mathsf{AE}.\mathrm{AuthDec}(k_2, N, C) \neq \perp$$

then such a scheme is considered **non-key-committing**. Interestingly, many common AE schemes are non-key-committing (GCM-SIV, ChaCha20-Poly1305, (Menda et al., 2023)).

However, simply being non-key-committing is not sufficient (but is necessary) for the partitioning oracle attack. As such, we introduce two more notions, adapted from Len et al. (2021):

An AE scheme AE is considered vulnerable to a **targeted multi-key collision attack** (TMKC) if an adversary $\mathcal{A}$ is able to construct a ciphertext $C \in \mathcal{C}$ & nonce N pair such that

$$\mathsf{AE}.\mathrm{AuthDec}(k, N, C) \neq \perp \ \forall k \in \mathbb{K}$$

for some target subset $\mathbb{K}$ of the keyspace $\mathcal{K}$. If the adversary can only construct such a pair for a random subset $\mathbb{K}$ of $\mathcal{K}$ such that $|\mathbb{K}| > \kappa$ for some κ, then the scheme is considered vulnerable to an **untargeted multi-key collision attack** (MKC_κ).

Although the scope of this may seem a bit limited (as having dishonest parties in symmetric encryption is rare), this notion of non-key-commitment is central to the execution of the partitioning oracle attack.

2.2. The Partitioning Oracle Attack

Abstractly speaking, the partitioning oracle attack enables an adversary $\mathcal{A}$ to recover the secret key of some AE scheme AE with ciphertext space $\mathcal{C}$ & keyspace $\mathcal{K}$ (with distribution specified by the probability mass function $F_{\mathcal{K}}$) under the following conditions:

1. AE is vulnerable to a TMKC attack.
2. $\mathcal{A}$ has access to an oracle $\mathcal{O}$ that accepts ciphertexts and outputs whether the decryption with the secret key K succeeds.

Specifically,

1. $\mathcal{A}$ uses the TMKC attack to construct a ciphertext C that can decrypt validly under some $\mathbb{K} \subset \mathcal{K}$ where $\Pr(K \in \mathbb{K} \mid K \in \mathcal{K}) = \frac{1}{2}$, i.e. K has a 50% chance to be in $\mathbb{K}$.
   - e.g. if $\mathcal{K}$ follows a *uniform distribution*, this is equivalent to constructing $\mathbb{K}$ such that $|\mathbb{K}| = \frac{1}{2}|\mathcal{K}|$.
2. It sends C to $\mathcal{O}$ to check if $K \in \mathbb{K}$ (in which case the decryption will succeed), thereby partitioning $\mathcal{K}$ into $\mathbb{K}$ and $\mathcal{K} \setminus \mathbb{K}$.

3. It restricts $\mathcal{K}$ to the set that contained K ($\mathbb{K}$ if it succeeded, else $\mathcal{K} \setminus \mathbb{K}$), before going back to 1.

3. Implementation

3.1. Keyspace Distribution

An important thing to note about the implementation of the attack is the *probability distribution* of the keyspace $\mathcal{K}$. Assuming that the TMKC attack runs in $O(G(|\mathbb{K}|))$ of the size of the subset $\mathbb{K}$ for some function G, we would want to minimise $|\mathbb{K}|$ to have the TMKC attack run as efficiently as possible (and thus the entire attack as a whole as well).

Without loss of generality, we take $|\mathbb{K}| \leq |\mathcal{K} \setminus \mathbb{K}|$, as if $|\mathbb{K}| > |\mathcal{K} \setminus \mathbb{K}|$ then we can reassign $|\mathbb{K}'| = |\mathcal{K} \setminus \mathbb{K}| \Rightarrow |\mathbb{K}'| \leq |\mathcal{K} \setminus \mathbb{K}'|$. As such, the maximum size for $|\mathbb{K}|$ is when

$$
\begin{aligned}
|\mathbb{K}| &= |\mathcal{K} \setminus \mathbb{K}| = |\mathcal{K}| - |\mathbb{K}| \\
2|\mathbb{K}| &= |\mathcal{K}| \\
\therefore |\mathbb{K}| &= \frac{1}{2}|\mathcal{K}|
\end{aligned}
$$

This is exactly the case as when $\mathcal{K}$ had a uniform distribution (in the example in Section 2.2), which implies that the uniform distribution has the worst case performance for the attack.

As such, to speed up the attack, we want the keyspace $\mathcal{K}$ to be very "non-uniform", i.e. have $|\mathbb{K}|$ as small as possible. An avenue for this would be the distribution of *passwords*: These tend to follow a Zipfian distribution (Wang et al., 2014) due to their relation to natural language. As such, $|\mathbb{K}|$ can be $\ll \frac{1}{2}|\mathcal{K}|$: For the dataset we utilised[1], $|\mathcal{K}| = 100000, \text{but } |\mathbb{K}| \approx 6122$, significantly smaller than $\frac{1}{2}|\mathcal{K}| = 50000$.

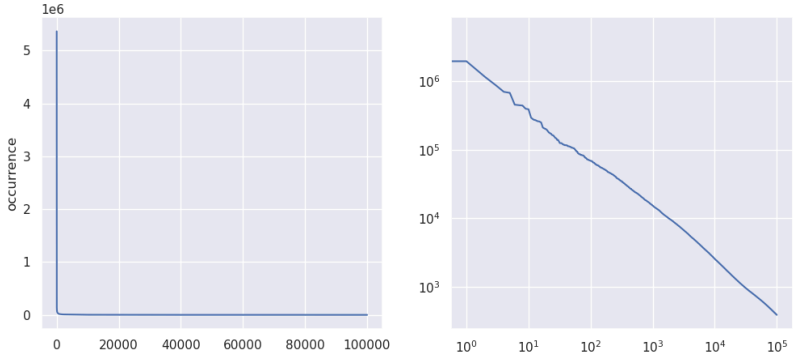

Figure 1: Frequencies of passwords in the dataset. As the log-log graph (left) is approximately linear, it approximately follows Zipf's Law.

[1] https://github.com/ignis-sec/Pwdb-Public/blob/master/statistical-lists/occurrence.100K.txt

3.2. Performance

As mentioned earlier, the performance of the partitioning oracle attack as a whole is dependent on the performance of the underlying TMKC attack that is utilised. More specifically, assuming that the TMKC attack runs in $O(G(|\mathbb{K}|))$ of the size of the subset $\mathbb{K}$ (from Section 3.1), then the entire partitioning oracle attack runs in

$$O\left(\sum_{i=1}^{\log n} \frac{G(n)}{2^i}\right)$$

where $n = |\mathcal{K}|$, equivalent to partitioning the keyspace $\mathcal{K}$ exactly in half each iteration (the worst case).

As the TMKC attack itself is dependent on the scheme it is attacking, we will be focusing on the most efficient & common example: for schemes that utilise (Carter-Wegman) polynomial based MACs (e.g. GCM's GHASH, Poly1305), the TMKC attack is equivalent to running polynomial interpolation (to construct the ciphertext that passes through each evaluation point at the MAC). As such, the runtime of the partitioning oracle attack becomes

$$\sum_{i=1}^{\log n} \frac{O(n^2)}{2^i} = O(n^2)$$

For naïve polynomial interpolation. If using fast polynomial interpolation (which runs in $O(n \log^2 n)$), we obtain

$$\sum_{i=1}^{\log n} \frac{O(n \log^2 n)}{2^i} \approx O(n \log^2 n)$$

3.2.1. Dictionary Attack

A **Dictionary Attack**, similarly to the partitioning oracle attack, enables an adversary $\mathcal{A}$ to recover the secret key of some AE scheme AE with ciphertext space $\mathcal{C}$ & keyspace $\mathcal{K}$ alongside a decryption oracle $\mathcal{O}$ that again takes ciphertexts and outputs whether they decrypt correctly or not. It does so by simply "brute-forcing" all $k \in \mathcal{K}$, from most probable to least probable. Trivially, this runs in $O(n)$ where $n = |\mathcal{K}|$. However, this means that the partitioning oracle attack runs *slower* than the basic dictionary attack, as even the best performance of $O(n \log^2 n) < O(n)$.

While this is true, it does not mean that the dictionary attack is strictly better than the partitioning oracle attack. Specifically, looking at the number of oracle queries,

- the dictionary attack takes $O(n)$ queries, as it must query the oracle with each key one by one (akin to linear search),
- but the partitioning oracle attack only takes $O(\log n)$ *queries*: as it halves the keyspace each time (akin to binary search).

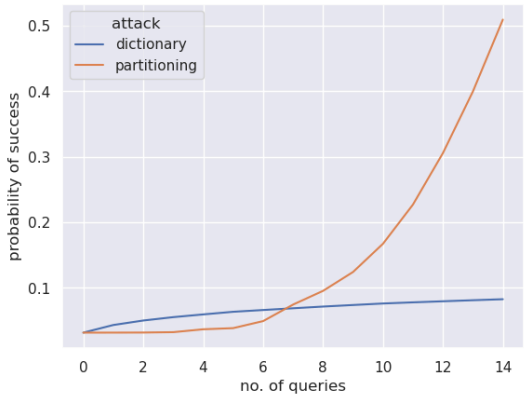

Figure 2: Graph of probability of recovering the key against the number of decryption oracle queries. At 14 queries, the partitioning oracle attack achieves > 50%, while the dictionary attack is still < 10%.

In concrete applications of these attacks, the oracle tends to be some form of server (e.g. Section 4), thereby making oracle queries "expensive": not only may they take very long to execute, but sending large volumes of queries may become very noticeable by the receiving party (perhaps through the excessive load). As such, if we disregard the "computation phase" (running of the TMKC attack), perhaps through precomputing the ciphertexts beforehand, the overall runtime does reduce to $O(\log n)$:

$$\sum_{i=1}^{\log n} \frac{1}{2^i} = O(\log n)$$

which is faster than the dictionary attack's $O(n)$.

4. Demo

4.1. Description

For our toy example, we will be utilising a simple messaging application (written with python `socket`) that encrypts and send messages between two parties who want to maintain confidentiality over a public network. The application utilises AES-GCM, keyed with some preshared passphrase sampled from the dataset used in Section 3.1.

Additionally, we have an "attacker dashboard" to aid visualisation of the attack, that was built with Websockets & a web-based UI.

4.1.1. Fulfilment of Conditions

This simple example fulfils all conditions we have outlined above, beginning with the abstract:

1. The underlying scheme used is AES-GCM, which is not only non-key-committing but also vulnerable to a TMKC attack.
2. There exists a decryption oracle: each packet is responded to with an acknowledgement (ACK) from the recipient. In an attempt to mitigate an adversary from using such as a decryption oracle, the app returns the ACK regardless of whether the packet decrypts successfully. However, if the packet were to decrypt correctly, yet still error before sending the ACK (in this case through incorrectly decoding as the wrong protocol), then such ACK would not be sent. Thus, we can utilise the lack of ACK as a decryption oracle (better illustrated in Figure 3.)

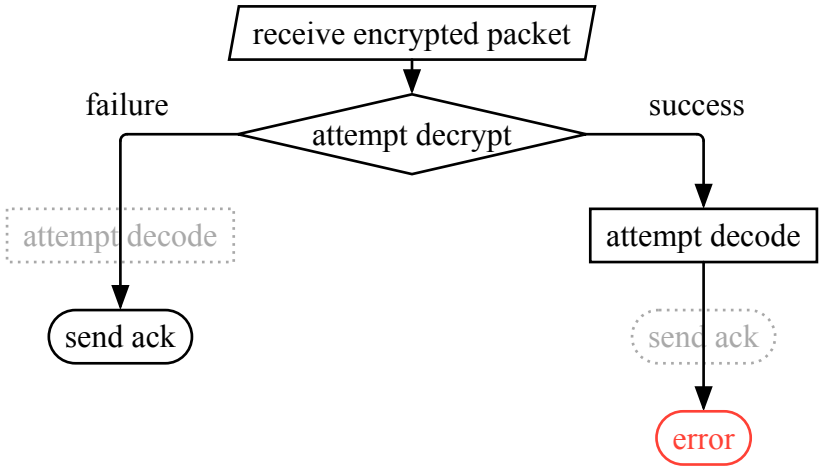

Figure 3: Illustration of the decryption oracle in our toy example.

Moreover, we also fulfill the concrete requirements:

1. The preshared passphrase is sampled from a non-uniform distribution (Figure 1).
2. The adversary has precomputed all ciphertexts that collide $\forall k \in \mathbb{K}$ where $|\mathbb{K}| > 32$.

4.1.2. Relevance

We model this toy example as a messaging app to mimic the most likely situation for this attack to crop up: when 2 parties are exchanging encrypted data over a public channel; as it enables each party to serve as some form of decryption oracle (as they would have to decrypt incoming packets). Unfortunately, we lacked sufficient time to attack a more concrete example (which will be elaborated on in Section 6.1).

5. Mitigation

To mitigate the partitioning oracle attack, any of the conditions we outlined above can be broken:

1. Use a key-committing scheme. This can be done by using a collision-resistant MAC (e.g. HMAC), or by performing the CTX transformation outlined in Chan & Rogaway (2022) on an existing AE scheme.

2. Rectify the decryption oracle. As the exact details of the oracle is necessarily implementation dependent, exactly how to do so again depends on the implementation.

3. Utilise a large & uniform keyspace. This is most easily done through using uniformly generated keys as opposed to some pre-shared passphrase.

6. Conclusion

All in all, we have illustrated the process and conditions required for the partitioning oracle attack, alongside showing algorithmically the speed-up against brute force. Moreover, we have created a demonstratory example to further exemplify these points.

6.1. Future Work

As mentioned earlier, we originally intended to apply the attack to a "real-world" pre-existing example, but were unable to due to a lack of time. As such, for possible future work, we may look into attacking *network protocols* that appear to be vulnerable (e.g. Kerberos, TLS).

Bibliography

Chan, John, & Rogaway, P. (2022). On Committing Authenticated-Encryption. *European Symposium on Research in Computer Security*.

Len, J., Grubbs, P., & Ristenpartmas, T. (2021). Partitioning Oracle Attacks. *30th USENIX Security Symposium (USENIX Security 21)*, 195–212.

Menda, S., Len, J., Grubbs, P., & Ristenpart, T. (2023). *Context Discovery and Commitment Attacks: How to Break CCM, EAX, SIV, and More*. https://eprint.iacr.org/2023/526

Wang, D., Jian, G., Huang, X., & Wang, P. (2014). *Zipf's Law in Passwords*. https://doi.org/10.1109/TIFS.2017.2721359