Cryptographic Hashes and MACs

CS 161 Fall 2022 - Lecture 7

Last Time: Block Ciphers

- Encryption: input a k-bit key and n-bit plaintext, receive n-bit ciphertext
- Decryption: input a k-bit key and n-bit ciphertext, receive n-bit plaintext
- Correctness: when the key is fixed, $E_K(M)$ should be bijective
- Security
  - Without the key, E_K(m) is computationally indistinguishable from a random permutation
  - Brute-force attacks take astronomically long and are not possible
- Efficiency: algorithms use XORs and bit-shifting (very fast)
- Implementation: AES is the modern standard
- Issues
  - Not IND-CPA secure because they're deterministic
  - Can only encrypt n-bit messages

Last Time: Block Cipher Modes of Operation

- ECB mode: Deterministic, so not IND-CPA secure
- CBC mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption is not parallelizable
  - Decryption is parallelizable
  - Must pad plaintext to a multiple of the block size
  - IV reuse leads to leaking the existence of identical blocks at the start of the message
- CTR mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption and decryption are parallelizable
  - Plaintext does not need to be padded
  - Nonce reuse leads to losing all security
- Lack of integrity and authenticity

Lack of Integrity and Authenticity

- Block ciphers are designed for *confidentiality* (IND-CPA)
- If an attacker tampers with the ciphertext, we are not guaranteed to detect it
- Remember Mallory: An *active* manipulator who wants to tamper with the message

Lack of Integrity and Authenticity

- Consider CTR mode
- What if Mallory tampers with the ciphertext using XOR?

	P	a	y		M	a	l		$	1	0	0
M	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	0x31	0x30	0x30

$\oplus$

$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	0x69	0xd2	0x96

=

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	0x58	0xe2	0xa6

Lack of Integrity and Authenticity

- Suppose Mallory knows the message *M*
- How can Mallory change the *M* to say `Pay Mal $900`?

	P	a	y		M	a	l		$	**1**	0	0
M	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	**0x31**	0x30	0x30

⊕

$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	**0x69**	0xd2	0x96

=

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x58**	0xe2	0xa6

Lack of Integrity and Authenticity

$C_i = M_i \oplus Pad_i$	$\texttt{0x58} = \texttt{0x31} \oplus Pad_i$	Definition of CTR
$Pad_i = M_i \oplus C_i$	$Pad_i = \texttt{0x58} \oplus \texttt{0x31}$ $= \texttt{0x69}$	Solve for the *i*th byte of the pad
$C'_i = M'_i \oplus Pad_i$	$C'_i = \texttt{0x39} \oplus \texttt{0x69}$ $= \texttt{0x50}$	Compute the changed *i*th byte

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x58**	0xe2	0xa6
C'	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x50**	0xe2	0xa6

Lack of Integrity and Authenticity

- What happens when we decrypt *C'*?
  - The message looks like “Pay Mal $900” now!
  - Note: Mallory didn’t have to know the key; no integrity or authenticity for CTR mode!

C'	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x50**	0xe2	0xa6

$\oplus$

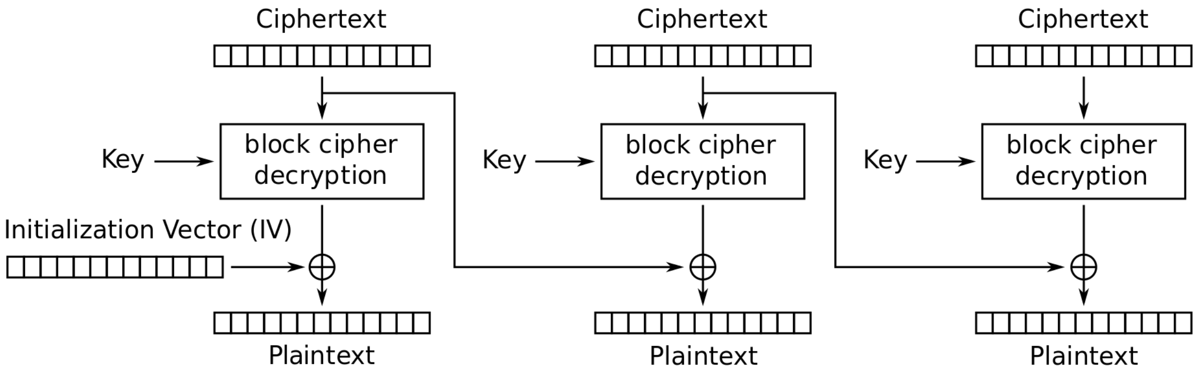
$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	**0x69**	0xd2	0x96

=

P'	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	**0x39**	0x30	0x30
	P	a	y		M	a	l		$	**9**	0	0

Lack of Integrity and Authenticity

- What about CBC?
  - Altering a bit of the ciphertext causes some blocks to become random gibberish
  - However, Bob cannot prove that Alice did not send random gibberish, so it still does *not* provide integrity or authenticity



Cipher Block Chaining (CBC) mode decryption

Today: Cryptography Hashes and MACs

- Hashing
  - Definition
  - Security: one-way, second preimage resistant, collision resistant
  - Examples
  - Length extension attacks
  - Application: Lowest-hash scheme
  - Do hashes provide integrity?
- MACs
  - Definition
  - Security: unforgeability
  - Example: HMAC
  - Do MACs provide integrity?
- Authenticated Encryption
  - Definition
  - Key Reuse
  - MAC-then-Encrypt or Encrypt-then-MAC?
  - AEAD Encryption Modes

Cryptographic Hashes

Textbook Chapter 7.1–7.3

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

Cryptographic Hash Function: Definition

- Hash function: $H(M)$
  - Input: *Arbitrary* length message M
  - Output: *Fixed* length, n-bit hash
  - Sometimes written as $\{0, 1\}^* \rightarrow \{0, 1\}^n$
- Properties
  - **Correctness**: Deterministic
    - Hashing the same input always produces the same output
  - **Efficiency**: Efficient to compute
  - **Security**: One-way-ness ("preimage resistance")
  - **Security**: Collision-resistance
  - **Security:** Random/unpredictability, no predictable patterns for how changing the input affects the output
    - Changing 1 bit in the input causes the output to be completely different
    - Also called "random oracle" assumption

Hash Function: Intuition

- A hash function provides a fixed-length “fingerprint” over a sequence of bits
- Example: Document comparison
    - If Alice and Bob both have a 1 GB document, they can both compute a hash over the document and (securely) communicate the hashes to each other
    - If the hashes are the same, the files must be the same, since they have the same “fingerprint”
    - If the hashes are different, the files must be different

Hash Function: One-way-ness or Preimage Resistance

- **Informal:** Given an output y, it is infeasible to find *any* input x such that $H(x) = y$
- **More formally:** For all polynomial time adversary,

  Pr[x chosen randomly from plaintext space; $y = H(x)$:

  Adv(y) outputs x' s.t. $H(x') = y$] is negligible
- Intuition: Here's an output. Can you find an input that hashes to this output?
  - Note: The adversary just needs to find *any* input, not necessarily the input that was actually used to generate the hash
- Example: Is H(x) = 1 one-way?
  - No, because given output 1, an attacker can return any number x

Hash Function: Collision Resistance

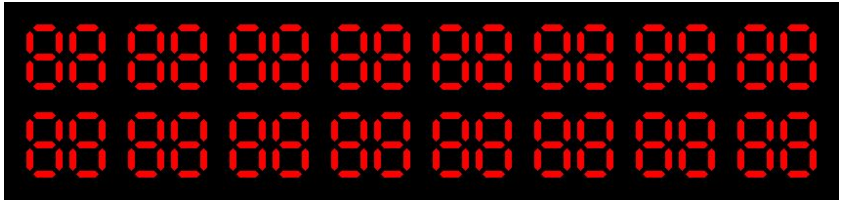
- **Collision**: Two different inputs with the same output
  - $x \neq x'$ and $H(x) = H(x')$
  - Can we design a hash function with no collisions?
    - No, because there are more inputs than outputs (pigeonhole principle)
  - However, we want to make finding collisions *infeasible* for an attacker
- **Collision resistance**: It is infeasible to (i.e. no polynomial time attacker can) find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$
- Intuition: Can you find *any* two inputs that collide with the same hash output for *any* output?

Hash Function: Collision Resistance

- **Birthday attack**: Finding a collision on an *n*-bit output requires only $2^{n/2}$ tries on average
  - This is why a group of 23 people are >50% likely to have at least one birthday in common

Hash Function: Examples

- MD5
  - Output: 128 bits
  - Security: Completely broken
- SHA-1
  - Output: 160 bits
  - Security: Completely broken in 2017
  - Was known to be weak before 2017, but still used sometimes
- SHA-2
  - Output: 256, 384, or 512 bits (sometimes labeled SHA-256, SHA-384, SHA-512)
  - Not currently broken, but some variants are vulnerable to a length extension attack
  - Current standard
- SHA-3 (Keccak)
  - Output: 256, 384, or 512 bits
  - Current standard (not meant to replace SHA-2, just a different construction)



A GIF that displays its own MD5 hash

Length Extension Attacks

- **Length extension attack**: Given $H(x)$ and the length of x, but not x, an attacker can create $H(x \mid\mid m)$ for any m of the attacker's choosing
  - See homework for a demo
  - Note: This doesn't violate any property of hash functions but is undesirable in some circumstances
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable

Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Mozilla publishes a new version of Firefox on some download servers
  - Alice downloads the program binary
  - How can she be sure that nobody tampered with the program?
- Idea: use cryptographic hashes
  - Mozilla hashes the program binary and publishes the hash on its website
  - Alice hashes the binary she downloaded and checks that it matches the hash on the website
  - If Alice downloaded a malicious program, the hash would not match (tampering detected!)
  - An attacker can't create a malicious program with the same hash (collision resistance)
- Threat model: We assume the attacker cannot modify the hash on the website
  - We have integrity, as long as we can communicate the hash securely

Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Alice and Bob want to communicate over an insecure channel
  - Mallory might tamper with messages
- Idea: Use cryptographic hashes
  - Alice sends her message with a cryptographic hash over the channel
  - Bob receives the message and computes a hash on the message
  - Bob checks that the hash he computed matches the hash sent by Alice
- Threat model: Mallory can modify the message *and the hash*
  - No integrity!

Do hashes provide integrity?

- It depends on your threat model
- If the attacker can modify the hash, hashes don't provide integrity
- Main issue: Hashes are *unkeyed* functions
    - There is no secret key being used as input, so any attacker can compute a hash on any value
- Next: Use hashes to design schemes that provide integrity

Message Authentication Codes (MACs)

Textbook Chapter 8.1–8.3 & 8.5–8.6

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

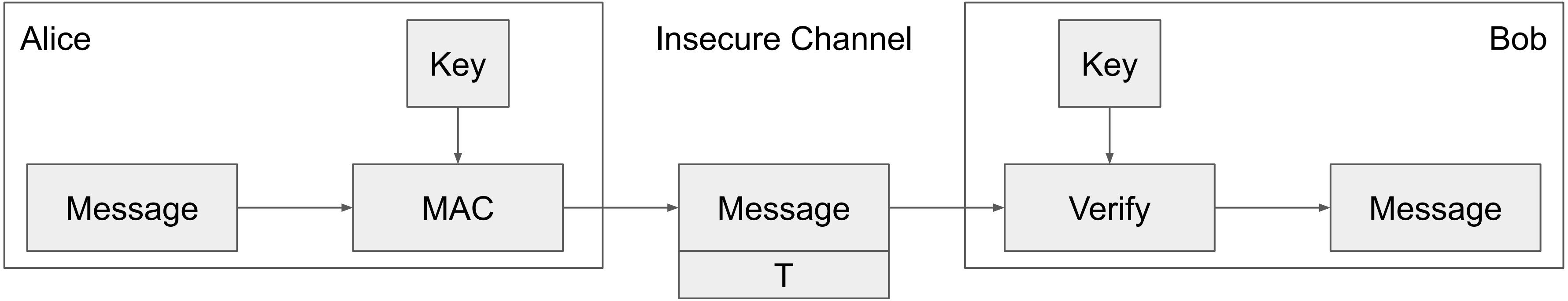
- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

How to Provide Integrity

- Reminder: We're still in the symmetric-key setting
  - Assume that Alice and Bob share a secret key, and attackers don't know the key
- We want to attach some piece of information to *prove* that someone with the key sent this message
  - This piece of information can only be generated by someone with the key

MACs: Usage

- Alice wants to send *M* to Bob, but doesn't want Mallory to tamper with it
- Alice sends *M* and *T* = MAC(*K*, *M*) to Bob
- Bob receives *M* and *T*
- Bob computes MAC(*K*, *M*) and checks that it matches *T*
- If the MACs match, Bob is confident the message has not been tampered with (integrity)



MACs: Definition

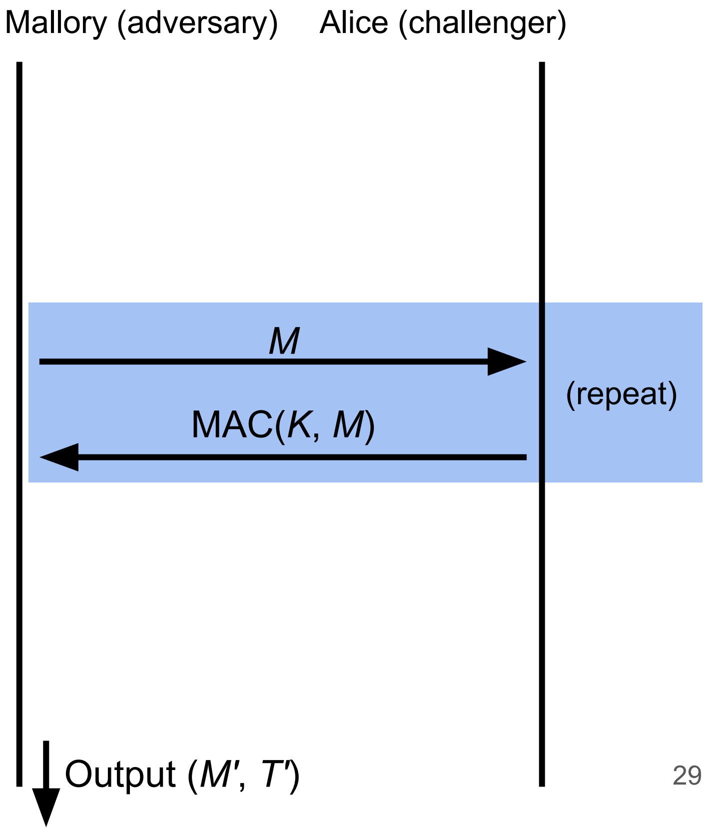
- Two parts:
  - KeyGen() → K: Generate a key K
  - MAC(K, M) → T: Generate a tag T for the message M using key K
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(K, M, T) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: EU-CPA (existentially unforgeable under chosen plaintext attack)

Defining Integrity: EU-CPA

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
  - Mallory cannot generate MAC(K, M') without K
  - Mallory cannot find any $M' \neq M$ such that MAC(K, M') = MAC(K, M)
- Formally defined by a security game: existential unforgeability under chosen-plaintext attack, or EU-CPA
- MACs should be unforgeable under chosen plaintext attack
  - Intuition: Like IND-CPA, but for integrity and authenticity
  - Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before

Defining Integrity: EU-CPA

1. Mallory may send messages to Alice and receive their tags
2. Eventually, Mallory creates a message-tag pair (M', T')
   - M' cannot be a message that Mallory requested earlier
   - If T' is a valid tag for M', then Mallory wins. Otherwise, she loses.
   - A scheme is EU-CPA secure if for *all* polynomial time adversaries, the probability of winning is 0 or negligible



Example: NMAC

- Can we use secure cryptographic hashes to build a secure MAC?
  - Intuition: Hash output is unpredictable and looks random, so let's hash the key and the message together
- KeyGen():
  - Output two random, n-bit keys K_1 and K_2, where n is the length of the hash output
- NMAC(K_1, K_2, M):
  - Output $H(K_1 \| H(K_2 \| M))$
- NMAC is EU-CPA secure if the two keys are different
  - Provably secure if the underlying hash function is secure
- Intuition: Using two hashes prevents a length extension attack
  - Otherwise, an attacker who sees a tag for M could generate a tag for $M \| M'$

Example: HMAC

- Issues with NMAC:
  - Recall: $\text{NMAC}(K_1, K_2, M) = H(K_1 \,\|\, H(K_2 \,\|\, M))$
  - We need two different keys
  - NMAC requires the keys to be the same length as the hash output (n bits)
  - Can we use NMAC to design a scheme that uses one key?
- $\text{HMAC}(K, M)$:
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0's to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it's n bits
  - Output $H((K' \oplus opad) \,\|\, H((K' \oplus ipad) \,\|\, M))$

Example: HMAC

- HMAC(K, M):
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0's to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it's n bits
  - Output $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$
- Use K' to derive two different keys
  - *opad* (outer pad) is the hard-coded byte `0x5c` repeated until it's the same length as K'
  - *ipad* (inner pad) is the hard-coded byte `0x36` repeated until it's the same length as K'
  - As long as *opad* and *ipad* are different, you'll get two different keys
  - For paranoia, the designers chose two very different bit patterns, even though they theoretically need only differ in one bit

HMAC Properties

- HMAC(K, M) = $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$
- HMAC is a hash function, so it has the properties of the underlying hash too
  - It is collision resistant
  - Given HMAC(K, M) and K, an attacker can't learn M
  - If the underlying hash is secure, HMAC doesn't reveal M, but it is still deterministic
- You can't verify a tag T if you don't have K
  - This means that an attacker can't brute-force the message M without knowing K

Do MACs provide integrity?

- Do MACs provide integrity?
    - Yes. An attacker cannot tamper with the message without being detected
- Do MACs provide authenticity?
    - It depends on your threat model
    - If a message has a valid MAC, you can be sure it came from *someone with the secret key*, but you can't narrow it down to one person
    - If only two people have the secret key, MACs provide authenticity: it has a valid MAC, and it's not from me, so it must be from the other person
- Do MACs provide confidentiality?
    - MACs are deterministic ⇒ No IND-CPA security
    - MACs in general have no confidentiality guarantees; they can leak information about the message
        - HMAC doesn't leak information about the message, but it's still deterministic, so it's not IND-CPA secure

Authenticated Encryption

Textbook Chapter 8.7 & 8.8

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

Authenticated Encryption: Definition

- **Authenticated encryption** (**AE**): A scheme that simultaneously guarantees confidentiality and integrity (and authenticity, depending on your threat model) on a message
- Two ways of achieving authenticated encryption:
    - Combine schemes that provide confidentiality with schemes that provide integrity
    - Use a scheme that is designed to provide confidentiality and integrity

Combining Schemes: Let's design it together

- You can use:
  - An IND-CPA encryption scheme (e.g. AES-CBC): Enc(K, M) and Dec(K, M)
  - An unforgeable MAC scheme (e.g. HMAC): MAC(K, M)
- First attempt: Alice sends Enc(K_1, M) and MAC(K_2, M)
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? No, the MAC is not IND-CPA secure
- Idea: Let's compute the MAC on the *ciphertext* instead of the plaintext: Enc(K_1, M) and MAC(k_2, Enc(K_1, M))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, the MAC might leak info about the ciphertext, but that's okay
- Idea: Let's encrypt the MAC too: Enc(K_1, M || MAC(K_2, M))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, everything is encrypted

MAC-then-Encrypt or Encrypt-then-MAC?

- MAC-then-encrypt
  - First compute $\text{MAC}(K_2, M)$
  - Then encrypt the message and the MAC together: $\text{Enc}(K_1, M \,||\, \text{MAC}(K_2, M))$
- Encrypt-then-MAC
  - First compute $\text{Enc}(K_1, M)$
  - Then MAC the ciphertext: $\text{MAC}(K_2, \text{Enc}(K_1, M))$
- Which is better?
  - In theory, both are IND-CPA and EU-CPA secure if applied properly
  - MAC-then-encrypt has a flaw: You don't know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it's more robust to mistakes

Key Reuse

- **Key reuse**: Using the same key in two different use cases
    - Note: Using the same key multiple times for the same use (e.g. computing HMACs on different messages in the same context with the same key) is not key reuse
- Reusing keys can cause the underlying algorithms to interfere with each other and affect security guarantees
    - Example: If you use a block-cipher-based MAC algorithm and a block cipher chaining mode, the underlying block ciphers may no longer be secure
    - Thinking about these attacks is hard

Key Reuse

- Simplest solution: Do not reuse keys! One key per *use*.
  - Encrypt a piece of data and MAC a piece of data?
    - Different use; different key
  - MAC one of Alice's messages to Bob and MAC one of Bob's messages to Alice?
    - Different use; different key
  - Encrypt one of Alice's files and encrypt another one of Alice's files?
    - It's *probably* fine to use the same key, but cryptographic design is tricky to get right!
  - Encrypt user metadata, encrypt file metadata, and encrypt file data?
    - You'll have to think about this in Project 2!

TLS 1.0 “Lucky 13” Attack

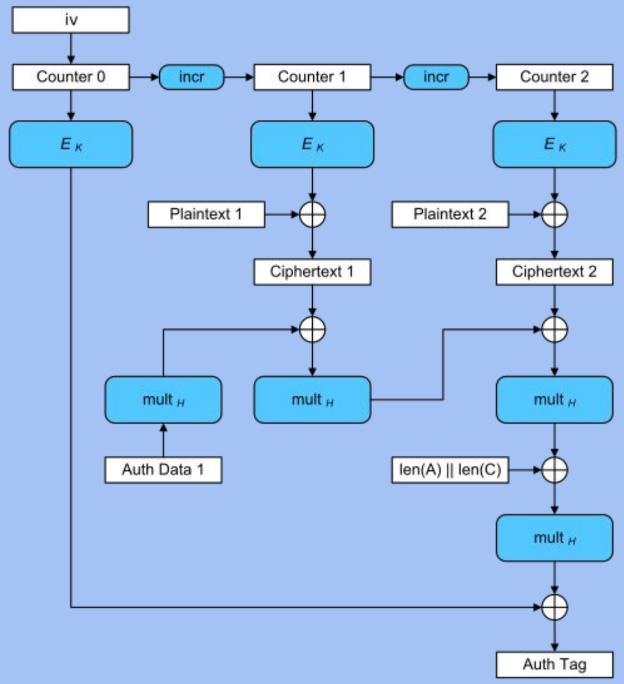
- TLS: A protocol for sending encrypted and authenticated messages over the Internet (we’ll study it more in the networking unit)
- TLS 1.0 uses MAC-then-encrypt: $\text{Enc}(K_1, M \,||\, \text{MAC}(K_2, M))$
  - The encryption algorithm is AES-CBC
- The Lucky 13 attack abuses MAC-then-encrypt to read encrypted messages
  - Guess a byte of plaintext and change the ciphertext accordingly
  - The MAC will error, but the time it takes to error is different depending on if the guess is correct
  - Attacker measures how long it takes to error in order to learn information about plaintext
  - TLS will send the message again if the MAC errors, so the attacker can guess repeatedly
- Takeaways
  - Side channel attack: The algorithm is proved secure, but poor implementation made it vulnerable
  - Always encrypt-then-MAC
  - You’ll try a similar attack in Homework 2!

AEAD Encryption

- Second method for authenticated encryption: Use a scheme that is designed to provide confidentiality, integrity, and authenticity
- **Authenticated encryption with additional data** (**AEAD**): An algorithm that provides both confidentiality and integrity over the plaintext and integrity over *additional data*
    - Additional data is usually context (e.g. memory address), so you can't change the context without breaking the MAC
- Great if used correctly: No more worrying about MAC-then-encrypt
    - If you use AEAD incorrectly, you lose *both* confidentiality and integrity/authentication
    - Example of correct usage: Using a crypto library with AEAD

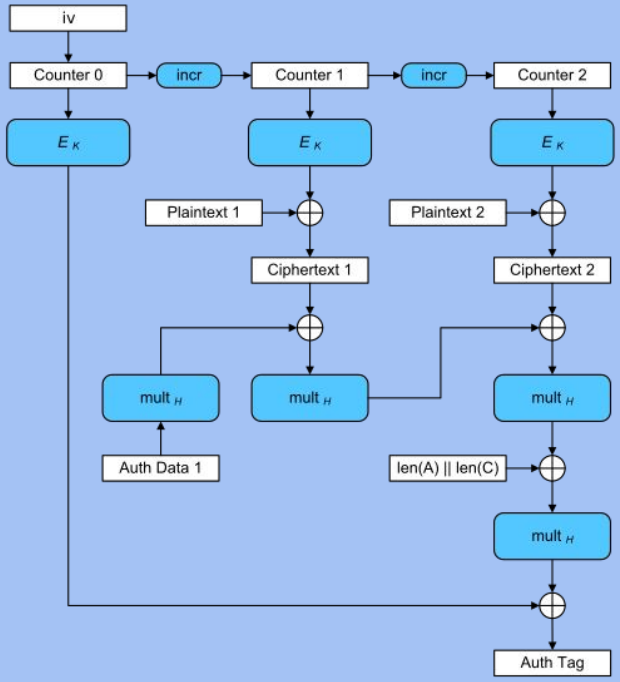
AEAD Example: Galois Counter Mode (GCM)

- **Galois Counter Mode** (**GCM**): An AEAD block cipher mode of operation
- E_K is standard block cipher encryption
- mult_H is 128-bit multiplication over a special field (Galois multiplication)
  - Don't worry about the math



AEAD Example: Galois Counter Mode (GCM)

- Very fast mode of operation
  - Fully parallel encryption
  - Galois multiplication isn't parallelizable, but it's very fast
- Drawbacks
  - IV reuse leads to loss of confidentiality, integrity, and authentication
  - This wouldn't happen if you used AES-CTR and HMAC-SHA256
  - Implementing Galois implementation is difficult and easy to screw up
- **Takeaway**: GCM provides integrity and confidentiality, but if you misuse it, it's even worse than CTR mode



Hashes: Summary

- Map arbitrary-length input to fixed-length output
- Output is deterministic and unpredictable
- Security properties
  - One way: Given an output y, it is infeasible to find any input x such that $H(x) = y$.
  - Second preimage resistant: Given an input x, it is infeasible to find another input $x' \neq x$ such that $H(x) = H(x')$.
  - Collision resistant: It is infeasible to find another any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.
- Some hashes are vulnerable to length extension attacks
- Application: Lowest hash scheme
- Hashes don’t provide integrity (unless you can publish the hash securely)

MACs: Summary

- Inputs: a secret key and a message
- Output: a tag on the message
- A secure MAC is unforgeable: Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before
  - Example: HMAC(K, M) = $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$
- MACs do not provide confidentiality

Authenticated Encryption: Summary

- Authenticated encryption: A scheme that simultaneously guarantees confidentiality and integrity (and authenticity) on a message
- First approach: Combine schemes that provide confidentiality with schemes that provide integrity and authenticity
  - MAC-then-encrypt: $\text{Enc}(K_1, M \,||\, \text{MAC}(K_2, M))$
  - Encrypt-then-MAC: $\text{Enc}(K_1, M) \,||\, \text{MAC}(K_2, \text{Enc}(K_1, M))$
  - Always use Encrypt-then-MAC because it's more robust to mistakes
- Second approach: Use AEAD encryption modes designed to provide confidentiality, integrity, and authenticity
  - Drawback: Incorrectly using AEAD modes leads to losing *both* confidentiality and integrity/authentication

Next Time

- Symmetric-key encryption schemes need randomness. How do we securely generate random numbers?
- When discussing symmetric-key schemes, we assumed Alice and Bob managed to share a secret key. How can Alice and Bob share a symmetric key over an insecure channel?