Chair for Network Architectures and Services
Institute of Informatics
TU München – Prof. Carle

Network Security

Chapter 2.3

Secure Channel and Authenticated Encryption

Acknowledgments

This course is based to a significant extend on slides provided by Günter Schäfer, author of the **book "Netzsicherheit - Algorithmische Grundlagen und Protokolle"**, available in German from **dpunkt Verlag**. The English version of the book is entitled “Security in Fixed and Wireless Networks: An Introduction to Securing Data Communications” and is published by Wiley is also available. We gratefully acknowledge his support.

The slides by Günter Schäfer have been partially reworked by Cornelius Diekmann, Heiko Niedermayer, Ali Fessi, Ralph Holz and Georg Carle.

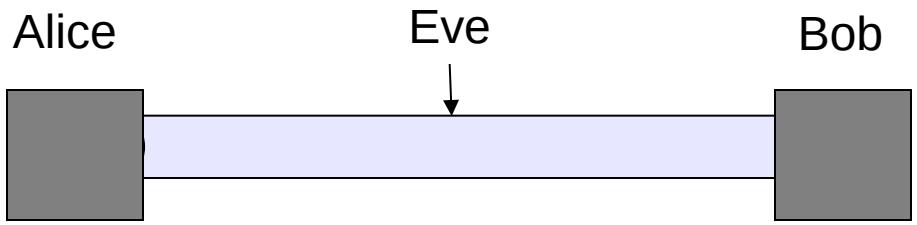
Overview

- **Part I: The Secure Channel**
- Part II: Attacks against Secure Channel
- Part III: Authenticated Encryption

Problem Statement (1)

(c.f. Niels Ferguson, Bruce Schneier: Practical Cryptography, Ch 8, pp. 111ff)

- Goal
  - This chapter illustrates the functionality of a “secure channel” between two parties Alice and Bob, provided by a simple security protocol, the so-called secure channel algorithm.
  - The functionality of this secure channel is a good start for understanding the functionality of common security protocols such as IPSec and SSL.



Problem Statement (2)

- Assumptions
  - The channel is bi-directional, i.e. Alice sends messages to Bob and Bob sends messages to Alice (almost all communications are bi-directional).
  - Eve tries to attack the secure channel in any possible way.
    - Eve can read all of the communication between Alice and Bob and arbitrarily manipulate exchanged messages.
    - Particularly, Eve can delete, insert, or modify exchanged messages.
- Requirement
  - Alice and Bob share a secret session key *K* that is known only to both of them.
  - The way how this key is established is discussed in the next Chapter.

What should Alice and Bob keep secret?

- Kerckhoff's Principle (short version): "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

- Now, is this a true fact?
  - No, it is a *guideline for good design*, but not a universal truth.
  - The assumption is that you gain more from making the system design public and publicly scrutinized than from hiding a system design where flaws at first may be unknown to attackers, but overlooked by designers.
  - *Kerckhoff's principle is widely accepted in cryptography.*

  → *So only the shared key is secret.*

- Does all security technology follow this principle?
  - Well, it is about cryptography.
  - But philosophically, does a …. obey it? Firewall? NAT? IDS?
    - Some technologies are more an arms race between defender and attacker.

Security Properties

- Processing
  - Alice needs to send a sequence of messages (Service Data Units: SDU) m_1, m_2, …
  - These messages are processed by the secure channel algorithm (i.e. the security protocol), which generates PDUs (Protocol Data Units) and sends them to Bob.
  - Bob processes the received PDUs using the corresponding secure channel algorithm and ends up with a sequence of messages m_1‘, m_2‘, …
  - In the ideal case {m_1‘, m_2‘, … } = {m_1, m_2, … }

Security Properties

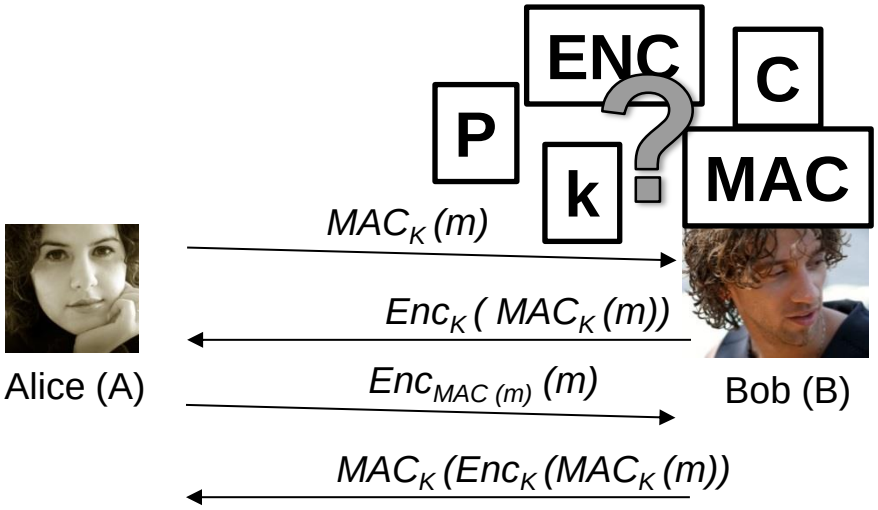
- Security properties of the secure channel algorithm
  - Eve does not learn anything about the messages m_i except for their timing and size.
  - $\{m_1', m_2', \ldots\} \leq \{m_1, m_2, \ldots\}$
  - Bob can not prevent Eve from deleting a message in transit

    (and Bob can not prevent message loss either).
  - The messages received are in the correct order.
  - There are no duplicate messages, no modified messages and no bogus messages sent by someone else other than Alice.
  - Bob knows exactly which messages he has missed.

General Remark

- Some protocols have some acknowledgement mechanisms for recovering from message loss.
- However, this is not handled by the secure channel, since it would make it more complicated.

MAC-then-Enc/Enc-then-MAC – design guidelines?



We know

- Symmetric ciphers
- Encryption modes
- MAC

How shall we combine them?

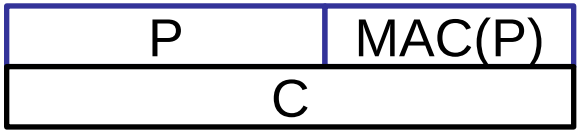
Horton Principle vs Encrypt-then-MAC

- Horton principle: “Authenticate what you mean, not what you say”
- Typical conclusion: this means that the plaintext should be authenticated and not the ciphertext.
- So, in our secure channel, we should do MAC-then-Encrypt
  - Because then the MAC protects the plaintext
- However, for *secure channel symmetric encryption* state-of-art in cryptography for symmetric encryption suggests that Encrypt-then-MAC is better.
  - Security proofs for Encrypt-then-MAC can be shown in more security models (~ succeed against a slightly stronger attacker)

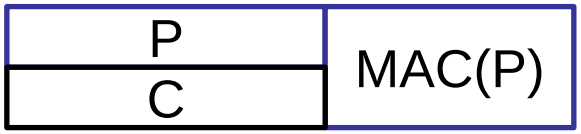
Encrypt-then-MAC, MAC-then-Encrypt, MAC&Encrypt

What are the options?

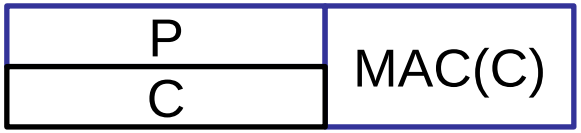
- MAC-then-Encrypt („SSL“)
  - Proposed by Horton Principle
  - Protects MAC, but not ciphertext C
  - Ciphertext can be interfered with



- MAC&Encrypt („SSH“)
  - Also follows Horton Principle
  - MAC and ciphertext not protected
  - Assumed to be the weakest of the three



- Encrypt-then-MAC („IPSec“)
  - Most supported by research (for secure channel)
  - Protects ciphertext C



Horton Principle (philosophic)

- Horton principle: “Authenticate what you mean, not what you say”

- But what is the meaning of a data transport channel?
  - Isn‘t it naive to think of the plaintext as what you mean?

- When we say the meaning is the data unit of the higher layer protocol, then it is the plaintext.
- When we say the meaning is the transport of bits, then we might also be able to think of the ciphertext as the meaning.
  - Logically it is not forbidden by the sentence that meaning and saying is the same.

- Whatever you might think about the philosophic question above, *the main message is that the mechanisms of one layer are not about meanings of other layers. The Secure Channel has no semantics for application-specific data*.

Secure Channel Toy Example

- The message processing in the secure channel consists of
  - Message numbering
  - Authentication
  - Encryption
- There are two approaches for the order of applying the authentication and the encryption to a message

  (1) One may either encrypt the message first, and then authenticate the obtained cipher text

  (2) Or one might authenticate first and then encrypt the message with the MAC value together
  (this approach is used subsequently)
- Both approaches have advantages and disadvantages
  - If encryption is applied first (1), Bob can discard bogus messages before spending CPU resources on decrypting them
  - If authentication is applied first (2), the MAC value will be also protected.
  - Also, the Horton principle: “Authenticate what you mean, not what you say” ➔ (2)
  - See [Fer03] for further details on this discussion

Message Numbering

- Message numbers allow Bob to reject replayed messages.
- They tell Bob which messages got lost in transit.
- They ensure that Bob receives the messages in their correct order.
- Messages numbers increase monotonically,

  i.e. later messages have a greater message number.
- Message numbers have to be unique;

  i.e. no two messages may have the same message number.
- A simple message numbering scheme functions as follow:
  - Alice numbers the first message as 1, the second message as 2, etc.
  - Bob keeps track of the last message number he has received.
  - Any new message must have a message number that is larger than the message number of the previous message.
  - If the message number overflows,

    e.g. message number is an 32-bit integer and the current message is 2^{32} -1, then Alice needs to stop using the current session key *K* before it can wrap back to 0.

Authentication/ Encryption

- For the data authentication, we need a MAC function,
  - e.g. HMAC-SHA-256
  - With a hash-value of 256 bits the collision rate is extremely low.
- The input to the MAC consists of
  - the message number i
  - the message m_i
  - extra authentication data x_i that is required by Bob to interpret m_i ,
    e.g., protocol version number, negotiated field size, etc.
  - Note: the length of x_i must be fix
- Let a_i := MAC (i || x_i || m_i)
  - The way how x_i is interpreted is out-of-scope and not a part of the functionality of the secure channel algorithm. The secure channel algorithm just considers it as a string.
  - However, the secure channel assures the integrity of x_i
- For encryption, we need an encryption algorithm,
  - e.g. AES in CTR mode with 256 bits (since it is pretty fast and secure)
- Frame Format
  - the message that Alice finally sends to Bob consists of
    message number i, followed by E(m_i || a_i).

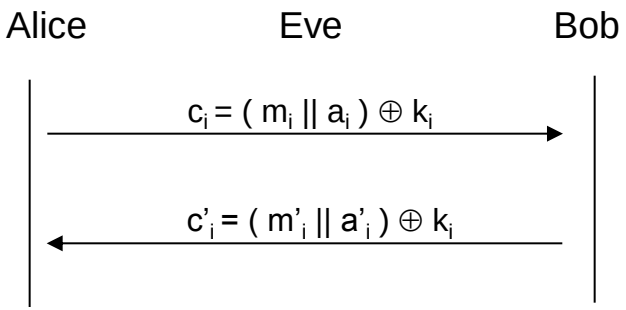
Initialization of the Secure Channel (1)

- The initialization procedure of the secure channel generates 4 different keys from the existing shared session key *K:*
  - An encryption key and an authentication key to send messages from Alice to Bob.
  - An encryption key and an authentication key to send messages from Bob to Alice.
  - It is strongly recommended not to reuse the same key for different purposes.
- Moreover, the initialization procedure sets the initial message numbers.

Possible Attacks on Secure Channels

- If the same key *K* is used for different purposes, different attacks become possible.
- E.g. **If *K* is used for encryption in both directions**, then *known-plain-text* attacks become possible:
  - Since encryption is done with AES in CTR mode
  - and assuming that Alice and Bob initialize the counter for the generation of the key streams k_i in the same way (starting with 0 and incrementing by 1 for each message)
  - The key stream k_i generated for each message m_i depends only on *K* and the sequence number *i*
  - Therefore, for each sequence number *i* the same key stream k_i will be generated on both sides.
  - If an attacker can guess a plain text m_i then, it can decrypt m'_i

    $m'_i \,||\, a'_i = c_i \oplus c'_i \oplus (m_i \,||\, a_i)$
  - Note: Eve does not need to guess a_i for this attack, since it can perform the xor operation only on the first bits that include m_i.



Initialization of an Example Secure Channel (2)

Function InitializeSecureChannel

Input:	K	Key of the channel
	R	Role: Specified if this party is Alice or Bob
Output	S	State for the secure channel

// First compute the 4 keys that are needed

KeySendEnc ← SHA-256 (K || „Enc Alice to Bob“)
KeyRecEnc ← SHA-256 (K || „Enc Bob to Alice“)
KeySendAuth ← SHA-256 (K || „Auth Alice to Bob“)
KeyRecAuth ← SHA-256 (K || „Auth Bob to Alice“)

// The strings „Enc Alice to Bob“, „Enc Bob to Alice“, etc. are simply used to generate different (uncorrelated) keys.They can be also substitued by other strings, e.g. „A“, „B“, „C“ and „D“.

// Swap the encryption and decryption keys if this party is Bob

If R = „Bob“ then { SWAP (KeySendEnc, KeyRecEnc)
SWAP (KeySendAuth, KeyRecAuth)
}

// Set the send and receive counters to zero. The send counter is the number of the last sent message. The receive counter is the number of the last received message

(MsgCNTSend, MsgCNTRec) ← (0,0)

// Package the state
S ← (KeySendEnc, KeyRecEnc, KeySendAuth, KeyRecAuth, MsgCNTSend, MsgCNTRec)
return S

Example – Sending a Message

Function SendMessage

Input:	S	Secure session state
	m	message to be sent
	x	addtional data to be authenticated
Output	t	data to be transmitted to the receiver

// First check the number and update it

if (MsgCNTSend >= MAX_MSG_NUMBER){

    print „MsgCNTSend overflow; re-keying is required“

    exit

}

MsgCNTSend ← MsgCNTSend + 1

i ← MsgCNTSend

// Compute the authentication

a ← HMAC-SHA-256 (KeySendAuth , i || x || m)

// Generate key stream k

k ← Enc(KeySendEnc, nonce || 0) || Enc(KeySendEnc, nonce || 1) || ...

// the message number *i* can be used as nonce as this would garantee

// that (nonce || block-number) will be unique each time E is applied

// Form the final text (i is an integer of 4 bytes length)

m_a ←m||a

t ← i || (m_a ⊕ first-lenght(m_a)-bytes(k))

return t

Example – Receiving a Message

Function ReceiveMessage

Input:	S	Secure session state
	t	text received from transmitter
	x	addtional data to be authenticated
Output	m	message that was sent

// Split t into i and the encrypted message plus authenticator.
// This split is unambiguous since i is an integer of 4 bytes length
i || t‘ ← t
// Generate the key stream, just as the sender did
k ← Enc(KeyRecEnc, nonce || 0) || Enc(KeyRecEnc, nonce || 1) || ...
// Decrypt the message and MAC field, and split.
// This split is also unambiguous since length of MAC value a is known (in this case 256 bits)
m || a ← t‘ ⊕ first-length(t‘)-bytes(k)
// Recompute the authentication
a’ ← HMAC-SHA-256 (KeyRecAuth, i || x || m)
if (a‘ ≠ a) {
    destroy k, m
    return MsgAuthenticationFailure }
else if (i <= MsgCNTRec) {
    destroy k, m
    return MessageOrderError }
MsgCNTRec ← i

return m

Message Reordering during Transmission

- The presented algorithm for processing received messages guarantees that no message is received twice.
- However, messages that are re-ordered during transmission, otherwise perfectly valid, will be lost.
- In some situations this can be inefficient, e.g. with IP packets, since they can be reordered during transport.
- We will see that IPSec, the IP security protocol that encrypts and authenticate IP packets, deals with this problem by maintaining a replay protection window instead of a single counter.

Further Design Criteria for a Secure Channel

- Negotiation of cryptographic algorithms
  - AES-256 and SHA-256 are just examples in this secure channel
  - Most security protocols support the negotiation of the cryptographic algorithms to be used.
  - This is the case, e.g., for IPSec and SSL

Further Design Criteria for a Secure Channel

- Multiple communication partners simultaneously
  - In many cases, Alice wishes to communicate with several partners simultaneously, e.g. with Bob and Carol.
  - Bob and Carol may use different cryptographic algorithms.
  - Therefore, Alice needs to know how to handle a message received from Bob or from Carol.
  - Each message needs to include a unique identifier for the connection in order to facilitate this task.
  - E.g.
    - In IPSec, this identifier is called the Security Parameter Index (SPI)
    - In SSL/TLS, there is a so-called Session Identifier (SessionID)
    - (Both IPSec and SSL/TLS will be explained in subsequent chapters of this lecture)

Secure Channels - Conclusions

- The secure channel is one of the most useful application of cryptography.
- Given good encryption and authentication primitives, it is possible to construct a secure channel.
- However, there are a lot of small details to pay attention to.
- Some applications require encryption.
- However, in most cases, authentication is more important.
- In fact Eve can cause a lot more damage if she manipulates messages and sends bogus messages, than by just listening to the message sent by Alice.
- Secure channel does not work without
  - Establishing the shared secret
  - Knowing to whom you are talking to ➔ Entity authentication

Overview

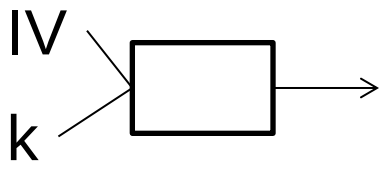
- Part I: The Secure Channel
- **Part II: Attacks against Secure Channel**
- Part III: Authenticated Encryption

Attacks against Secure Channel with Stream Ciphers

- Part I: The Secure Channel
- Part II: Attacks against Secure Channel
  - **Attacks against Secure Channel with Stream Cipher**
- Part III: Authenticated Encryption

Re-use of Initialization Vector

- Re-use of Initialization Vector (IV)



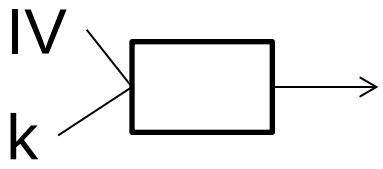
1 0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0 1

xor

P1 = 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0

C1 = 1 0 0 1 0 0 1 1 1 1 1 0 1 1 0 1 1 1 1

Then some time later the same IV is used again:



1 0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0 1

xor

P2 = 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1

C2 = 0 1 1 1 0 0 0 1 0 1 0 0 0 1 1 1 1 1 0

Re-use of Initialization Vector

- Re-use of Initialization Vector (IV) continued

C1 = 1 0 0 1 0 0 1 1 1 1 1 0 1 1 0 1 1 1 1

C2 = 0 1 1 1 0 0 0 1 0 1 0 0 0 1 1 1 1 1 0

C1+C2 = 1 1 1 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1

= = = = = = = = = = = = = … → P1+P2=C1+C2

P1+P2 = 1 1 1 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1

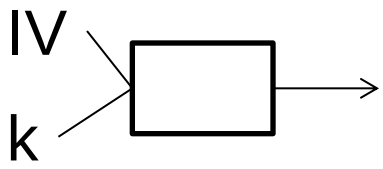
P1 = 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0

P2 = 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1

- As we see from the example, the attacker can computer C1+C2 because he observes C1 and C2, but that means he knows also P1+P2.
- Known Plaintext (e.g. P1) ➔ attacker can compute other plaintext
- Statistical properties of plaintext can be used if plaintext is not random-looking. That means if entropy of P1+P2 is low.

Weak Integrity Check, Linearity of Stream Cipher

- No integrity check or weak integrity check, e.g. CRC in WEP
  - To simplify example, we use the last bit as parity bit to check integrity.



1 0 1 1 0 0 0 1 0 1 0 1 1 0 0 0 1 0 1

xor

P =	0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0	8 „1“s → 0
C =	1 0 0 1 0 0 1 1 1 1 1 0 1 1 0 1 1 1 1	
Δ =	1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1	
CΔ =	0 1 0 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 0	
PΔ =	1 1 1 0 0 0 1 0 1 1 1 1 0 1 0 1 0 1 1	11 „1“s → 1 → ok

- Attacker can target individual bits, plaintext and checksum are linear in ciphertext. Thus, checksum can be overcome and targeted edits in a text could be done (e.g. change price information).

Same issues with block cipher modes?

- The attacks from the previous slides would not have worked that way against a block cipher mode like CBC.
  - The re-use of an IV can give hints about identical first blocks, but plaintext cannot be calculated from it.
    - The plaintext is not linear in the ciphertext. Thus, such trivial attacks won‘t work.
  - Weak checksums cannot be attacked directly, since single individual bits cannot be controlled by an attacker modifying the cipher text, again due to the fact that the plaintext is sent through the encryption algorithm.
- However, the attacks resulted from severe usage errors and not from proper use.
  - Moreover, integrity is not the goal of encryption and, thus, the weak checksum algorithm should be blamed.
  - Block cipher modes can also fail when used badly.
- We will learn about attacks against block cipher modes next!

Attacks against Secure Channel with Stream Ciphers

- Part I: The Secure Channel
- Part II: Attacks against Secure Channel
  - **Padding Oracle Attack against bad combination of CBC mode and MAC**
- Part III: Authenticated Encryption

Guessing a secret (revisited)

- Passwords
  - N: size of alphabet (number of different characters)
  - L: length of password in characters
- Complexity of guessing a randomly-generated password / secret
  - The assumption is, we generate a password and then we test it.

  → $O(N^L)$
- Complexity of guessing a randomly-generated password character by character
  - The assumption is that we can check each character individually for correctness.
  - For each character it is N/2 (avg) and N (worst case)
  - So, overall L*N/2 (avg)
- In the subsequent slides we will show an attack that reduces the decryption of a blockcipher in CBC mode to byte-wise decryption (under special assumptions).

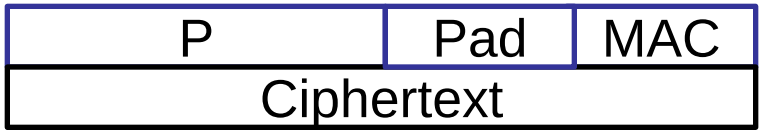
MAC-then-Encrypt Issues

- Operation
  - P and MAC are encrypted and hidden in the ciphertext.
  - Receiver
    - Decrypts P
    - Decrypts MAC
    - Computes and checks MAC →MAC error or success
- Consequence
  - MAC does not protect the ciphertext.
  - Integrity check can only be done once everything is decrypted.
  - As a consequence, receiver will detect malicious messages at the end of the secure channel processing and not earlier.
  - But is that more than a performance issue? Well, yes.

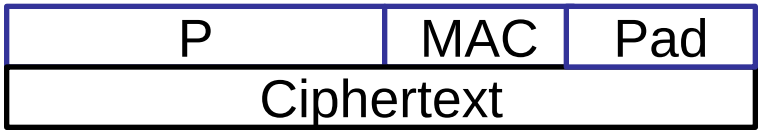
MAC-then-Encode-then-Encrypt

- If we use a block cipher, we have to ensure that the message encoding fits to the blocksize of the cipher.

- Encode-then-MAC-then-Encrypt:



  - Format P so that with the MAC added the encryption sees the right size.
  - Needs that we know the size of the MAC and blocksize of cipher when generating P | Padding.
- MAC-then-Encode-then-Encrypt



  - Used in TLS/SSL
  - Here, we add the MAC first and then pad the P | MAC to the correct size.
  - How do we know what is padding and what not? Padding in TLS/SSL:
    - If size of padding is 1 byte, the padding is 1.
    - If size of padding is 2 bytes, the padding is 2 2.
    - If size of padding is 3 bytes, the padding is 3 3 3.
    - ….

Oracles and Side Channels

- In ancient times, people asked oracles for guidance.

- In computer science, oracles are functions that give as cheaply access to information that would otherwise hard to compute.
  - E.g. O(1) cost to ask specific NP-complete question → polynomial hierarchy

- In cryptography, an attacker can trigger some participant O in a protocol or communication to leak information that might or might not be useful.
  - Participant O may re-encrypt some message fragment
  - Participant O responds with an error message explaining what went wrong
  - Response time of participant O may indicate where error happened
  - Response time may leak information about key if processing time depends (enough) on which bits are set to 1.
    - More obvious for the computationally expensive public key algorithms, but implementations of symmetric ciphers have also been attacked.

Side Channels and Padding Oracles

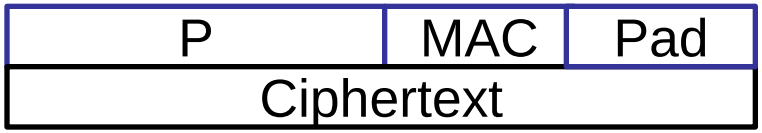
- Side Channel Attacks
  - A general class of attacks where the attacker gains information from aspects of the physical implementation of a cryptosystem.
  - Can be based on: Timing, Power Consumption, Radiation, …

ok

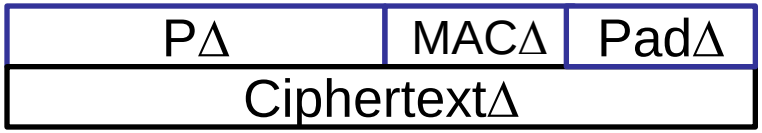
- Padding Oracle
  - The oracle tells the attacker if the padding in the message was correct.
  - This may be due to a *message with the information*.
  - It can also be due to *side channel like the response time*.

Concept of Padding Oracle Attack (against CBC)

- Attacker sees unknown ciphertext C = that was sent from Alice to Bob



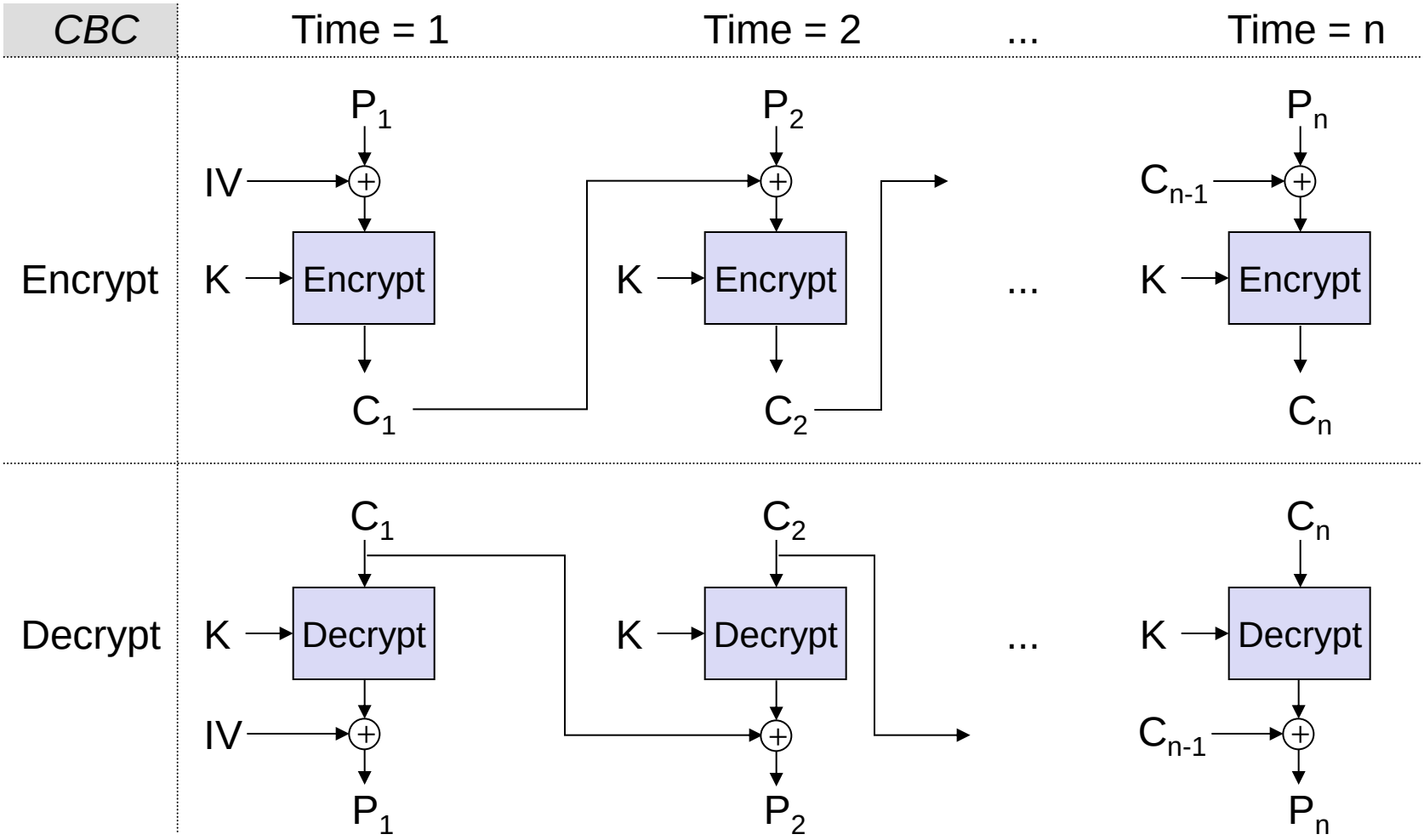
- To decrypt the ciphertext, the attacker modifies C and sends it to Bob.



- It is unlikely that the MAC and padding are correct. So, Bob will send an error back to Alice (and the attacker).
- In earlier versions of TLS, Bob sent back different error messages for padding errors and for MAC errors.

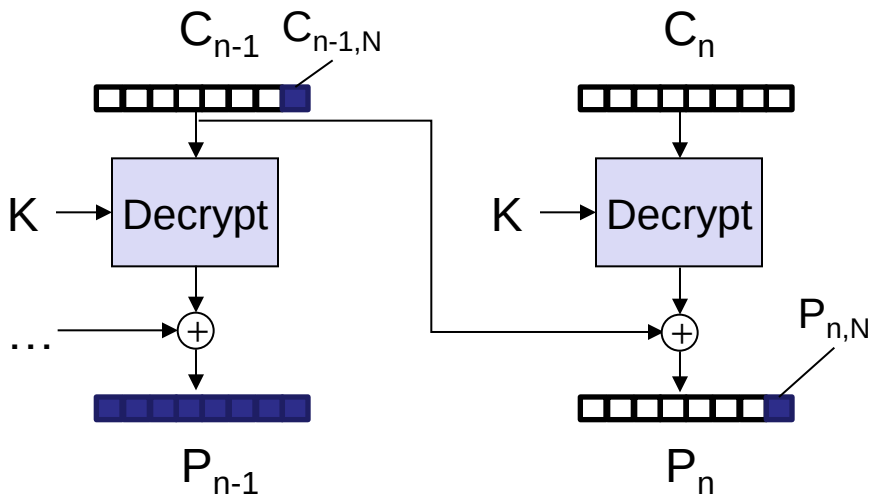
Padding Oracle Attack – CBC mode decryption (revisited)

- Encryption and Decryption in CBC mode



Padding Oracle Attack against CBC

- We have n blocks and N bytes per block. The attacker first wants to decrypt the last block C_n.
- In order to do so, he starts with the last byte $C_{n-1,N}$ of the block C_{n-1}. If he changes this byte (blue bytes are changed bytes)



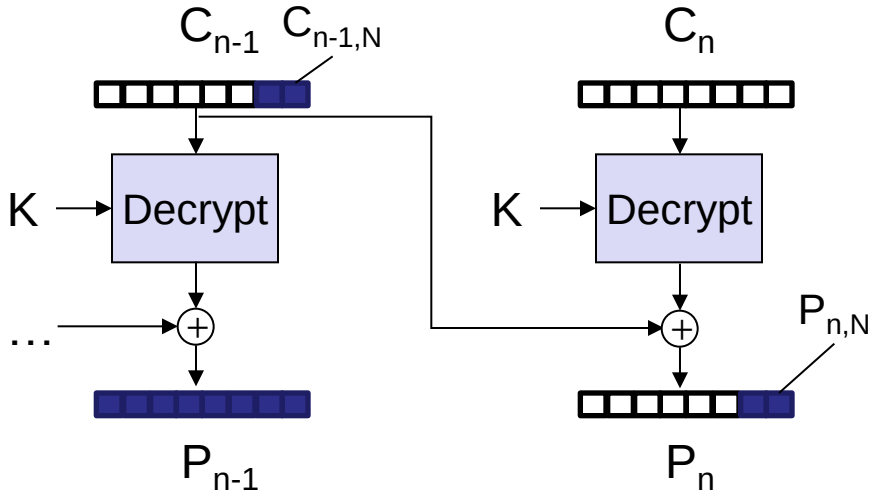
  - the MAC will most likely be invalid (chance 1 in 2^m for MAC length m)
  - the padding will be invalid unless $C_{n-1,N}$ xor $P_{n,N}$= 1 (chance 1 in 256)

➔ After testing the 256 values for **C**$_{n-1,N}$ all of them produced padding errors except for one that matches $C_{n-1,N}$ xor $P_{n,N}$= 1.

➔ We know $P_{n,N}$.

Padding Oracle Attack against CBC (2)

- Now, the byte $P_{n,N-1}$. For that we produce a padding of length 2.
- Since we know $P_{n,N}$ we can calculate $C_{n-1,N}$ so that $C_{n-1,N}$ xor $P_{n,N}$= 2
- Now, we have to find the $C_{n-1,N-1}$ that satisfies $C_{n-1,N-1}$ xor $P_{n,N-1}$= 2

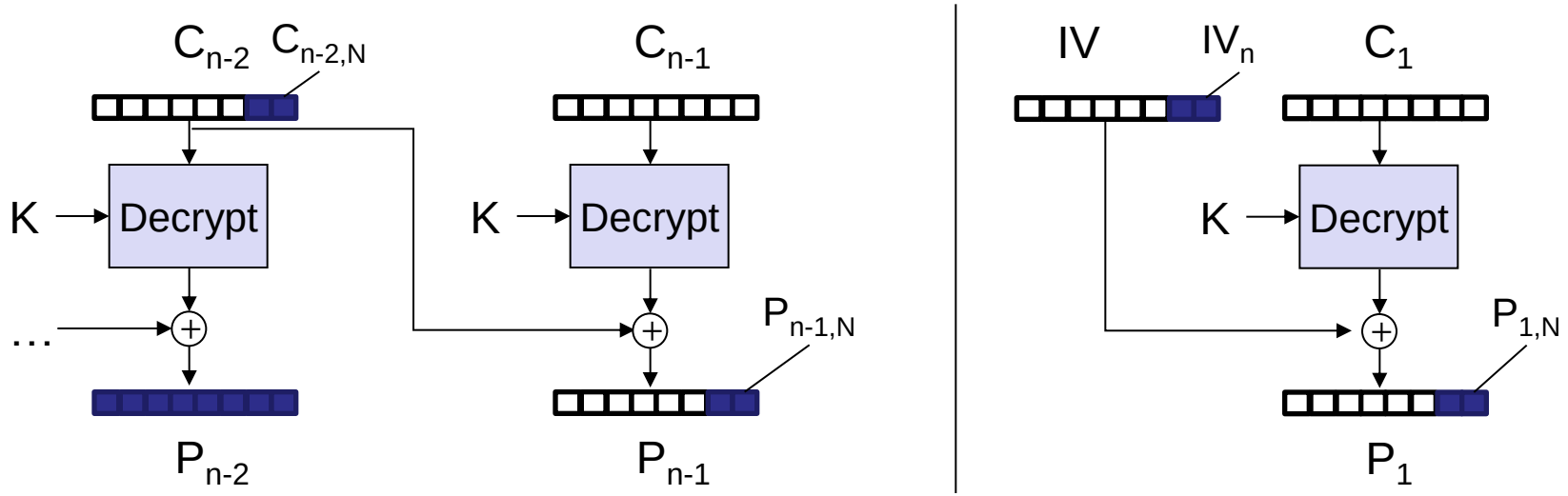


- With the same argument as before, we need to try up to 256 values, all values except for the correct one will generate a padding error. The correct one will produce a MAC error.

  ➔ We know $P_{n,N-1}$.

Padding Oracle Attack against CBC (3)

- To completely decrypt C_n we have to repeat the procedure until all bytes of the block are decrypted. In the figure with 8 bytes per block, the last padding we generate is 8 8 8 8 8 8 8 8.
- To decrypt C_{n-1} we can cut off C_n and repeat the same procedure with C_{n-1} as last block. For decrypting C_1 we can use the IV as ciphertext for the attack modifications.



Final Remarks

- The attack was against CBC mode used in MAC-then-Encode-then-Encrypt mode.
  - Padding Oracle attack known long in cryptography.
  - Mode still used in SSL / TLS. Hacks have utilized that. However, defenses have been added.

- CBC with Encode-then-Encrypt-then-MAC does not have this vulnerability.
  - Because MAC check would fail first, process would be aborted, and padding problems would then not be leaked.

Overview

- Part I: The Secure Channel
- Part II: Attacks against Secure Channel
- **Part III: Authenticated Encryption**

Authenticated Encryption

- Observations and Thoughts
  - Encryption → go over the data with some encryption mode
  - Integrity and authentication → go over the data with some MAC mode
  - Usually, both is needed. → Two passes over the data.
  - Difficult to do right. → Why not simplify process by providing both with one API call.

- Authenticated Encryption (AE)
  - Block Cipher Mode that provides Confidentiality, Integrity, and Authenticity
    - Any combination (e.g. AES-CTR-SHA-1-HMAC) would fall into the category
  - Some modern authenticated encryption modes do not *combine an encryption mode with a MAC mod*e, but they provide *both in one mode*.
    - Needs only one pass over the data.
  - Examples for AE modes are GCM (Galois/Counter Mode), OCB (Offset Codebook Mode), CCM (Counter with CBC-MAC).

Offset Codebook Mode (OCB)

- Offset Codebook Mode
  - Authenticated Encryption Mode
  - Proposed 2001 [OCB1]
  - Standardized May 2014 [RFC 7253]

  - Encryption
    - Inspired by ECB with block-dependent offsets (avoids ECB problems!)
  - Associated Data *A*
    - *A* is not encrypted but authenticated
    - For example: Unencrypted header data
  - MAC
    - Checksum = XOR over plaintext, length- and key-dependent variables
    - MAC = (Encryption of checksum with shared key k) XOR (hash(k,A))

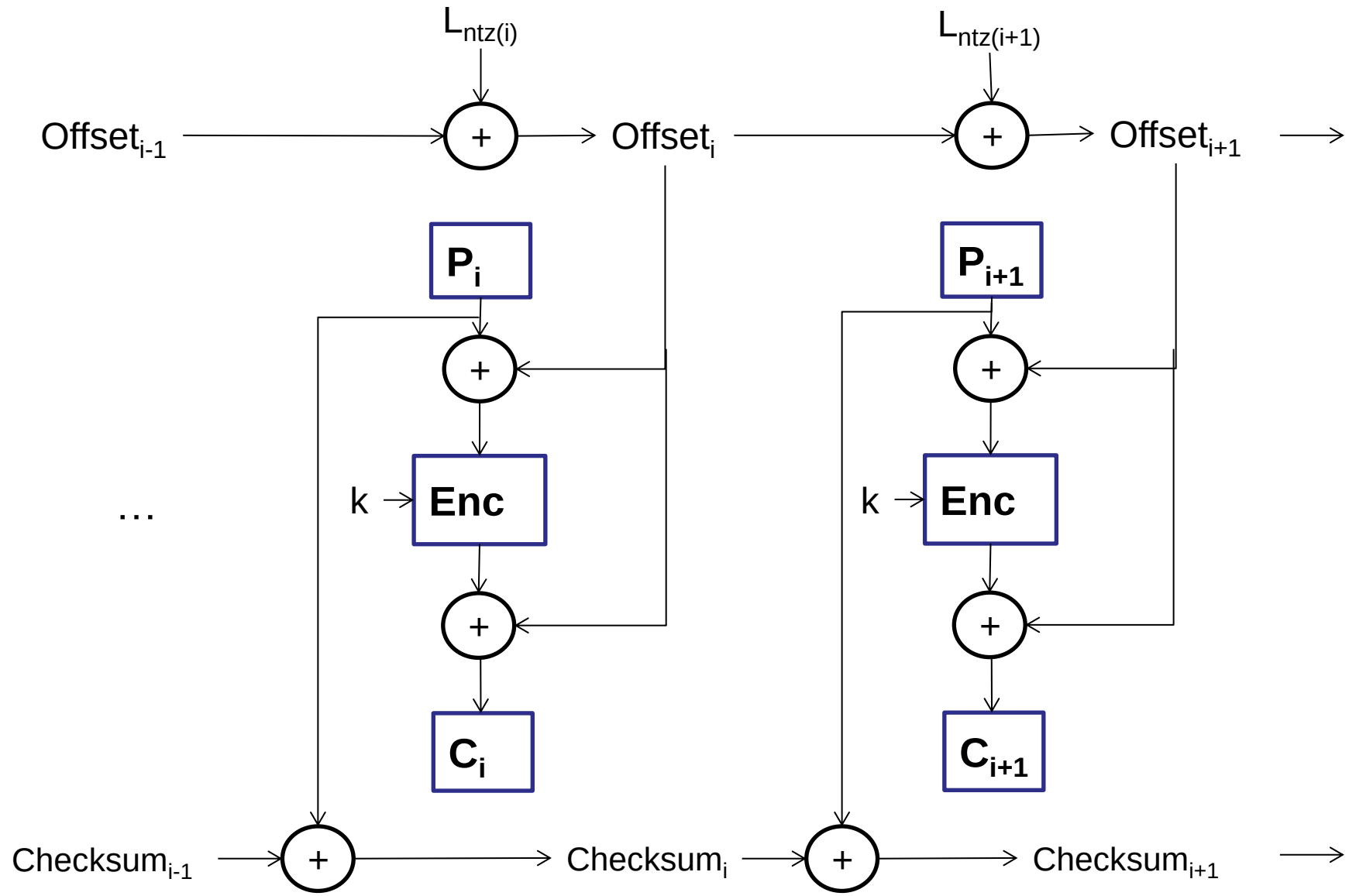
  - Requires only one key K for encryption and authentication
  - Requires a fresh nonce every time

Offset Codebook Mode

- Let *double* be multiplication by the variable in the OCB Galois Filed
- Variables depending on the key: $L_\star$, $L_\$$, L_0, L_1, L_2 …
  - $L_\star = Enc_K(0)$
  - $L_\$ = \text{double}(L_\star)$
  - $L_0 = \text{double}(L_\$)$
  - $L_i = \text{double}(L_{i-1})$
- Let *ntz* be number of trailing zeros (zero bits at the end)
- Usage of the L's
  - $L_\$$ → MAC
  - $L_\star$ → Last Block
  - $L_{ntz(i)}$ → intermediate blocks

- Note: $L_{ntz(i)}$ is used
  - Only few L_i are needed (for a fixed K)
  - They can be pre-computed and stored in a **L**ookup table

Offset Codebook Mode (OCB)



OCB Initialization

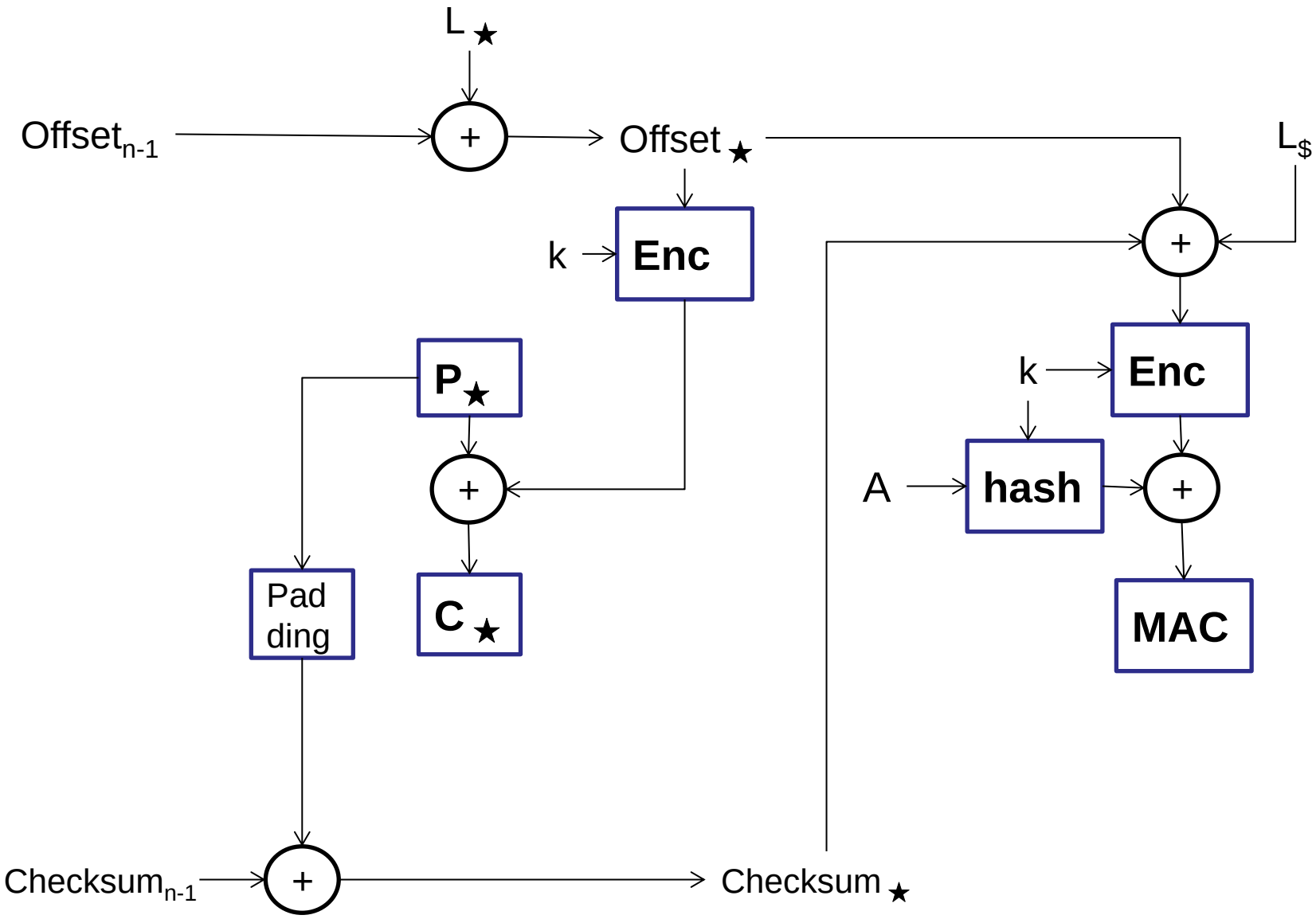
- Offset_0 depends on the key and the **nonce**
- “It is crucial that, as one encrypts, one does not repeat a nonce.”

  [RFC 7253, §5.1]
- Nonce *may* not be random, e.g. a counter works fine
- A new nonce for every authenticated encryption API call is needed!

- Details about the initialization:

  http://www.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm

OCB – Last Block and MAC



Offset Codebook Mode

- Question: XOR plaintext and then encrypt, that sounds like the weak MAC example from Chapter 2.2. Why is OCB more secure than the easy-to-break example?

- “OCB enjoys provable security: the mode of operation is secure assuming that the underlying blockcipher is secure. As with most modes of operation, security degrades as the number of blocks processed gets large” [RFC 7253]

Galois/Counter Mode (GCM)

- Galois/Counter Mode (GCM)
  - Developed by John Viega and David A. McGrew
  - Standardized by NIST in 2007, IETF standards for cipher suites with AES-GCM for TLS (SSL) and IPSec exist.
  - Follows the Encrypt-then-MAC concept
  - Combines concept of Counter Mode for encryption with Galois Field Multiplication to compute MAC on the ciphertext
  - GF(2^128) based on polynomial x^128 + x^7 + x^2 + x+1

- Definitions
  - H is Enc(k,0)
  - Auth Data is data not to be encrypted. GCM generates check value by XOR and GF multiplication with H for each block.
  - For the MAC, this process continues on the ciphertext and a length field in the end.

Galois/Counter Mode (GCM)

Starts with IV,
not with 0.

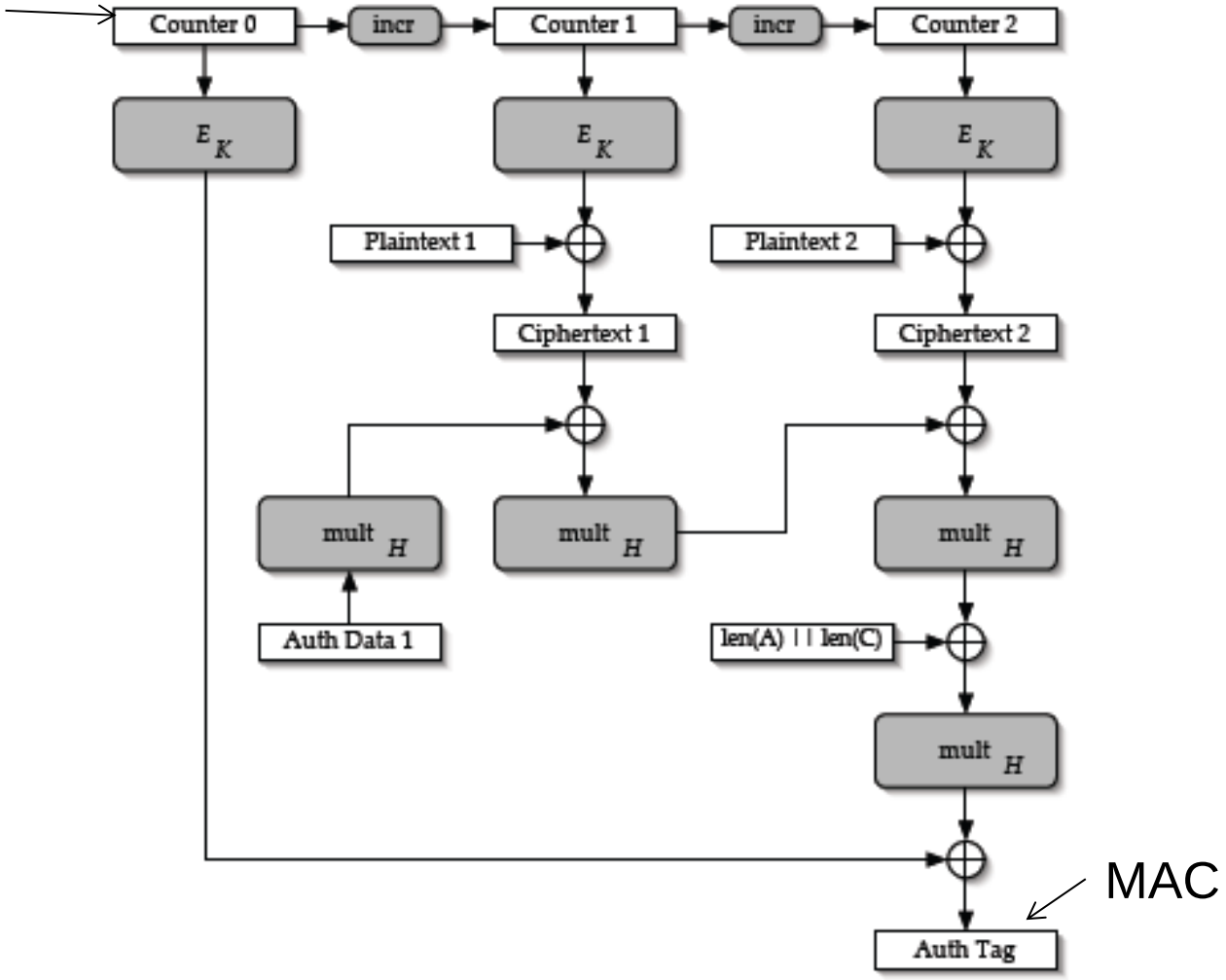


Image from Wikipedia, Author from NIST.

Galois Field Multiplication

- In a Galois Field we consider the bitstring to represent a polynomial.
  - E.g. 1011 = x^3 + x +1
- As a consequence Galois Field Multiplication is based on polynomial multiplication modulus the polynomial of the field.

- Example: In GF(2^128) based on polynomial g(x) = x^128 + x^7 + x^2 + x+1
  - P(x) = x^127+x^7
  - Q(x) = x^5 + 1
  - P(x)*Q‘(x) = x^132 + x^127 + x^12 + x^7
  - To compute the modulus, we have to compute a polynomial division P(x)*Q(x)/g(x).
  - We can see that x^4 * g(x) removes the x^132, so P(x)*Q(x)-x^4*g(x) = x^127 + x^12 + x^11 + x^7 + x^6 + x^5 + x^4
  - Since this polynomial fits into the 128 bit, this is the remainder of the division, thus the result, in bits: 1000…01100011110000.

References

[Bell95] M. Bellare and P. Rogaway, Provably Secure Session Key Distribution - The Three Party Case, Proc. 27th STOC, 1995, pp 57--64

[Boyd03] Colin Boyd, Anish Mathuria, “Protocols for Authentication and Key Establishment”, Springer, 2003

[Bry88a] R. Bryant. *Designing an Authentication System: A Dialogue in Four Scenes.* Project Athena, Massachusetts Institute of Technology, Cambridge, USA, 1988.

[Diff92] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. Designs, Codes, and Cryptography, 1992

[Dol81a] D. Dolev, A.C. Yao. *On the security of public key protocols.* Proceedings of IEEE 22nd Annual Symposium on Foundations of Computer Science, pp. 350-357, 1981.

[Fer00] Niels Ferguson, Bruce Schneier, “A Cryptographic Evaluation of IPsec”. http://www.counterpane.com/ipsec.pdf 2000

[Fer03] Niels Ferguson, Bruce Schneier, „Practical Cryptography“, John Wiley & Sons, 2003

[Gar03] Jason Garman, “Kerberos. The Definitive Guide”, O'Reilly Media, 1st Edition, 2003

References

[Kau02a] C. Kaufman, R. Perlman, M. Speciner. *Network Security*. Prentice Hall, 2nd edition, 2002.

[Koh94a] J. Kohl, C. Neuman, T. T'so, *The Evolution of the Kerberos Authentication System.* In Distributed Open Systems, pages 78-94. IEEE Computer Society Press, 1994.

[Mao04a] W. Mao. *Modern Cryptography: Theory & Practice*. Hewlett-Packard Books, 2004.

[Nee78] R. Needham, M. Schroeder. *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, Vol. 21, No. 12, 1978.

[Woo92a] T.Y.C Woo, S.S. Lam. *Authentication for distributed systems.* Computer, 25(1):39-52, 1992.

[Lowe95] G. Lowe, „An Attack on the Needham-Schroeder Public-Key Authentication Protocol”, *Information Processing Letters*, volume 56, number 3, pages 131-133, 1995.

References

[OCB1] Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption", ACM Conference on Computer and Communications Security 2001 - CCS

[OCB] T.Krovetz, P. Rogaway, „The OCB Authenticated-Encryption Algorithm“ http://tools.ietf.org/html/draft-irtf-cfrg-ocb-03

[RFC 4106] The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)

[RFC 5288] AES Galois Counter Mode (GCM) Cipher Suites for TLS.

[RFC 7253] The OCB Authenticated-Encryption Algorithm

Additional references from the IETF

[RFC2560] M. Myers, et al., “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP”, June 1999

[RFC3961] K. Raeburn, “Encryption and Checksum Specifications for Kerberos 5”, February 2005

[RFC3962] K. Raeburn, “Advanced Encryption Standard (AES) Encryption for Kerberos 5”, February 2005

[RFC4757] K. Jaganathan, et al., “The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows ”, December 2006

[RFC4120] C. Neuman, et al., “The Kerberos Network Authentication Service (V5)”, July 2005

[RFC4537] L. Zhu, et al, “Kerberos Cryptosystem Negotiation Extension”, June 2006

[RFC5055] T. Freeman, et al, “Server-Based Certificate Validation Protocol (SCVP)”, December 2007