CSE 127 Lecture 13: Symmetric Cryptography

Lecturer: Nadia Heninger Lecture #13

These lecture notes were scribed by students in CSE 127 during Winter 2021. They have been lightly edited but may still contain errors.

---

1 Cryptography Introduction

Cryptography is a very useful tool in computer security. However, it is not the solution to all problems in computer security. In addition, it does not have the same meaning as the word blockchain. It is important to note that a person should not attempt to invent cryptography unless they have sufficient academic knowledge to do so since it is very difficult to ensure security in cryptography. There are a number of primitives in cryptography, which we will treat as a black box and focus on their applications.

We can keep two examples of applications of cryptography in the real world in mind: handshake protocols for SSL/TLS, which is how browsers encrypt communications with web servers, and file encryption, which encrypts disk contents.

Consider a model in which Alice and Bob are the two communicating parties and Eve is an attacker who might be passive or actice. Three security properties that we may want to guarantee using cryptography are authenticity, secrecy, and integrity. Authenticity implies that Eve cannot impersonate Alice or Bob to the other. Secrecy implies that no one other than Alice and Bob can read the messages that are encrypted to them. Integrity implies that Eve cannot modify messages between Alice and Bob without being detected. Eve can be either a passive attacker or an active attacker.

2 Symmetric-Key Encryption

For encryption, a key is used to encrypt a plaintext input and to generate a ciphertext output. In a symmetric decryption algorithm, the same key is used to decrypt a ciphertext input and recover the plaintext output. In order for correctness to be guaranteed, decrypting the encrypted message must output the original message. The definition of security we will use, defined informally, is that the ciphertext shouldn't reveal any information about the plaintext.

An example of symmetric-key encryption is the one-time pad encryption scheme, invented by Vernam in 1917. In a one-time pad, the key is a uniformly random sequence of bits the same length as the message to be encrypted. To encrypt, the plaintext is XORed with the key bit by bit to generate the ciphertext. Correctness holds because decrypting an encrypted ciphertext is equivalent to XORing with the key twice, which just gives the message back. Shannon proved in 1949 that the one-time pad is "information-theoretically" secure as the ciphertext reveals no information about the plain text.

However, even though the one-time pad is theoretically perfectly secure under this definition, it is not used in modern cryptography, and has drawbacks that make it difficult to

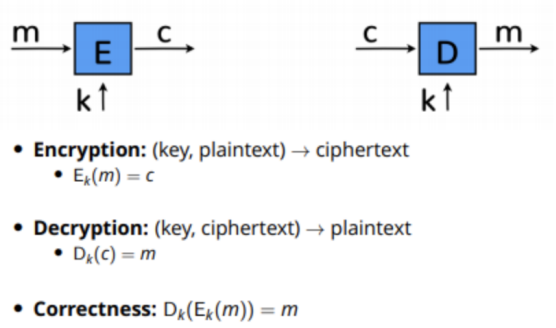

Figure 1: Symmetric-Key Encryption and Decryption Equations [**Heninger2021.**]

use in the real world. In particular, the key and the message must be of the same length, and the key must be generated perfectly uniformly at random. If two entities can securely exchange that much key material, they likely are able to communicate. Also, the key can only be used once, and security is totally compromised if the key is ever reused.

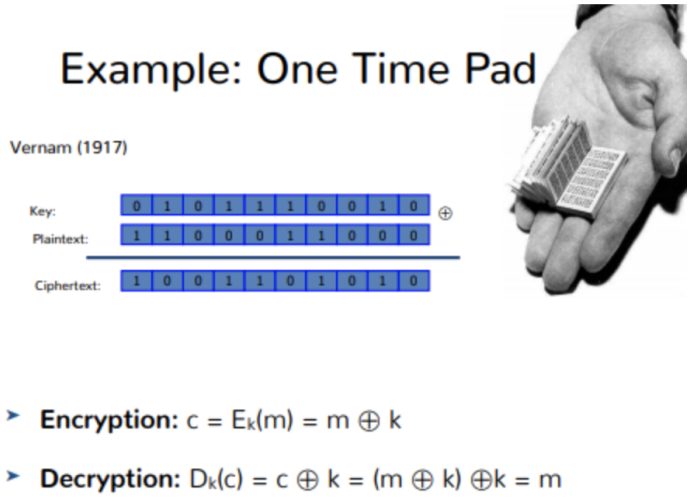

Figure 2: One Time Pad [**Heninger2021.**]

3 Stream Ciphers

In order to develop cryptography that is easier to use in the real world, we need to relax our security definition. We would like to exchange a short key that can be used to encrypt much longer messages. This isn't possible with information-theoretic security, so we will only ask for security against a computationally bounded adversary.

Intuitively, we would like to use the short key to generate a longer pseudorandom keystream that can be used to encrypt like a one-time pad. This is called a stream cipher, and the function that generates such a keystream is called a pseudorandom generator.

A pseudorandom generator inputs a seed and outputs an expanded stream of bits. The output should be pseudorandom in the sense that it is not possible to distinguish the output from a truly randomly generated sequence of bits in a computationally feasible amount of time. The pseudorandom generator can be used to implement a stream cipher. The basic idea of a stream cipher is that we apply the pseudorandom generator to the short key, to expand the key into a pseudorandom stream as long as the message. Then we XOR the expanded key with the message to get the ciphertext as the result, just like a one-time pad, except now with a pseudorandomly generated pad.

Two examples of stream ciphers used in the real world are ChaCha and Salsa.

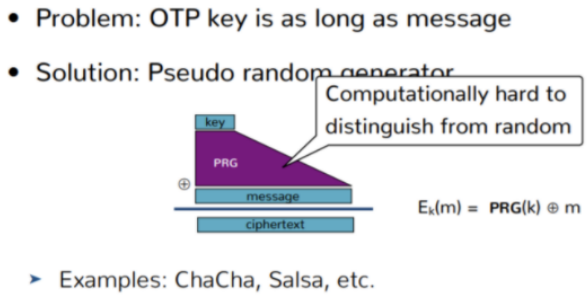

Figure 3: Use case for pseudorandom generators [**Heninger2021.**]

We still need to be careful about security with this construction.

What if we use the same key twice? Imagine that we use a key k twice to encrypt two different messages, M_1 and M_2, with the same key stream to obtain two ciphertexts $C_1 = M_1 \oplus PRG(k)$ and $C_2 = M_2 \oplus PRG(k)$. Then $C_1 \oplus C_2 = M_1 \oplus M_2$, that is, the key streams are canceled out. From $M_1 \oplus M_2$, what can we learn? The distribution of, say, English text or ASCII characters is very non-random, and an adversary can use frequency analysis or similar to derive likely sequences of plaintext that would result in the observed xored messages.

A historical example of this type of attack was the Venona Project, when the Soviets used a one-time pad key twice, which resulted in different one-time pad key books containing duplicate pages that have the same key, and they are captured by American intelligence to find out the plaintext messages.

A more modern example of this vulnerability is insecurities in the WEP WiFi protocol. The randomness used by WEP was insufficient, which meant that on a busy network an attacker could observe repeated keystreams after a few thousand packets, and thus use this information to break the password.

A chosen-plaintext attack is an active adversary model, where the attacker can learn the encryptions of messages of their choice under the unknown key used by the victim/target. A historical example of this type of attack comes from World War II when the US wanted to learn the location corresponding to the codeword "AF" used by the Japanese. The US Navy sent messages about Midway Island and observed that the encoded Japanese communications then referred to "AF".

WEP is also vulnerable to chosen plaintext attacks: an attacker may be able to inject traffic and cause it to be encrypted, which speeds up the attacks.

4 Block Ciphers

A block cipher is a permutation function that maps a fixed-size input block to a fixed-size output block. The particular permutation is chosen depending on the key. Since it is a permutation, the mapping also has a well-defined inverse: every input block corresponds to a unique output block, and this output block can be mapped via the inverse operation back to its corresponding input block.

Three common historically important examples of block ciphers include DES, 3DES, and AES. DES was the first block cipher to be publicly standardized by the US government in the late 1970s. Its block size was 64 bits and its key strength was 56 bits. This short key strength meant that it became computationally feasible for public researchers to decrypt DES traffic by the 1990s. Practitioners tried to fix this by using an iterated version of DES, called 3DES. 3DES's block size is 64 bits, and the key has 112 bits of security. The algorithm itself iterates DES encryption and decryption three times.

The US government held a competition in the late 1990s for a replacement for DES, and the algorithm that was selected was called AES. This algorithm remains a good choice of block cipher to use today, and no significant weaknesses have been found. AES has a 128-bit block size, and there are algorithm variants for 128, 192, and 256-bit keys.

Block cipher construction is a black art that we don't have time to go into in this class. However, to give a flavor of what these functions look like, AES consists of several iterated *rounds*. Each round mixes in a *round key* into the intermediate encryption state, permutes bits, xors bits, and substitutes bits. This is intended to be efficiently implementable in hardware and software, and yet still result in random-looking outputs.

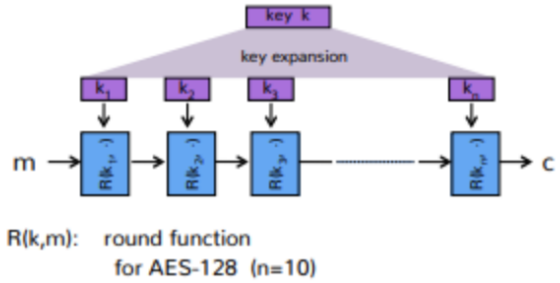

Figure 4: Visualization of AES-128 [**Heninger2021.**]

4.1 Block Cipher Modes of Operation

We would like to be able to use a block cipher to encrypt messages larger than a single block. The way this is accomplished is called a *mode of operation*. We present the following for historical reasons, but before we do that we will warn you that if you are using a cryptographic library and it asks you to choose a mode of operation for your block cipher, you should use a different cryptographic library that doesn't ask you to do this. The right way to use AES is with an AEAD mode that a cryptographer has already chosen for you.

On a small side note, you may wonder what happens if we try to use a block cipher to encrypt a message whose length is not an exact multiple of the length of a block. In this case, we pad the message to a multiple of a block to encrypt, and then strip off the padding after decryption to recover the original plaintext.

4.2 ECB

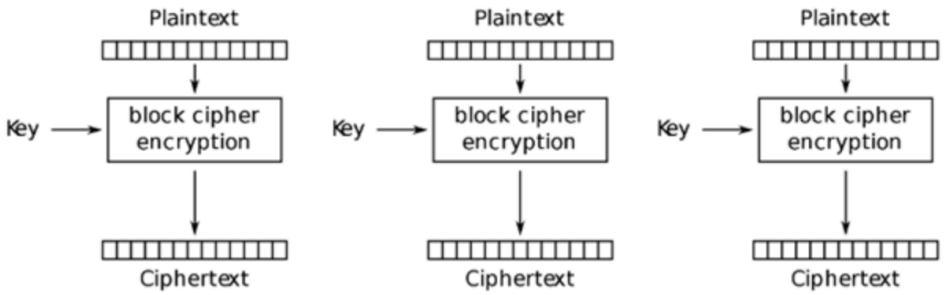

Figure 5: Electronic Codebook (ECB) mode encryption [**Heninger2021.**]

This section is intended to illustrate why cryptography can be hard and trap unsuspecting implementers.

The simplest way you might think to use a block cipher is to break up your (padded) plaintext into blocks, and then run each block through the block cipher to obtain a ciphertext block, which you concatenate together to obtain the full ciphertext output.

This is called "ECB" mode, Electronic CodeBook mode, and is terribly insecure. The reason that it is insecure is because the same input block will always result in the same output block. Thus even though individual 128-bit output blocks might be random, many 128-bit output blocks together, some of them repeating, will reveal information about the original input data.

The most famous illustration of this is the "ECB penguin". This is a bitmap of the Tux penguin that was encrypted using ECB mode. You can still see the penguin. Remember that ECB mode lets you see the penguin, so you should never use it.

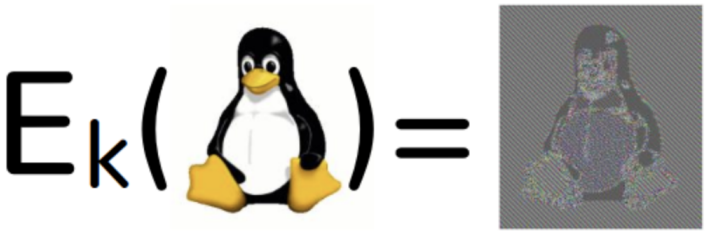

Figure 6: Linux "ECB penguin" [**Wikipedia**]

4.3 Cipher block chaining (CBC)

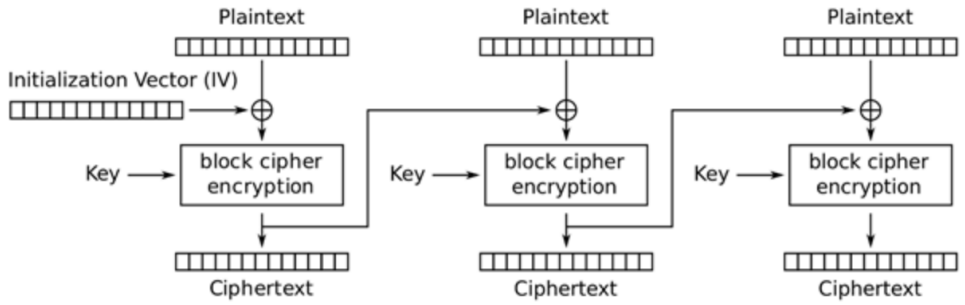

Figure 7: Cipher block chaining (CBC) mode encryption [**Heninger2021.**]

Another historically common block cipher mode of operation is CBC mode, or Cipher Block Chaining mode.

Cipher block chaining mode addresses the weakness of ECB mode by adding some randomization. There is a random value called the "Initialization Vector" (IV) which is used to randomize the inputs to the first encryption block, so that the outputs of the encryption should also be randomized. Then each block of ciphertext output is used to randomize the inputs to the next block. Informally, as long as two encryptions use different IVs, they should not result in the same ciphertexts, and thus the vulnerabilities we discussed above should not happen.

CBC mode is an improvement over EBC mode and is provably secure. Unfortunately, it can be difficult to implement properly in practice, and many CBC mode implementations and deployments in protocols have been found to be vulnerable to a class of attack called a padding oracle attack.[**JulianoRizzoThaiDuong.25.05.2005**] Thus CBC mode is no longer considered a good choice to use in practice.

4.4 Counter Mode Encryption (CTR)

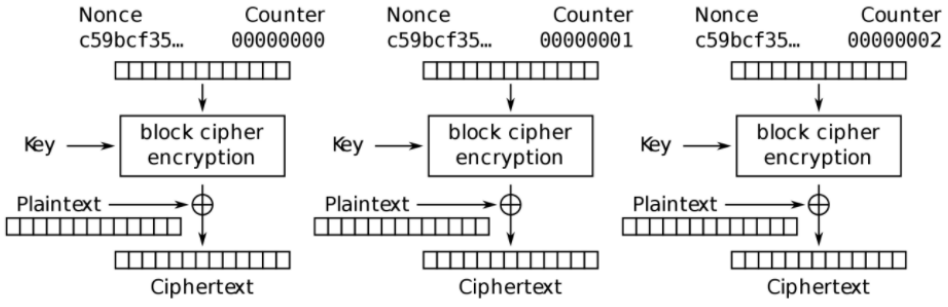

Figure 8: Counter Mode Encryption (CTR) mode encryption [**Heninger2021.**]

In CTR (counter) mode, the idea is to use a block cipher to make a stream of outputs that look like a stream cipher. To do this, you can pick a random starting point (this will be

our counter), and encrypt counter with the block cipher to obtain the first block, counter+1 to obtain the second block, counter+2 to obtain the third block, and so on. Then to encrypt a message with counter mode, you xor the message with this pseudorandom putput stream to obtain your ciphertext.

CTR is a fine mode, though again you should not be using a cryptographic library that asks you to choose a mode of operation.

If you are choosing a crypto library and it asks you to choose the block cipher mode of operation, please use a different library. There are several nuances/details to be aware of, and choosing the wrong one can be detrimental. Libsodium is a good example of a library with a user-friendly API. Please read towards the end of this lecture section for information about the block cipher mode of operation being built into an AEAD mode, which is preferable when choosing libraries.

We will also remind you here that brute force can be used to break any encryption algorithm. A brute force attack means simply running through all keys possible to decrypt the ciphertext until the correct plaintext is found. The complexity of the attack depends on the size of the keyspace. For example, brute forcing a 128 bit key needs 2^{128} attempts at decryption.

Encryption doesn't provide integrity against active attacks Returning to the example with Alice, Bob, and Eve, symmetric encryption on its own is not sufficient for security if Eve decides to switch to altering the ciphertexts between Alice and Bob. There need to be additional steps to protect against the modification of the messages/ciphertexts. To see why this is the case, return to the example of a one-time pad. If Eve observes a ciphertext $c = m \oplus k$ where Eve does not know the secret keystream k, observe that Eve can xor anything she wants into the ciphertext, $z \oplus c = z \oplus m \oplus k$. If Eve sends this message to Alice and Alice decrypts it with key k, the resulting message is $z \oplus m$ and Alice has no way to detect that this tampering has occurred.

We will take a brief detour into another cryptographic algorithm before solving the integrity problem.

5 Hash Functions

Another family of cryptographic algorithms is hash functions. Hash functions are cryptographic algorithms that map input of some arbitrary length, usually large, into a fixed-size output string. In the real world, hash function outputs might be 128 to 512 bits long. The outputs returned by a hash function are called hashes, hash values, or (historically) a message digest. [**tutorial**]

There are a few different security properties of hash functions that you might see reference to. These are: pre-image resistance, second pre-image resistance, and collision resistance. Of these, collision resistance is the most important. Collision resistance means that it is computationally hard to find two differing inputs, of any length, resulting in the same hash (Find m_1 and m_2 such that $H(m_1) = H(m_2)$). Pre-image resistance means that it is computationally hard to find the inverse of a hash function (Given a value h, find an input m such that

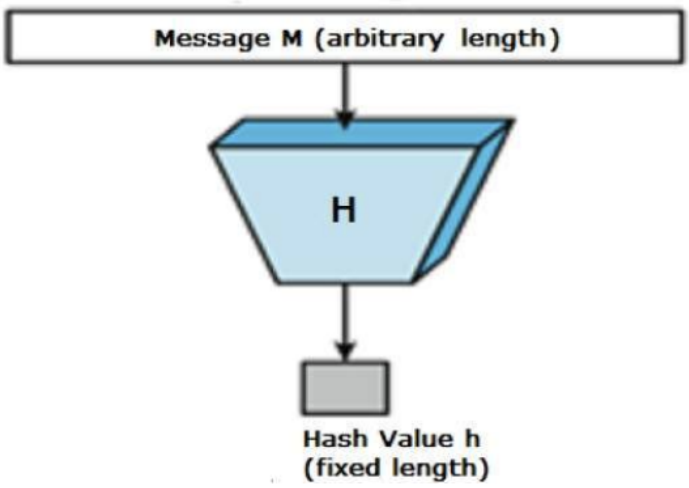

Figure 9: Visualization of a hash function

$H(m) = h$). Second pre-image resistance means that it is hard to find a different input with the same hash, given an input and its hash (Given m_1, find m_2 such that $H(m_1) = H(m_2)$).

Some applications of hash functions in real-world cryptography are for password storage [**tutorial**], versioning systems such as git, and data integrity checks. Data integrity checks provide the user with assurance that the data is correct. For example, if you want to be sure that the file you downloaded from a CDN is the file you intended to download, you can store the hash of it, and compare the hash of the file you downloaded to the hash you saved. This is implemented now in sub-resource integrity for web libraries.

Current popular hash functions include SHA-2 and SHA-3. Either of these is a fine choice for modern applications. SHA-2 was designed by NSA, and can be used to provide outputs of 224, 256, 384, or 512 bits. (These are often referred to as SHA-256 etc.) SHA-3 was chosen by NIST after running an open contest. It can be used to provide outputs of any size.

Examples of hash functions that were historically popular but are no longer secure are MD5 and SHA-1. MD5 results in an output of 128 bits, while SHA-1 results in an output of 160 bits. MD5 collisions are very fast to compute, and a SHA-1 collision was computed in 2017. These algorithms should no longer be used.

6 MACs

Sections 3 and 4 covered stream ciphers and block ciphers. The primary goal of these is to ensure secrecy so that no one else can read the messages. However, this does not prevent an attacker from changing the encrypted message during the transmission. Depending on the threat model an attacker might be able to alter the whole encrypted message or change parts of it. For example, if an attacker has access to the network infrastructure they could try to flip bits. This poses a serious security concern and violates the key concept of integrity.

We imagine a professor transmitting grades of their students encrypted to the adminis-

tration. The message contains the student's name and grade:

$$p_{ascii} \coloneqq \text{Eve's grade: C}$$

The plaintext can also be expressed in hexadecimal as $p_{hex} \coloneqq 457665277320677261646 53a2043$. Without ensuring the integrity of the encrypted message an attacker might be able to change the grade to an A (or whatever they prefer). To demonstrate this let $k_{hex} \coloneqq$ 616e2d27584a5a5e2f70494c3476 be a random key. By performing the XOR-operation we obtain the ciphertext:

$$c_{hex} \coloneqq p_{hex} \oplus k_{hex} = 241848002b6a3d2c4e142c761435$$

We assume that for all students the grades are transmitted using the same syntax as in above's plaintext and Eve somehow obtained this information. While Eve cannot reverse the ciphertext without the key she can manipulate the ciphertext as she knows at which position the grade is located:

$$c_{2_{hex}} \coloneqq 241848002b6a3d2c4e142c76143\mathbf{7}$$

Notice how the last char changes from 5 to 7. When reversing the ciphertext to plaintext Eve's grade changed from a C to an A:

$$p_{2_{hex}} \coloneqq c_{2_{hex}} \oplus k_{hex} = 457665277320677261646 53a2041$$

$$p_{2_{ascii}} \coloneqq \text{Eve's grade: A}$$

It becomes clear that symmetric encryption alone is not secure enough. Alice and Bob would like to authenticate the messages they exchange. Therefore, a message authentication code (MAC) should be used. A MAC is a tag that allows validating the original message based on a shared secret. Every party that is in possession of the shared secret can compute the MAC of a message and validate it. The validation tells a party if the message is authentic, that is if the message has not been altered during the transmission and that the validating party can verify the source of the message.

A MAC system is defined by a triple of algorithms (G, MAC, V) [**Katz.2008**]:

- G is a key-generator that outputs k (key)
- MAC is a keyed, tag generating function over its inputs k and m (message). It should be hard for an attacker to construct a valid pair (m, t).
- V is a function to verify its inputs k, m, and t with 0 (rejected) or 1 (accepted) as output

Such triple must satisfy the following correctness property, meaning that a correctly computed MAC tag on a message should verify with the same key:

$$V(k, m, MAC(k, m)) = 1$$

Going back to the previous example where Alice sends a message to Bob containing Eve's grade we can use a MAC to prevent Eve from tampering with the message. To do this, Alice

and Bob need to agree on a shared private key k. Then, Alice computes the MAC with key k and sends it with the original message to Bob. Finally, Bob validates the message and MAC tag using the key k. If V returns 1 the message is authentic and the content has not been changed. If Eve does not know key k, she should not be able to forge a valid MAC tag for an altered message m'. Please read section 6.2 for information about the actual construction of a secure MAC.

6.1 Length extension attack

It is tempting to try to implement a MAC function using a hash function.

Observe first that a hash function on its own is not a good choice of MAC: since everything about the hash function is known to Eve, she can easily compute $H(m')$ for any altered m' she chooses. Thus Alice and Bob actually need a keyed function.

The next simplest keyed MAC function you might try to construct from a hash function might look like $\text{MAC}(k, m) = H(k||m)$, where H is a hash function.[1]

Totally counterintuitively, this is completely insecure if H is MD5, SHA-1, or SHA-2. (It is fine if H is SHA-3. This is why cryptography is hard.) The reason is that MD5, SHA-1, or SHA-2 are all based on the Merkle–Damgård construction. The Merkle–Damgård construction is a provably secure way of building collision-resistant cryptographic hash functions by iterating a fixed-length compression function. Unfortunately, it is susceptible to length extension attacks, because any output can also serve as an intermediate state for a longer message. Thus, when the keyed function uses a vulnerable hash function like MD5 an attacker can append arbitrary data to the end of the message. The attacker would also compute a new MAC and send both, the modified message and the new MAC, to the destination: [**Heninger.**]

$$m_{modified} = m||padding||m'$$

$$MAC_{bad}(k, m_{modified}) = H(k||m||padding||m')$$

V will still accept the forged MAC as it is technically valid despite the fact that the message has been modified. The attacker does not have full control over the messages, since the Merkle-Damgård construction adds some padding, but being able to alter a message and construct a valid MAC tag invalidates the security requirements for a MAC and may be enough for a practical attack in some scenarios.

Figure 10 visualizes the process of a length extension attack. Also, the length extension attack is good example that constructing secure MACs is complex. You should not try to do it yourself as cryptography is hard to get right.

6.2 HMAC

There are two reasonable options to avoid attacks like length extension attacks. One is to use SHA-3. The other is to use SHA-2 with a construction called HMAC to generate a secure

[1]The MAC can be constructed differently. For example, $\text{MAC}(k, m) = H(k||m||k)$ is not vulnerable but also not widely used. [**Heninger.**]

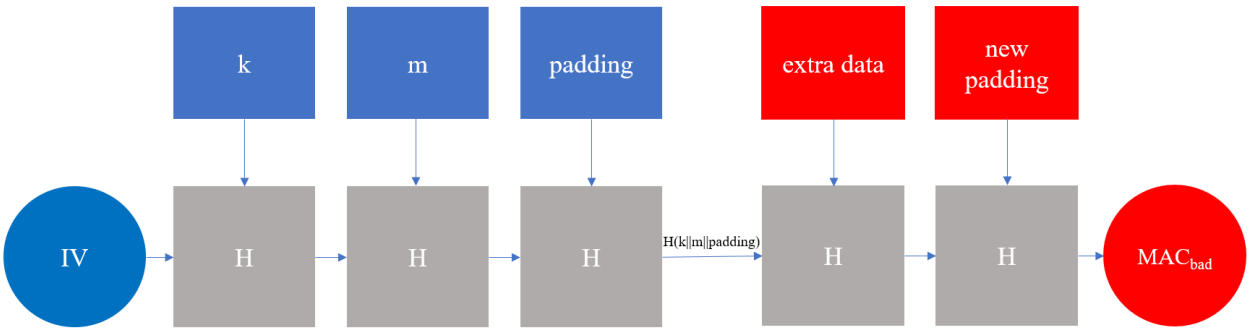

Figure 10: The length extension attack visualized. IV is the initialization vector and H a vulnerable hash function used to compute the MAC. After the first three computations, a non-tampered but insecure MAC has been constructed. As H is vulnerable the attacker uses the output from $H(k||m)$ to forge a bad MAC. They achieve this by appending their extra data, adding the correct padding according to the hash function construction, and applying the hash compression function.

MAC.[2].

An HMAC is a specific type of MAC that uses a hash algorithm and a key. To avoid length extension attacks, 6.1, an HMAC uses two passes of hash computation. An HMAC is constructed as follows:

$$MAC(k,m) = HMAC(k,m) = H((k' \oplus opad)||H((k' \oplus ipad)||m))$$

$$k' = \begin{cases} H(k) & \text{if k is longer than the block size of H} \\ k & \text{otherwise} \end{cases}$$

First, k' is determined based on the length of k. Then, the first pass hashes the derived key k' with the message. The second pass takes the hash and appends it to k' and produces a final hash, the HMAC. opad is the outer padding and ipad the inner padding. They consist of the byte 0x36 and 0x5C respectively repeated multiple times [**IETF.21.02.2021**]. Informally, the outer application of the hash function masks the intermediate result of the internal hash, so that HMACs are protected against length extension attacks.

6.3 Combining MAC with encryption to ensure better security

Encryption provides Bob with a sense of privacy: Bob knows Eve is not listening in to the messages he sends to Alice. Message Authentication Codes provide Bob with Integrity of a message and authentication of the sender, so Alice knows that the message is actually sent from Bob and it also hasn't been changed by Eve. But what Bob and Alice really strive for is a system of communication where they get all of these properties: Privacy, Integrity and Authentication. To do this, we need to use Encryption and MACs together. Unfortunately, it can be difficult to implement correctly.

[2]HMAC is described in RFC 2104. Mihir Bellare, a professor at UCSD, has notably contributed to this RFC.

There are a few possible ways to combine Message Authentication Codes and encryption in order to provide more robust security measures. An encryption scheme that provides both secrecy and integrity is called Authenticated Encryption. Unfortunately, it is not obvious which method of combining encryption and MACs actually provably gives authenticated encryption, and three historically prominent protocols, SSH, SSL, IPSec, all managed to make different choices.

We will tour these different choices as another example of why cryptography is hard to get right, in hopes of scaring you away from trying to do it yourself.

6.3.1 MAC-then-Encrypt

In MAC-then-Encrypt, an implementation would MAC the message and then encrypt the message and MAC together to obtain the ciphertext.

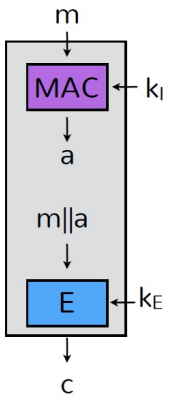

Figure 11: Visualization of MAC-then-Encrypt [**Heninger2021.**]

$$c = Enc_{k_E}(m||MAC_{k_I}(m))$$

SSL uses a MAC then Encrypt scheme that turns out to be secure for the choices that SSL makes, but this construction is *not* secure in general. Informally, this is because it only provides integrity for the plaintext, so an attacker may still be able to modify the ciphertext and convince Alice to accept a modified ciphertext.

6.3.2 Encrypt-and-MAC

In Encrypt-and-MAC, an implementation encrypts a message to obtain a ciphertext, and MACs the message to obtain a MAC, and appends these two together.

$$c||a = Enc_{k_E}(m)||MAC_{k_I}(m)$$

This is used in the SSH protcol. It is secure as SSH does it, but not provably secure in general. Informally, this is for two reasons: first, this scheme has the same problem as MAC-then-encrypt that only the plaintext is MACed, so an attacker may be able to modify the

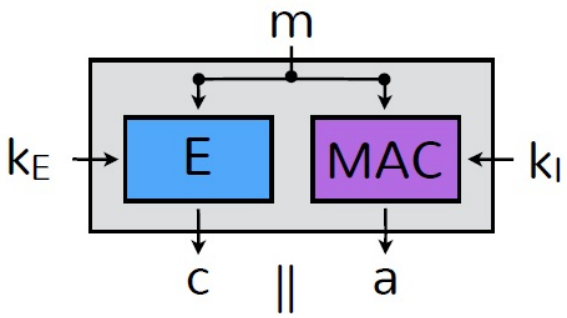

Figure 12: Visualization of Encrypt-and-MAC [**Heninger2021.**]

ciphertext without detection. Second, there is nothing in the security definition of a MAC that requires the message to be secret, so a secure MAC could reveal information about the message. (Even though the constructions used in practice probably don't.)

6.3.3 Encrypt-then-MAC

In Encrypt-then-MAC, an implementation encrypts a message m to produce ciphertext c, then MACs c to produce an authentication code a. The ciphertext is $c||a$.

This method is provably secure if used with a secure encryption scheme and a secure MAC scheme. Informally, this is because the ciphertext is MACed, which means that it cannot be modified by an adversary in transit. This turns out to be the security property we want.

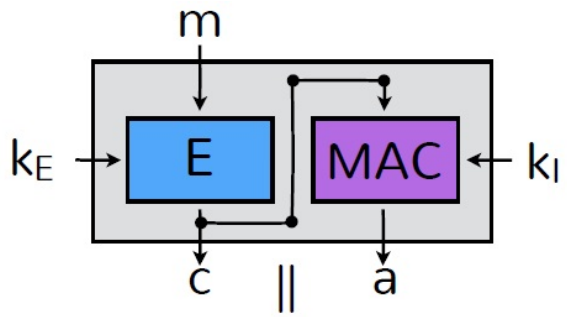

Figure 13: Visualization of Encrypt-then-MAC [**Heninger2021.**]

$$c||a = Enc_{k_E}(m)||MAC_{k_I}(Enc_{k_E}(m))$$

This is what the IPSec protocol does.

However, being provably secure doesn't mean an implementation can't still mess up. As an example, WinZip used Encrypt-then-MAC with secure constructions, but combined them poorly with compression and managed to be insecure.[**Kohno.Tadayoshi**]

Additional information regarding the general security notions of the above methods.[**BMC**]

7 Authenticated Encryption with Associated Data

Having to decide which methods and functions you want to use in order to create the next new Authenticated Encryption scheme is not something that you should try to do unless you decide to do a PhD in cryptography with Mihir Bellare. Instead of attempting to implement these things on your own, you should find a library that does the right thing by default and use that.

The keyword to look for is Authenticated Encryption with Associated Data, which provides encryption and authentication. Additionally, the associated data is plaintext that is authenticated as well to ensure integrity. Two examples of authenticated encryption modes of operation are AES-GCM and AES-GCM-SIV and it is recommended that you always use an authenticated encryption mode as it combines the mode of operation with integrity protection/MAC in a correct manner.