Key Reinstallation Attacks

Breaking WPA2 by forcing nonce reuse

Presentation Materials by Alihusein Kuwajerwala and Jacob Atzori

Vulnerability discovered by Mathy Vanhoef *of imec-DistriNet, KU Leuven*

The links are optional, since we tried to keep the write-up simple, they are meant for when you want more context or if you are interested in a topic. So don't feel like you have to read all of them.

Explaining the attack...

In a sentence:

A man in the middle attack that can be used to read information being sent between a device and access point that was previously assumed to be safely encrypted.

In a paragraph:

So when a device(eg: your phone, laptop) connects to an access point(eg: a router), it establishes a secure connection using a 4 way handshake. This 4 way handshake is used to perform a zero knowledge proof that both you, and the router know the Wi-Fi password. Your messages are then encrypted using a stream cipher. The WPA2 protocol says that if the handshake is interrupted, the interrupted messages should be resent, and in particular, if the device gets message 3 again, it should re-install its encryption key, which causes the vulnerability. The attacker jams the signal, causing an interruption, which forces the device to reinstall the key. The device then sends its following messages with this reused key, and this leaks information. Now the attacker can exploit this to read the messages encrypted with the same key.

In a special case, the attacker can force the device to install an all zero encryption key, making it trivial to read all encrypted communication between the device and the access point.

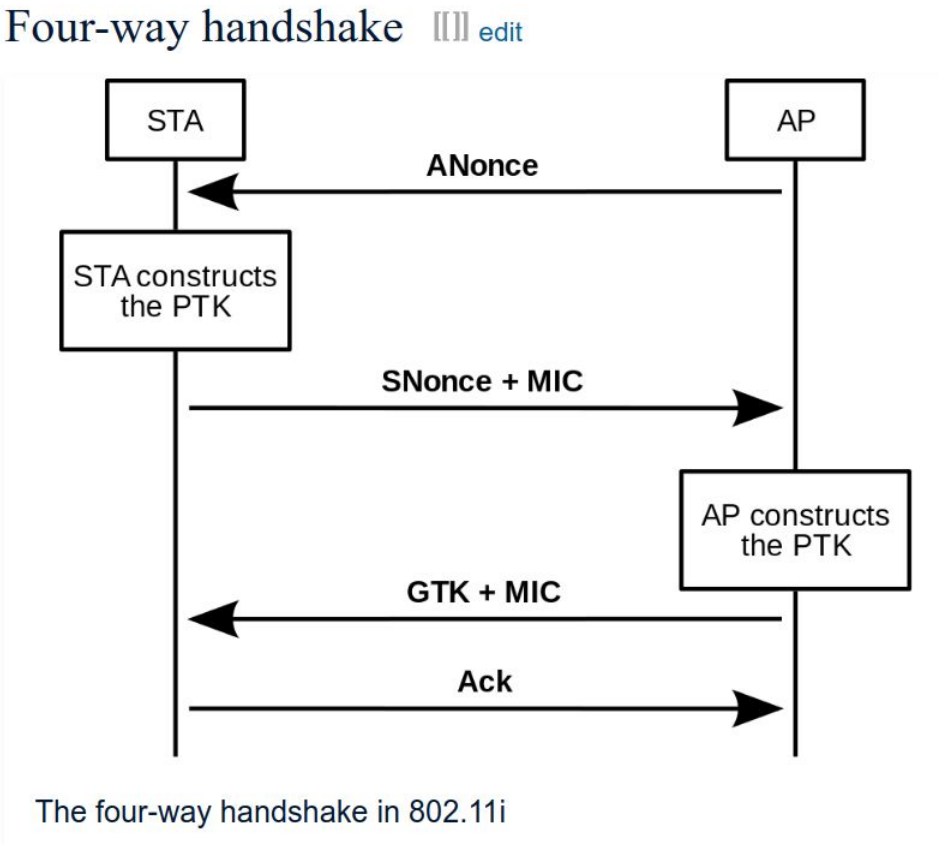
In detail:

First of all, see the attack in action. Next, if small text and acronyms is your thing, read the paper here.

So first a little bit from the paper about the WPA2 Protocol:

"The idea behind our attacks is rather trivial in hindsight, and can be summarized as follows. When a client joins a network, it executes the 4-way handshake to negotiate a fresh session key. It will install this key after receiving message 3 of the handshake. Once the key is installed, it will be used to encrypt normal data frames using a data-confidentiality protocol. However, because messages may be lost or dropped, the Access Point (AP) will retransmit message 3 if it did not receive an appropriate response as acknowledgment. As a result, the client may receive message 3 multiple times. Each time it receives this message, it will reinstall the same session key, and thereby reset the incremental transmit packet number (nonce) and receive replay counter used by the data-confidentiality protocol."

Here is a simple diagram of the 4 way handshake and a description of the messages:



The four-way handshake in 802.11i

Before the handshake, the client requests a connection and the AP authorizes it.

Then the handshake begins: (Note: These messages contain more things than what is mentioned here, I have left these out to keep things simple because, those details are not relevant to KRACK attacks)

The AP and Client have the PSK(WiFi Password), they compute their own nonces.

The AP computes the Anonce, and the Client computes the Snonce.

Message 1: The AP sends the Anonce to the client

Now the client can has everything it needs(Anonce, Snonce, PSK) to compute the PTK. AES is used(in block cipher mode) for encryption, and computing the PTK(PTK is described in detail below). PTK has everything used for encryption.

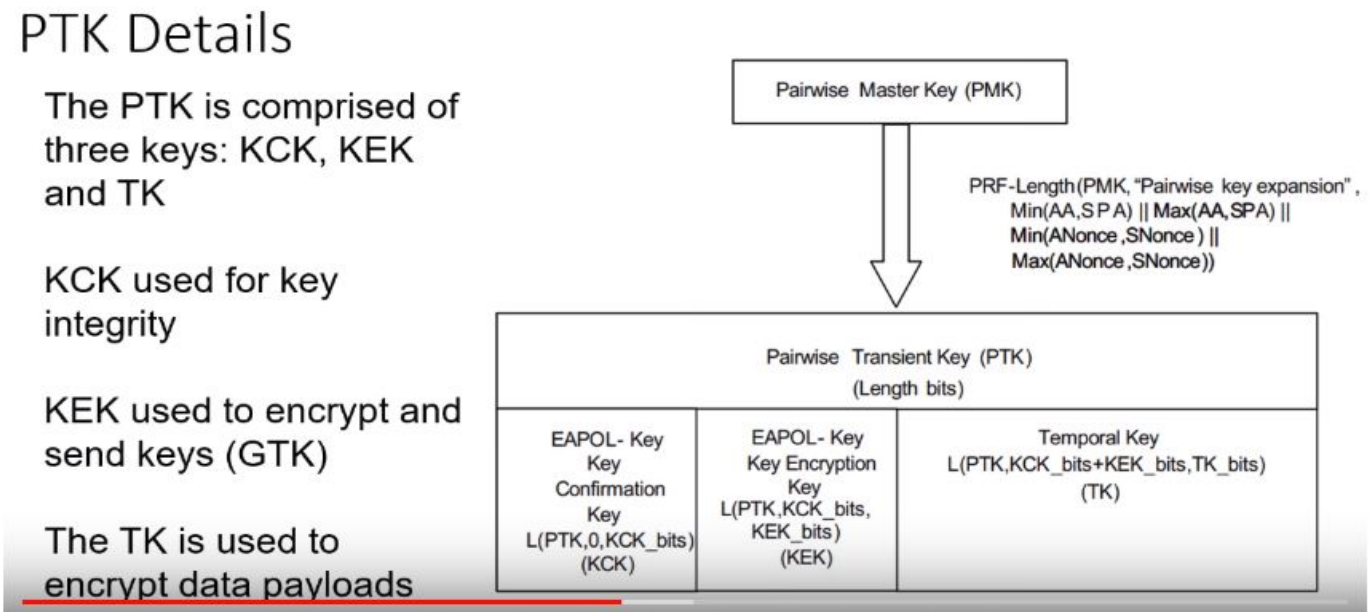
Message 2: The Client sends the Snonce to the AP

Now the AP can also compute the PTK. Next the AP and Client confirm that they both computed the same PTK by sending one more message. (if they didn't know the Wifi Password, they cannot do this)

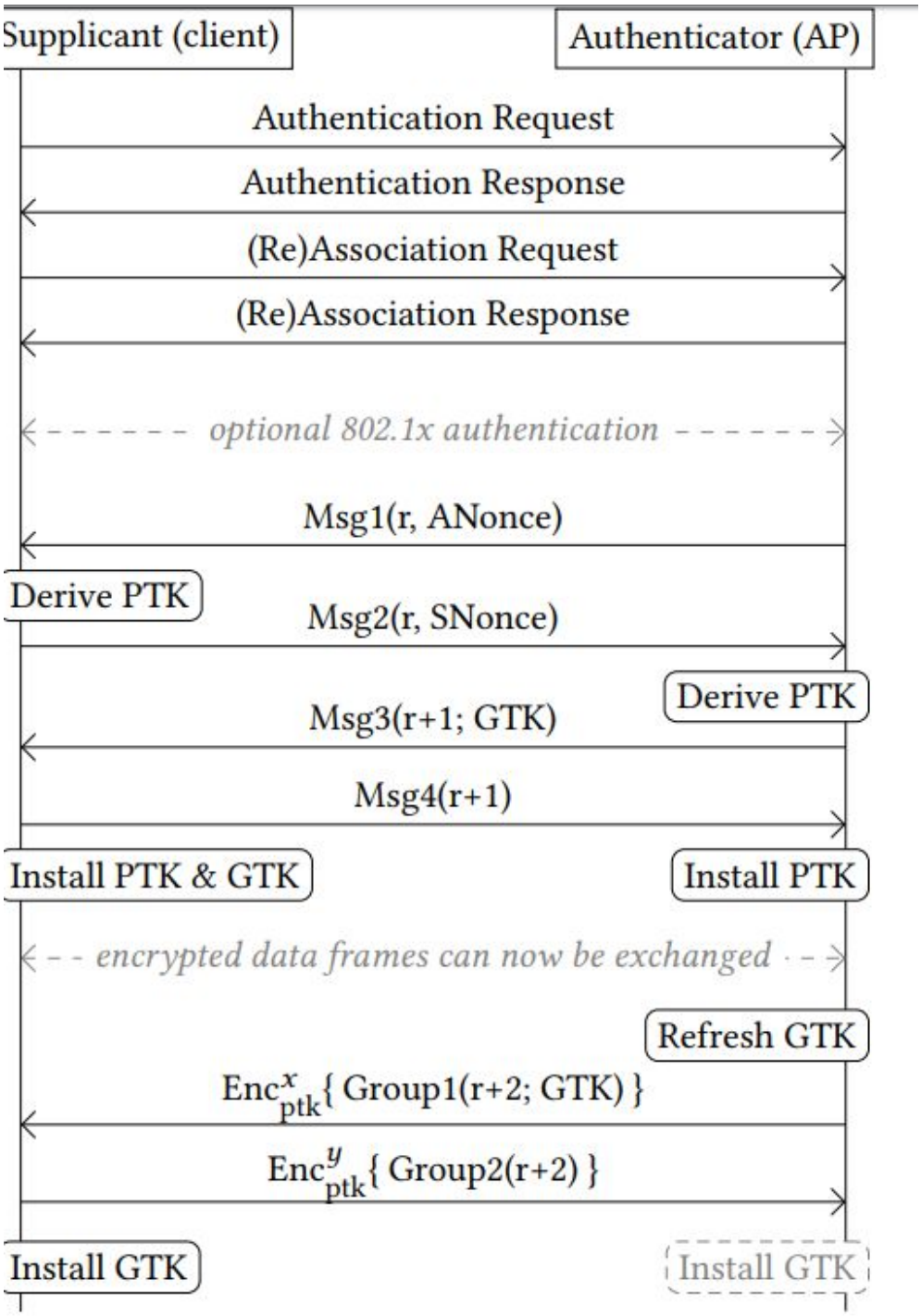
Message 3: The AP sends the Client the GTK, (used for group communication, dont worry about it)

Message 4: Client sends the AP confirmation that everything went well and the connection is secure.

Here are the details of the PTK:



Here is a diagram of the regular 4 way handshake with no attack:



Here is a diagram of the handshake with a KRACK attack:

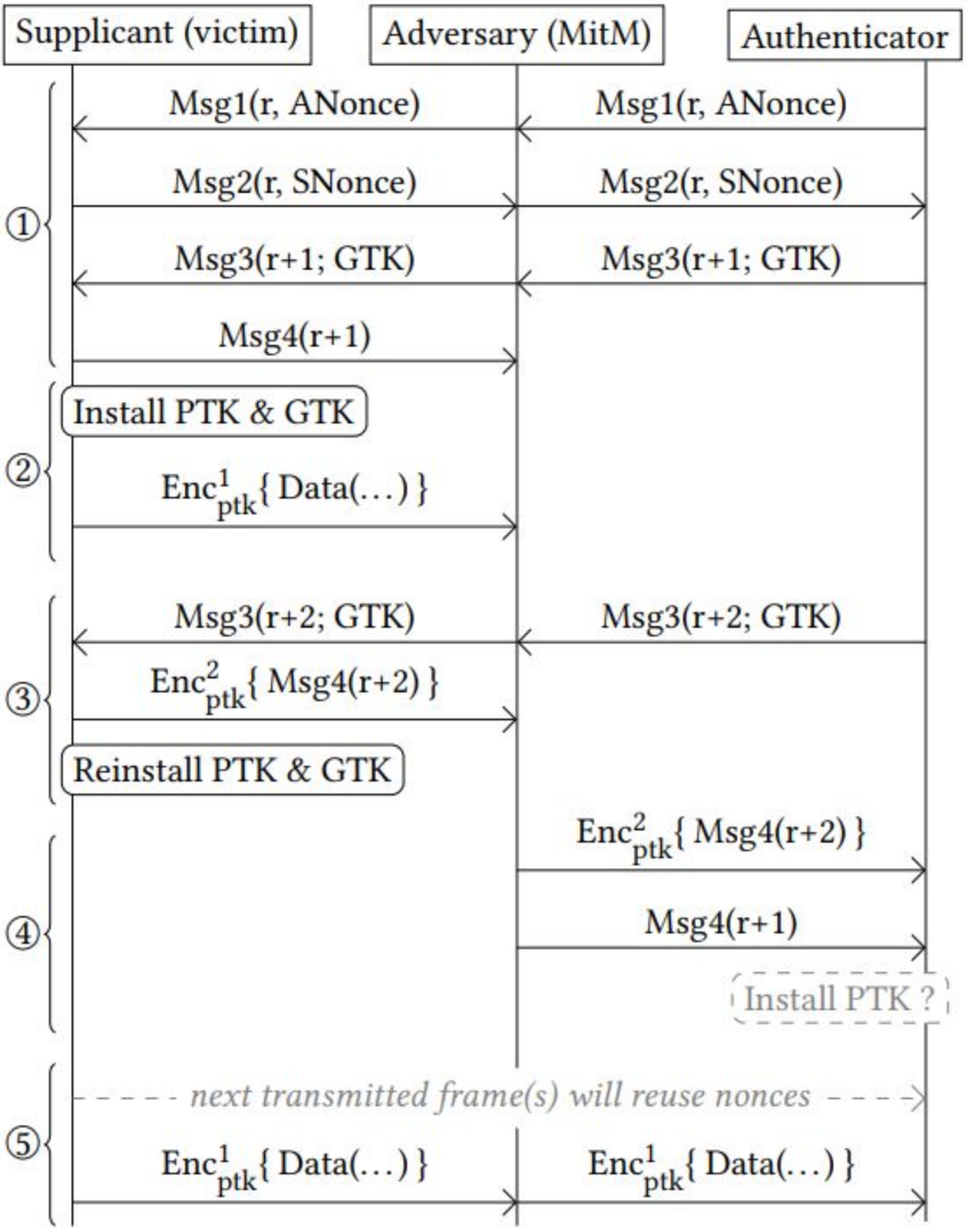


Figure 4: Key reinstallation attack against the 4-way handshake, when the supplicant (victim) still accepts plaintext retransmissions of message 3 if a PTK is installed.

So now the attack:

The MIM/Attacker jams the signal after the client receives message 3. Now the client's confirmation (message 4) does not get to the AP. The client assumes the handshake is done after sending message 4 and starts to send encrypted information over the network, but these messages are only seen by the attacker jamming the network and not the AP. The AP assumes the connection was interrupted and resends message 3. Now the client receives message 3 again, and following the WPA2 protocol, reinstalls its encryption key. It now continues it's communication using the same key. These messages can now be decrypted by the attacker.

About the all-zero encryption key:

This only affects versions 2.4 and 2.5 of wpa_supplicant, a Wi-Fi client commonly used on Linux.

Android 6.0 devices at the time used a modified version of wpa_supplicant and for security reasons, would overwrite the encryption key with zeros in their memory, which makes a lot of sense, because you don't want people to access it if they somehow steal the phone. However this totally backfired because when the same attack is used on these devices, the would re-install the key, which would be all zeros. This is really bad and allows all communications to be decrypted.

Mitigation:

The vulnerability is simple to patch. Just check if you're installing the same key again, and if that is the case, don't do it.

Before patches were out, a temporary fix was to just not accept message 3 more than once.

Important excerpts from the paper:

“Interestingly, our attacks do not violate the security properties proven in formal analysis of the 4-way and group key handshake. In particular, these proofs state that the negotiated session key remains private, and that the identity of both the client and Access Point (AP) is confirmed”

“Required functionality of both WPA and WPA2, and used by all protected Wi-Fi networks, is the 4-way handshake. Even enterprise networks rely on the 4-way handshake. Hence, all protected Wi-Fi networks are affected by our attacks.”

“In practice, some complications arise when executing the attack. First, not all Wi-Fi clients properly implement the state machine. In particular, Windows and iOS do not accept retransmissions of message 3 (see Table 1 column 2). This violates the 802.11 standard. As a result, these implementations are not vulnerable to our key reinstallation attack against the 4-way handshake. Unfortunately, from a defenders perspective, both iOS and Windows are still vulnerable to our attack against the group key handshake (see Section 4). Additionally, because both OSes support 802.11r, it is still possible to indirectly attack them by performing a key reinstallation attack against the AP during an FT handshake”

“Because Android internally uses a slightly modified version of wpa_supplicant, it is also affected by these attacks. In particular, we inspected the official source code repository of Android’s wpa_supplicant, and found that all Android 6.0 releases contain the all-zero encryption key vulnerability. Though third party manufacturers might use a different wpa_supplicant version in their Android builds, this is a strong indication that most Android 6.0 releases are vulnerable. In other words, 31.2% of Android smartphones are likely vulnerable to the all-zero encryption key vulnerability. Finally, we also empirically confirmed that Chromium is vulnerable to the all-zero encryption key vulnerability.”

Interesting:

This vulnerability is unique in the sense that the vulnerability does not lie in a specific implementation, but in the protocol we use for secure communication.

Ironically enough, no one noticed the protocol vulnerability until looking at its implementation in code.

Do formal security proofs for protocols cause us to stop paying attention and become lazy when constructing our security features? RSA is protected by such a proof for the amount of time it takes to do integer factorization. Quantum computer algorithms may affect this as well.

We need to work on our protocols, they need to be more specific. Hopefully WPA3 will address these issues.

How to decrypt the messages:

XOR:

`0 ^ 0 = 0`
`1 ^ 0 = 1`
`0 ^ 1 = 1`
`1 ^ 1 = 0`

Some properties of XOR:

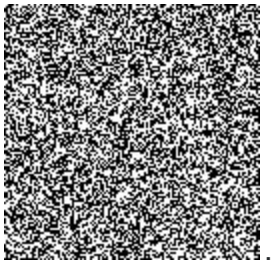
`A ^ B = C`
`C ^ B = A`
`C ^ A = B`
`A ^ 0 = A`
`A ^ A = 0`

The following was taken from here. It is too good to leave out. If this stuff interests you, this paper should provide you with more than enough detail.

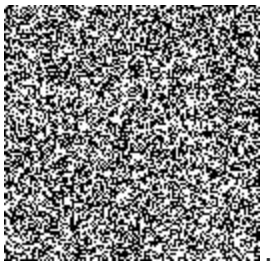
“There is a great graphical representation (which I found on cryptosmith, but they keep changing their url structures, so I've added the graphics in here) of the possible problems that arise from reusing a one-time pad.

Let's say you have the image

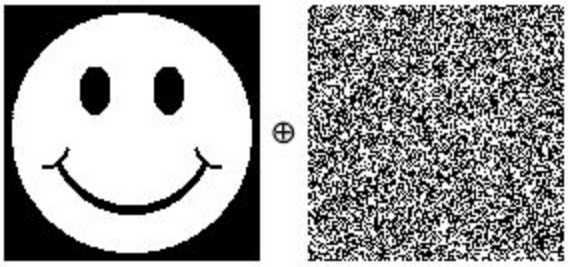
and you encrypt it by using the binary one-time-pad (xor-ing on black and white)



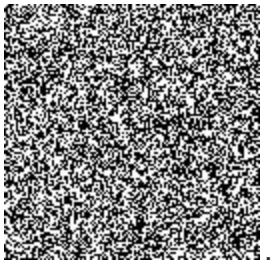
You get the following extremely secure encryption



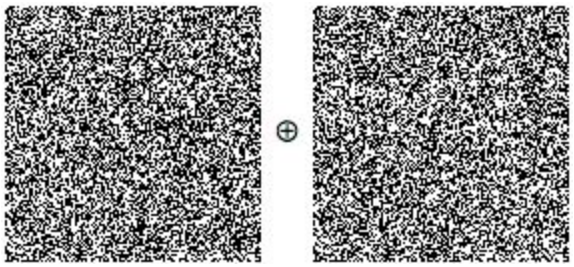
If you then encrypt a smiley face with the same one-time-pad,



you get another secure encryption



But if you have *both* and you xor them together



then you get the image

which, as you can qualitatively and intuitively see is very insecure.

Reusing the same key multiple times is called giving the encryption 'depth' - and it is intuitive that the more depth given, the more likely it is that information about the plaintext is contained within the encrypted text.

The process of 'peeling away' layered texts has been studied, as ir01 mentions, and those methods improve with more layers."

How WPA2 encrypts:
A key stream (KS) is created using AES by picking a starting value (IV) and encrypting with a private key, IV, then IV+1, then IV+2, ...
These values are placed one after another to create an unending stream of sudo-random numbers.
KS = e(IV) , e(IV+1) , e(IV+2) , e(IV+3) , ...
A cypher text is then created by XORing the KS with a message m.
c = KS ^ m

The Attack:

Given 2 cypher texts encrypted using the same KS
c1 = KS ^ m1
c2 = KS ^ m2

We XOR the 2 cypher texts.

```
c1 ^ c2   = (KS ^ m1) ^ (KS ^ m2)
          = KS ^ m1 ^ KS ^ m2
          = KS ^ KS ^ m1 ^ m2
          = 0 ^ m1 ^ m2
          = m1 ^ m2
```

Notice that the KS is no longer involved. So it doesn't matter what KS is used.

We then crib drag to find both m1 and m2.

This involves guessing a common "crib" (phrase) and dragging it along the message to find a "hit", a possible part of the message.

Once we have both we can solve for KS to decrypt any further messages
KS = m1 ^ c1

Example of crib dragging:

m1 = “super secret encrypted message”
m2 = “use cribdrag to decipher this.”

m1.hex =
737570657220736563726574206 5e6372797074656420 6d657373616765
m2.hex =
757365206372696264726167207 46f20646563697068657220746869732e

C1 =
d5e227f5ef7d9d9cfbaaa38bc7aa1dd44091bffcf7a43d478d8c8b49668c
C2 =
d3e432b0fe2f879bfcaaa798c7bb1c97568dace1e2a87858c88b904172c7

C1 ^ C2 = m1 ^ m2 = L
0606154511521a0707000413001101431 61c131d150c451f45071b08144b

Let’s try “decipher” as our first crib

“decipher”.hex = 6465636970686572
At position 0, we try XOR L ^ “decipher” and see the output
  0606154511521a0707000413001101431 61c131d150c451f45071b08144b
^ 6465636970686572

6263762c613a7f75 (“bcv,a:u” in asci)

This is clearly nonsense.
So we try position 1:

```
  0606154511521a07070004130011014316 1c131d150c451f45071b08144b
^  6465636970686572
------------------------------------------------------------------
040437386549c502 (“78eI?” in asci)
```

This is also nonsense.

We keep dragging the crib until we find something meaningful.

```
  0606154511521a07070004130011014316 1c131d150c451f45071b08144b
^                                    6465636970686572
------------------------------------------------------------------
                                     727970746564206d
(“rypted m” in asci)
```

This makes sense! -rypted seems like the end of the word “encrypted”

So far we think our 2 messages might be:
_____decipher_____
_____rypted m_____

We try this again with the word “encrypted”.
We even know where to check.

```
0606154511521a07070004130011014316 1c131d150c451f45071b08144b
^                          656e63727970746564
------------------------------------------------------------------
746f20646563697068
(“to deciph” in asci)
```

Then we get:
_____to decipher____
_____encrypted m____

We carry this one until both messages are decrypted.