Midterm

Problem 1

In GF (2^4) Compute $f(x)+g(x), f(x)-g(x), f(x)\times g(x)\ mod\ P(x)$, where $P(x)=x^4+x+1$.

a) $f(x) = x^2 + 1,\ g(x) = x^3 + x^2 + 1$.

1. Compute $f(x)+g(x)$

In finite fields, addition is performed by adding the coefficients of like terms and then taking modulo 2 (as we are in GF(2^4)):

$f(x)+g(x)=(x^2+1)+(x^3+x^2+1)=x^3+2x^2+2=x^3$

The terms $2x^2$ and 2 become 0 because 2 modulo 2 is 0.

2. Compute f(x) − g(x)

Subtraction in finite fields is like addition because -1 ≡ 1 (mod2).

$f(x)$ - $g(x)$ = (x^2+1) - (x^3+x^2+1) = - x^3= x^3

3. Compute $f(x)\times g(x)\ mod\ P(x)$

$f(x)\times g(x)=(x^2+1)(x^3+x^2+1)=x^5+x^4+x^3+x^2+x^2+1=x^5+x^4+x^3+2x^2+1=x^5+x^4+x^3+1$

now reduce x^5 and x^4 using P(x):

x^5 mod P(x) = x(x^4) mod P(x) = x(x+1) = x^2+x

x^4 mod P(x) = x+1

so:

$x^5+x^4+x^3+1=(x^2+x)+(x+1)+x^3+1=x^3+x^2+2x+2=x^3+x^2$

b) $f(x) = x^2 + 1,\ g(x) = x + 1$.

1. Compute $f(x)+g(x)$

$f(x)+g(x)=(x^2+1)+(x+1)=x^2+x+2=x^2+x$

2. Compute f(x) − g(x)

Subtraction in finite fields is like addition because -1 ≡ 1 (mod2).

$f(x)$ - $g(x)$ = (x^2+1) - $(x+1)$ = x^2 - x

3. Compute $f(x)\times g(x)\ mod\ P(x)$

$f(x)\times g(x)=(x^2+1)(x+1)=x^3+x^2+x+1$

Reducing x^3 using P(x):

x^3mod P(x) = x+1

so:

$x^3 + x^2 + x + 1 = (x+1) + x^2 + x + 1 = x^2 + 2x + 2 = x^2$

Problem 2

In GF(2^8), $f(x) = x^7 + x^5 + x^4 + x + 1$ and $g(x) = x^3 + x + 1$.

a) Calculate $f(x) + g(x), f(x) - g(x), f(x) \times g(x)$ *mod* $m(x)$, where $m(x) = x^8 + x^4 + x^3 + x + 1$.

b) Show that $f(x) = x^4 + 1$ is reducible over GF (2^8).
Hint: it can be written as the product of two polynomials.

Solution:-

Given that

$f(x) = x^7 + x^5 + x^4 + x + 1$

$g(x) = x^3 + x + 1$

(i) $f(x) + g(x) = x^7 + x^5 + x^4 + x + 1$

$+ x^3 + x + 1$

$\therefore f(x) + g(x) = x^7 + x^5 + x^4 + x^3 + 2x + 2$

(ii) $f(x) - g(x) = x^7 + x^5 + x^4 + x + 1$

$- x^3 - x - 1$

$f(x) - g(x) = x^7 + x^5 + x^4 - x^3$

(iii) $f(x) \times g(x) \Rightarrow x^7 + x^5 + x^4 + x + 1$

$\times (x^3 + x + 1)$

$x^7 \cdot x^3 + x^7 \cdot x + x^7 + x^5 \cdot x^3 + x^5 \cdot x + x^5 + x^4 \cdot x^3 + x^4 \cdot x +$

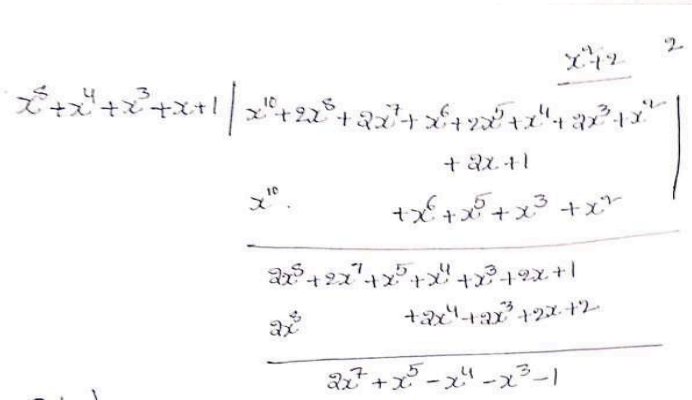
$x^4 \cdot 1 + x^3 \cdot x + x \cdot x + x + x^3 + x + 1$

$= x^{10} + x^8 + x^7 + x^8 + x^6 + x^5 + x^7 + x^5 + x^4 + x^3 + x^2 + x + x^3 + x + 1$

$\Rightarrow x^{10} + 2x^8 + 2x^7 + x^6 + 2x^5 + x^4 + 2x^3 + x^2 + 2x + 1$

GF(2^8) where $m(x) = x^8+x^4+x^3+x+1$

$f(x) \times g(x) \pmod{x^8+x^4+x^3+x+1}$



Get the result of $f(x) \times g(x) \bmod x^8+x^4+x^3+x+1$

$= 2x^7+x^5-x^4-x^3+1$

(b) A Polynomial $f(x)$ over field F is called reducible if and only if $f(x)$

can't be expressed as a product of two polynomials.

the polynomial $f(x) = x^4+1$

over GF(2^8) is reducible

Because it can be written as the product of two polynomials.

$x^4+1 = (x+1)(x^3+x^2+x+1)$

Product 1 Product 2

Hence we proved that

$\therefore f(x) = x^4+1$ is reducible over GF(2^8).

Problem 3

Consider the field GF (2^4) with the irreducible polynomial $P(x) = x^4 + x + 1$. Find:

a) the inverse of $f(x) = x$ and

The inverse of $f(x) = x$ in GF(2^4) under the irreducible polynomial $P(x) = x^4 + x + 1$ is : $h(x) = x^3+1$ .This means $x(x^3+1) \equiv 1 \pmod{x^4+x+1}$.

b) the inverse of $g(x) = x^2 + x$ by trial and error.

The inverse of $g(x) = x^2+x$ in GF(2^4) under the irreducible polynomial $P(x) = x^4 + x + 1$ is : $h(x) = x^2$ .This means $(x^2+1)(x^2) \equiv 1 \pmod{x^4+x+1}$.

Problem 4

In GF(2^8), find:

a) $(x^3 + x^2 + x)(x^3 + x^2)^{-1} \bmod (x^8 + x^4 + 1) =$

To find $(x^3 + x^2 + x)(x^3 + x^2)^{-1} \ mod \ (x^8 + x^4 + 1)$

Determine the inverse of $x^3 + x^2 \ mod \ (x^8 + x^4 + 1)$. The inverse $g(x)$ satisfies $(x^3+x^2)g(x) \equiv 1 \bmod (x^8+x^4+1)$

Using the Extended Euclidean Algorithm:

$x^8+x^4+1=(x^3+x^2) \cdot q(x)+r(x)$

Solving this, we get:

$(x)=x^5- x^3 + x, \quad r(x)=x^2+1$

Continuing this process:

$x^3 + x^2 = (x^2+1) * q(x) + r(x)$

$r(x)=x-1$

Solving these steps yields the inverse $g(x)$

$g(x)=x^5+x^3+x$

Multiply $x^3 + x^2 + x$ by the inverse of x^3+x^2:

$(x^3+x^2+x) \cdot (x^5+x^3+x) \bmod (x^8+x^4+1)$

Expanding and reducing modulo $(x^8 + x^4 + 1)$,we get:
$=(x^8+x^6+x^4+x^5+x^3+x+x^4+x^2) \bmod (x^8+x^4+1)$

Simplifying:

$= x^6+x^5+x^3+x^2+x$

b) $(x^6 + x^3 + 1)(x^4 + x^3 + 1) \ mod \ (x^8 + x^4 + 1) =$

To find $(x^6 + x^3 + 1)(x^4 + x^3 + 1) \ mod \ (x^8 + x^4 + 1)$:

Perform the polynomial multiplication:
$(x^6 + x^3 + 1)(x^4 + x^3 + 1) = x^{10}+x^9+x^7+x^6+x^4+x^3+1$

Reduce the result modulo $(x^8 + x^4 + 1)$:

Since $x^8 \equiv x^4 + 1 \ mod \ (x^8 + x^4 + 1)$:

$x^{10}=x^2(x^8) \equiv x^2(x^4+1) \equiv x^6+x^2$
$x^9=x(x^8) \equiv x(x^4+1) \equiv x^5+x$

Substitute these into the polynomial:
$x^{10}+x^9+x^7+x^6+x^4+x^3+1 \equiv (x^6+x^2)+(x^5+x)+x^7+x^6+x^4+x^3+1$

$=x^7+2x^6+x^5+x^4+x^3+x^2+x+1$

In GF(2), simplify (where $2x \equiv 0$)

$=x^7+x^5+x^4+x^3+x^2+x+1$

Problem 5

Regarding the mix column operation of the AES round function, it is performed with a pre-defined matrix, i.e.,

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

a) Explain why this **mix column operation** can be implemented with a simple look up table and XOR.

The mix column operation in AES is essentially a matrix multiplication in the finite field $GF(2^8)$. Each element of the state (a 4x4 matrix) is multiplied by a fixed 4x4 matrix.

Multiplication in $GF(2^8)$

In $GF(2^8)$ the multiplication can be reduced to bitwise shifts and XOR operations due to the properties of binary fields. Specifically, multiplication by 2, 3, 9, 11, 13, and 14 (as required by the mix columns and inverse mix columns steps) can be implemented using combinations of shifts and XORs.

Look-Up Tables:

To simplify the implementation, precompute the results of the multiplication of all possible byte values (0-255) by 2, 3, 9, 11, 13, and 14.

Store these precomputed results in look-up tables.

During the mix columns operation, instead of performing the multiplication directly, use the look-up tables to fetch the precomputed results.

Advantages of Look-Up Tables:

Speed: Accessing a value from a look-up table is faster than performing the actual multiplication.

Consistency: Ensures consistent and error-free multiplication since the results are precomputed.

Implementation:

For each byte in the state matrix, fetch the corresponding results from the look-up tables and XOR them according to the mix columns matrix.

b) Apply the same idea used above, explain why the **byte substitution**, **shift row** and **mix column** can be combined and implemented as a simple look-up table operation. And then the whole AES is implemented by look up table and few XORs.

Byte Substitution (SubBytes):

In AES, each byte of the state is replaced by its corresponding value from the S-box, which is a

non-linear substitution table.
This is already implemented using a look-up table.

Shift Row:
The ShiftRows step involves cyclically shifting the rows of the state matrix.
This can be viewed as a simple permutation of the bytes.

Combining Steps:
By combining the SubBytes, ShiftRows, and MixColumns steps, we can precompute the effect of applying these operations to each byte position in the state.
For each byte, precompute its transformation through SubBytes, its new position after ShiftRows, and the resulting bytes after MixColumns.

Precomputed Transformation:
Create a 3D look-up table where each entry corresponds to a byte's transformation through all three steps.
During encryption, use the look-up table to fetch the final transformed byte directly.

Advantages:
Efficiency: Reduces the number of operations by combining multiple steps into a single look-up.
Simplicity: Simplifies the implementation, making it faster and less prone to errors.

Example:
Consider a byte a in the state matrix at position (I,j).

Byte Substitution:
b=S-box[a]

Shift Rows:
c=ShiftRows(b,i)

Mix Columns:
d=MixColumns(c)

Combining these steps:
Precompute d=MixColumns(ShiftRows(S-box[a],i)) for each byte a and each row index i.

During encryption:
Fetch the precomputed result d from the combined look-up table.

Problem 6

In order to make AES encryption and decryption more similar in structure, the Mix-Columns operation is missing in the last round.

Explain how to take **advantage** of this property to share some of the code (for soft-ware implementation) or chip area (for hardware implementation) for AES encryption and decryption.

Hint: Denote InvSubBytes, InvShiftRows, and InvMixColumns as the inverses of Sub- Bytes, ShiftRows, and MixColumns operations, respectively. Try to show:
1. The order of InvSubBytes and InvShiftRows is indifferent;
2. The order of AddRoundKey and InvMixColumns can be inverted if the round key is adapted accordingly.

AES avoids the MixColumns operation in the last round. The construction of the last encryption round is exactly like the decryption round. Since the construction of the last round of AES encryption is like that of decryption, optimization by sharing code or hardware for encryption or decryption is possible. The steps InvSubBytes and InvShiftRows may be executed in any order because they are functions operating on separate bytes. By adjusting the order of the round key, AddRoundKey and InvMixColumns can be interchanged in decryption. These optimizations reduce redundancy and hence enhance efficiency in AES implementations with shared functions or hardware blocks.

Problem 7

Under **what circumstances** could you choose 3DES over AES? Also, what are the **ad- vantages** of choosing AES over 3DES? Is 3DES **susceptible** to Meet-in-the-Middle At- tacks like 2DES? Please explain.

Circumstances to Choose 3DES Over AES

- **Legacy Systems**:

  When systems and applications are already built around 3DES, and transitioning to AES would require significant modifications.

  Ensuring compatibility with older systems that only support 3DES, such as certain hardware devices or financial services protocols.

- **Regulatory Requirements**:

  In certain industries or regions, regulations might mandate the use of 3DES for specific applications. An example is financial transactions where 3DES might still be in use due to existing standards.

- **Resource Constraints**:

  In some cases, older or resource-constrained hardware may not support AES efficiently but can still handle 3DES adequately.

  For simple systems where the cryptographic strength of 3DES is considered sufficient and the implementation is straightforward.

Advantages of Choosing AES Over 3DES

- **Security:**

  AES is considered more secure than 3DES. AES has larger key sizes (128, 192, 256 bits) compared to the effective key length of 3DES (112 or 168 bits), providing a higher security margin against brute-force attacks.

  AES benefits from a modern cryptographic design, making it more resistant to various cryptographic attacks.

- **Performance:**

  AES is generally faster and more efficient than 3DES. It is optimized for both software and hardware implementations, resulting in better performance in encryption and decryption operations.

  AES uses fewer computational resources compared to 3DES, which requires three separate DES operations (encryption-decryption-encryption) for each data block.

- **Adoption and Support**:

  AES is widely adopted as the standard for symmetric encryption. It is supported by most modern systems, libraries, and protocols.

  Given its broad adoption and standardization (NIST FIPS 197), AES is more future proof compared to 3DES, which is being phased out in many applications.

Susceptibility to Meet-in-the-Middle Attacks

- **2DES (Double DES):**

  2DES is susceptible to Meet-in-the-Middle (MitM) attacks, significantly reducing its effective key length from 112 bits to about 56 bits. This makes it only marginally better than single DES in terms of security.

- **3DES (Triple DES):**

  3DES is designed to mitigate the weaknesses of 2DES, specifically addressing the vulnerability to MitM attacks. It uses three DES operations with two or three different keys (encrypt-decrypt-encrypt), resulting in an effective key length of 112 bits (for two keys) or 168 bits (for three keys).

  Although theoretically susceptible to MitM attacks, the effective security of 3DES with a 112-bit key is still considered adequate for many purposes. However, it is not as strong as AES with its 128, 192, or 256-bit key lengths.

Problem 8

If a company has encrypted its most sensitive data with a key held by the chief technology officer and that person was fired, the company would want to change its encryption key. Describe **what would be necessary** to revoke the old key and deploy a new one.
Hint: NIST Special Publication 800-57 Part 1

A company has encrypted its most sensitive data with a key held by the chief technology officer who has been fired. The company needs to change its encryption key.

Steps to Revoke the Old Key and Deploy a New One:
Assess the Situation:
Identify all data and systems encrypted with the old key.
Check for any potential compromise of the old key.

Generate a New Key:

Create a new, secure cryptographic key.
Store the new key in a secure key management system (KMS) or hardware security module (HSM).

Re-encrypt Data:
Decrypt data using the old key.
Re-encrypt data with the new key.
Automate the process, if possible, to ensure consistency and minimize errors.

Update Key References:
Modify system configurations to use the new key for encryption and decryption.
Ensure access controls reflect the key change.

Revoke the Old Key:

Remove the old key from the KMS, HSM, or other storage locations.
Log the revocation process for auditing purposes.

Communicate the Change:
Inform IT staff and security teams about the key change.
Update security policies and procedures to reflect the change.

Post-Implementation Review:
Verify that all data has been re-encrypted with the new key.
Conduct an audit to ensure compliance with policies and regulations.

Ongoing Key Management:
Establish a regular key rotation schedule. Continuously monitor access to encryption keys and encrypted data

Problem 9

Bitdiddle Inc. requires every employee to have an RSA public key. It also requires the employee to change his or her RSA key at the end of each month.

a) Alice just started working at Bitdiddle, and her first public key is (n, e) where n is the product of two safe primes, and $e = 3$.
Whenever a new month starts, Alice (being lazy) changes her public key as little as possible to get by the auditors. What she does, in fact, is just to advance her public exponent e to the next prime number. So, month by month, her public keys look like:

$$(n, 3), (n, 5), (n, 7), (n, 11), \ldots$$

Explain how Alice's laziness might get her in trouble.

In part (a), Alice's scheme of advancing her public exponent e to the next prime each month (e.g., 3, 5, 7, 11, etc.) makes the RSA encryption vulnerable to low exponent attacks, such as Coppersmith's attack, which exploit the small size and predictability of these public exponents.

b) The next year, Alice tries a different scheme.
In January, she generates a fresh public key (n, e) where n is the product of two primes, p and q. In February, she advances p to the next prime p_1 after p, and q to the next prime q_1 after q, and sets her public key to (n_1, e_1) for a suitable e_1. Similarly, in March, she advances p_1 to p_2 and q_1 to q_2, and so on.

Explain how Alice's scheme could be broken.

In part (b), Alice's method of generating new public keys by incrementing the prime factors p and q to the next primes p_1 and q_1 each month leads to a vulnerability in factorization. This predictable increment allows attackers to factorize n by trying likely candidates for p_1 and q_1, thereby compromising the security of the RSA encryption.

Problem 10

The Advanced Encryption Standard (AES) requires not only the MixColumns step during encryption but also the Inverse MixColumns during decryption.

a) Given the matrix for the Inverse MixColumns operation:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Describe how the matrix multiplication in the Inverse MixColumns step is performed within AES decryption. Your answer should include a **brief explanation of the arith- metic** used in the finite field GF(2^8) and **how it differs from standard matrix mul- tiplication**.

In AES decryption, the Inverse MixColumns step involves matrix multiplication within the finite field GF(2^8). The matrix used for this operation is fixed and consists of constants such as 0E, 0B, 0D, and 09. Each element of the state matrix is multiplied by these constants and combined using XOR to produce the final output. This arithmetic, performed in GF(2^8), uses modular operations with an irreducible polynomial (e.g., $x^8+x^4+x^3+x+1$), ensuring results remain within 8 bits. Unlike standard matrix multiplication, the operations in GF(2^8) involve both addition (XOR) and multiplication, which are specifically defined for the finite field. This allows for secure and efficient transformations necessary for the AES decryption process.

b) Multiplications in GF (2^8) can be complex. However, they can be simplified using repeated application of simpler operations. Explain how multiplication by 9, 11, 13, and 14 in GF (2^8) can be implemented using the simpler multiplication by 2 and addition (XOR). Provide the **mathematical expressions** that represent these multiplications.

In GF(2^8), multiplications by constants such as 9, 11, 13, and 14 can be simplified using repeated applications of simpler operations, specifically multiplication by 2 and XOR:

Multiplication by 9 (0x09):

$a \cdot 9 = a \cdot (8+1) = (a \cdot 2^3) \oplus a$
Here, $a \cdot 2^3$ means shifting a left by three bits and then reducing modulo the irreducible polynomial if necessary, followed by XOR with a

Multiplication by 11 (0x0B):

$a \cdot 11 = a \cdot (8+2+1) = (a \cdot 2^3) \oplus (a \cdot 2) \oplus a$
This expression indicates a is first multiplied by 2 three times, XORed with a multiplied by 2 once, and then XORed with a.

Multiplication by 13 (0x0D):

$a \cdot 13 = a \cdot (8+4+1) = (a \cdot 2^3) \oplus (a \cdot 2^2) \oplus a$

Similar to the above, a is multiplied by 2 three times, XORed with a multiplied by 2 twice, and then XORed with a.

Multiplication by 14 (0x0E):
$a \cdot 14 = a \cdot (8+4+2) = (a \cdot 2^3) \oplus (a \cdot 2^2) \oplus (a \cdot 2)$
Here, a is multiplied by 2 three times, XORed with a multiplied by 2 twice, and then XORed with a multiplied by 2 once.

These simplifications leverage the properties of the finite field $GF(2^8)$, where multiplication by 2 corresponds to a left shift, followed by a conditional reduction with the irreducible polynomial to ensure the result stays within 8 bits.

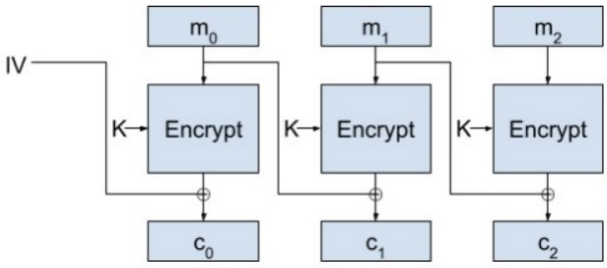
c) Discuss how the use of lookup tables (LUTs) can further simplify the implementation of Inverse MixColumns in AES decryption. What are the **advantages** and potential **drawbacks** of using LUTs for this purpose?

Using LUTs for Inverse MixColumns in AES decryption simplifies implementation by storing precomputed multiplication results, allowing for rapid retrieval, and eliminating the need for on-the-fly calculations. This approach increases efficiency and results in consistent, error-free results. On the other hand, it increases memory usage and creates potential security risks, like cache-timing attacks, and these need to be implemented carefully.

Hint 1: https://www.youtube.com/watch?v=dRYHSf5A4lw
Hint 2: https://dl.acm.org/doi/abs/10.1145/3405669.3405819
Hint 3: https://xilinx.github.io/Vitis_Libraries/security/2019.2/guide_L1/internals/aes.html

Problem 11

In class we saw several modes such as ECB, CBC and CTR mode. Let's look at another possible mode of operation "plaintext block chaining" (PBC) which is similar to cipher block chaining (CBC) but allows for encryption in parallel:



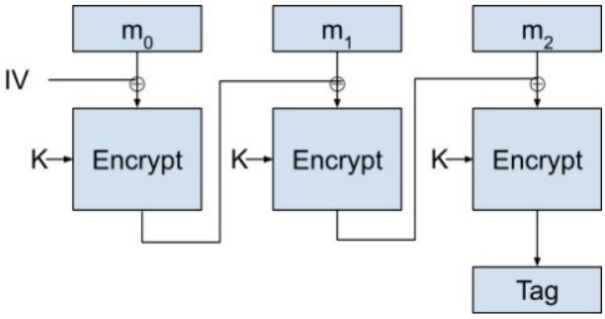
Unfortunately, PBC mode is not secure.

To see this, show how an attacker who knows IV, C_0, C_1, C_2 (which are public) and also knows that $m_1 = m_2 = x$ (for a known x) can easily compute m_0.

Plaintext Block Chaining (PBC) mode is insecure because an attacker who knows the IV, the first three ciphertext blocks C_0, C_1, and C_2, and that two plaintext blocks m_1 and m_2 are equal (e.g., $m_1=m_2=x$), can easily compute m_0. In PBC, each ciphertext block is derived from the previous ciphertext block and the current plaintext block. Given $m_1=m_2=x$, the attacker can use this knowledge to deduce the relationship between the plaintext and ciphertext blocks, ultimately revealing m_0. This predictability and the ability to compute unknown plaintext blocks make PBC mode insecure.

Problem 12

Alice is trying to design a MAC using a block cipher. She decides to use the following construction, which is essentially just CBC encryption, throwing away all but the final block.



Unfortunately, this construction is not secure.

Describe how to produce an existential forgery against this MAC scheme.

Hint 1: Start with two messages M_1 and M_2 (not to be confused with the individual blocks of a message in the diagram above) for which you know the outputs (IV_1, T_1) and (IV_2, T_2). Produce another message M_3 for which (IV_1, T_2) will be the MAC. M_3 will be close to the concatenation $M_1||M_2$, but with one block altered.

Hint 2: There is also a way to produce a forgery with only one known block if you look closely.

Caution: The blocks $m_0, m_1, \ldots$ *in the diagram are distinct from the complete messages*
$M_1, M_2, \ldots$

Alice's MAC scheme, which uses CBC encryption and discards all intermediate blocks except the final one, is insecure. An attacker can create an existential forgery by combining parts of two known messages,
M_1 and M_2, to form a new message M_3. By carefully choosing a modification Δ, the attacker can manipulate the intermediate state so that the final block of M_3 produces the same MAC as T_2. This allows the attacker to generate a valid MAC for M_3 without knowing the secret key, demonstrating the vulnerability of Alice's MAC scheme to forgery attacks.

Extra Credit 1

To help you understand how to develop security policies for systems used by many people and might be particularly relevant if you use GitHub or plan to design similar systems.

Please write a security policy for a website like GitHub that hosts distributed version control systems. Define the various **roles**, **functions**, and **policy** that should be in- cluded. This policy should be sufficiently detailed for someone who wants to create a site similar to GitHub. For this task, assume that GitHub is the official maintainer of the Git technology. If you cannot find relevant material on GitHub as currently implemented, invent new material as appropriate. Try to be as complete as possible, but emphasize the GitHub-specific aspects: particularly, **what security goals are the most relevant for GitHub? What roles does (or, rather, should) GitHub have, and what should each principal be allowed to do?**

Objective

This document aims to spell out the rules of use and operation of GitHub by defining the Roles and permissions of its members and the policies that concern usage, protection, and sharing of its assets.

Definitions

A distributed revision control hosting service powered by Git revision control.
software. It allows collaborative software development, letting a user (or a group of users) in teams) to track, share, and discuss source code hosted in public and private repositories. It is available to its users as a website and as a remote endpoint for Git software.

- Repositories
  A repository can be best understood as a versioned folder for a software project. It contains all the files, documentation, and history pertaining to that project. Repositories can either be private or public. A public repository is open to website visitors, who can be non-registered users. They can view, clone, and pull from it, but must first create a user account before performing any additional actions.

- User
  A user is an individual GitHub account holder and can create repositories, be invited to join an organization, and collaborate with other users via their repositories. A user can be authorized to perform Git actions on a repository, such as pull, push, commit, clone, branch, and merge. Users can either be free or paid; only paid users are allowed to create private repositories.

User Roles and Access Levels

Primary Roles

OWNERS

Users that sign up to use GitHub become the OWNER of the repositories they create. Only paying users are allowed to create private repositories. They have absolute control of their account and agree to the following usage guidelines:
1. Protect personal account credentials: Users must choose a strong account password and must never share it with other users. GitHub provides, and strongly recommends, two-factor authentication so that users can protect their account with more than just a password see §4.
2. Maintain repositories: Each OWNER is responsible for all content they post under their account. They can create and delete repositories, with read/access privileges.
3. Appoint collaborators: The OWNER of a repository can collaborate on a project by assigning read/write privileges to other users, who are known as COLLABORATORS.

COLLABORATORS
If a user is given COLLABORATOR access by the OWNER of a repository, the user has read/write permissions to that repository.

Secondary Roles
A signed-in GitHub user can create an organization account and grant teams-based permissions to the repositories created under that account. Users can also be added to the OWNERS or ADMIN team of that account. A user can be a member of an organization at one of four levels in each team, as described below. Note that each level includes the permissions of all the levels below it.

OWNERS
The users appointed as OWNERS of an account have full access to all repositories and teams within it. They control the account profile and settings, billing and payment information, and members. They can delete the account and all its content.

ADMIN Team Members

A user in an ADMIN team can:
1. Create repositories within the team. Delete repositories assigned to the team.
2. Change team settings
3. Transfer repositories to the organization account
4. Add and remove users from the team. Add collaborators to the repositories assigned to the team.

WRITE Access Team Members

A user in a WRITE access team can, for the repositories assigned to the team:
1. Push to the repository.
2. Apply labels and milestones. Assign, close, and re-open issues.
3. Edit commits, pull requests, and issue comments.
4. Merge and close pull requests.

READ Access Team Members
A user in a READ access team can, for the repositories assigned to the team:
1. Pull from the repository.
2. Fork the repository. Send pull requests from forks.
3. Open issues.
4. Edit their own commits, pull requests, and issue comments.

Authentication

- The service password-protects user accounts and prevents brute-force attacks via rate limiting.
  The passwords are encrypted and salted.
- GitHub allows users to enable two-factor authentication, which requires the user to enter
- a password as well as a security code sent to the user's mobile phone or generated via a two-factor application.
- Git actions require either HTTPS authentication via GitHub account username and password
  or, for convenience, SSH authentication via keys.
- Finally, the service uses SSL for the transmission of all private data.

Employee Access

- GitHub employees cannot access private repositories unless it is required for user support.
- Employees that do access those repositories must temporarily attach their SSH key to the owner's account.

Information Gathering and Usage

GitHub collects information via these channels:

1. User Account Information: GitHub stores the names and email addresses of all registered users. Additionally, paid users are required to enter billing and payment information, which is securely stored on PCI compliant servers.
2. Cookies: GitHub uses browser cookies to record current session information, but does not use permanent cookies. To protect against unauthorized access to user accounts, users are logged out after a certain period of inactivity.
3. Communication with GitHub: GitHub stores all communication with the service via website visits, website registrations, surveys, and email.

The information collected by the service is used to improve its quality; it is not shared with or sold to third parties except in special circumstances, such as to comply with subpoenas from law enforcement or to respond to user violations of the Terms of Service (ToS). GitHub users retain all rights to their data, including software projects.

Amendments to the Security Policy

So far we have defined a security policy for the main GitHub hosting service, but it is open to amendments depending on the needs of its users. To give an example, here are a couple amendments:
• GitHub Enterprise: GitHub provides an external service known as GitHub Enterprise that allows users to host their repositories on their own infrastructure.3
• Government Users: A user that is a U.S. government agency member and account holder of the service on behalf of an agency must agree to a modified ToS.4 This amendment primarily waives the requirements within the ToS that GitHub be used for private, personal,

or noncommercial purposes. GitHub will look solely upon the agency, and not the individual using the service on behalf of the agency, to enforce the modified ToS.

Extra Credit 2

It is well known that re-using a "one-time pad" can be insecure. This problem explores this issue, with some variations.
In this problem all characters are represented as 8-bit bytes with the usual US-ASCII encoding (e.g.,"A" is encoded as 0x41). The bitwise exclusive-or of two bytes x and y is denoted $x \oplus y$.

Let $M = (m_1, m_2, \ldots, m_n)$ be a message, consisting of a sequence of n message bytes, to be encrypted. Let $P = (p_1, p_2, \ldots, p_n)$ denote a pad, consisting of a corresponding sequence of (randomly chosen) "pad bytes" (key bytes).
In the usual one-time pad, the sequence $C = (c_1, c_2, \ldots, c_n)$ of ciphertext bytes is obtained by xor-ing each message byte with the corresponding pad byte:

$$c_i = m_i \oplus p_i, \text{ for } i = 1 \ldots n .$$

When we talk about more than one message, we will denote the messages as $M_1, M_2, \ldots, M_k$ and the bytes of message $M_j = (m_{j1}, m_{j2}, \ldots, m_{jn})$; we'll also use similar notation for the corresponding ciphertexts.

a) Here are two 8-character English words encrypted with the same "one-time pad".

e9 3a e9 c5 fc 73 55 d5

f4 3a fe c7 e1 68 4a df

What are the words? Describe how you figured out the words.

To decode the ciphertexts, we XORed them together, which gives C1⊕C2 = (M1⊕P)⊕(M2⊕P) = M1 ⊕ M2. Thus, the pad P is irrelevant. Then we found a set of 8-character words from an English dictionary. For each M1 in this set, we constructed M2 by XORing M1 with C1 ⊕ C2, since M1 ⊕ (C1 ⊕ C2) = M1 ⊕ (M1 ⊕ M2) = M2. Lastly, we checked if M2 was also in the set. If both M1 and M2 were in the set (as valid English words), then we outputted them.

```
# Given ciphertexts
C1 = [0xe9, 0x3a, 0xe9, 0xc5, 0xfc, 0x73, 0x55, 0xd5]
C2 = [0xf4, 0x3a, 0xfe, 0xc7, 0xe1, 0x68, 0x4a, 0xdf]

# XOR the ciphertexts to get M1 ⊕ M2
M1_XOR_M2 = [c ^ d for c, d in zip(C1, C2)]

# Load the English dictionary
with open('/usr/share/dict/words', 'r') as words_file:
    words = words_file.read().split()
```

```
    words = set([word for word in words if len(word) == len(M1_XOR_M2)])

# Find valid pairs of words
for word1 in words:
    M1 = [ord(c) for c in word1]
    M2 = [c ^ d for c, d in zip(M1, M1_XOR_M2)]
    word2 = ''.join([chr(c) for c in M2])
    if word2 in words:
        pad = [c ^ d for c, d in zip(M1, C1)]
        print(f'word1 = {word1}, word2 = {word2}, pad = {pad}')
```

Output of running the code:
word1 = networks, word2 = security, pad = [135, 95, 157, 178, 147, 1, 62, 166] word1 = security, word2 = networks, pad = [154, 95, 138, 176, 142, 26, 33, 172]
Though their ordering is indiscernible, the two words are security and networks.

b) Ben Bitdiddle decided to fix this problem by making sure that you can't just "cancel" pad bytes by xor-ing the ciphertext bytes.

In his scheme the key is still as long as the ciphertext. If we define $c_0 = 0$ for notational convenience, then the ciphertext bytes $c_1, c_2, \ldots, c_n$ are obtained as follows:

$$c_i = m_i \oplus ((p_i + c_{i-1}) \bmod 256).$$

That is, each ciphertext byte is added to the next key byte and the addition result (modulo 256) is used to encrypt to the next plaintext byte.

Ben is now confident he can reuse his pad, since $(k_i + c_{i-1}) \bmod 256$ will be different for different messages, so nobody would be able to cancel the k_i's out. You are provided with otp-feedback.py, which contains an implementation of Ben's algorithm.
You are also given the file tenciphs.txt, containing ten ciphertexts $C_1, C_2, \ldots, C_{10}$ produced by Ben, using the same pad P. You know that these messages contain valid English text.
Submit the messages and the pad, along with a careful explanation of how you found them, and any code you used to help find the messages. The most important part is the explanation.

Messages and Pad

We stand today on the brink of a revolution in cryptography.
Probabilistic encryption is the use of randomness in an encr
Secure Sockets Layer (SSL), are cryptographic protocols that
This document will detail a vulnerability in the ssh cryptog
MIT developed Kerberos to protect network services provided
NIST announced a competition to develop a new cryptographic
Diffie-Hellman establishes a shared secret that can be used
Public-key cryptography refers to a cryptographic system req
The keys used to sign the certificates had been stolen from
We hope this inspires others to work in this fascinating fie

pad = [119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71, 101, 71, 88, 116, 78, 113,

102, 113, 87, 84, 65, 51, 55, 99, 56, 107, 69, 116, 105, 110, 109, 97, 113, 79, 106, 122, 68, 66, 98, 77, 72, 112, 72, 55, 53, 104, 54, 99, 71, 87, 97, 68, 98, 112, 49]

This part of the code was more interactive since, because of Ben's addition of feedback, we couldn't just XOR the 10 ciphertexts together and look up possible messages in the dictionary. Our plan of attack was to first calculate all the possible pad bytes, pi's) that would yield the valid and likely English characters, that is, letters and common punctuation, from all 10 ciphertexts.

```
# Define valid characters (A-Z, a-z, space, common punctuation)
valid_chars = set(range(65, 65+26) + range(97, 97+26) + [32, 44, 46, 63, 33, 45,
40, 41])
```

Let's consider one specific index i in the 60-character messages, ($0 \leq i < 60$). For our purposes pi is independent of pad bytes and depends only on mi, ci and c_{i-1} for it is included in the following calculate pad function. We could try all 28 possible pi's. However, |valid_chars| is just 60. So we take each valid character, calculate which pi would give that character in the first ciphertext and then check if it results to valid characters for the other 9 ciphertexts.

```
# Calculate the pad
def calculate_pad(ctext, msg, prev_c=0):
    assert len(ctext) == len(msg)
    pad = []
    for i in range(len(ctext)):
        p = ((ctext[i] ^ msg[i]) - prev_c) % 256
        pad.append(p)
        prev_c = ctext[i]
    return pad

def prev_c_at(ciph, index):
    return 0 if index == 0 else ciph[index - 1]

def ctext_at(ciph, index):
    return ciph[index:index + 1]

msglen = 60
possible_pad_bytes = [[] for _ in range(msglen)]
for index in range(msglen):
    for c in valid_chars:
        possible_pad_byte = calculate_pad(ctext_at(tenciphs[0], index), [c],
prev_c=prev_c_at(tenciphs[0], index))
        is_valid = True
        for ciph in tenciphs:
            msg = ben_decrypt(ctext_at(ciph, index), possible_pad_byte,
prev_c=prev_c_at(ciph, index))
            if not set(msg).issubset(valid_chars):
                is_valid = False
                break
        if is_valid:
            possible_pad_bytes[index].append(possible_pad_byte[0])
```

This calculated all possible pi ’s at each i, which is a good start since some choice of P = p1, . . . , p60 within these pi ’s would result in Ben’s messages. However, here’s the issue: >>> [len (p) for p in possible_pad_bytes] [20 , 6 , 2 , 1 , 2 , 1 , 1 , 1 , 1 , 5 , 5 , 4 , 1 , 1 , 1 , 8 , 3 , 2 , 2 , 2 , 1 , 1 , 1 , 3 , 3 , 1 , 1 , 1 , 1 , 5 , 4 , 1 , 1 , 2 , 1 , 2 , 2 , 9 , 1 , 4 , 2 , 1 , 1 , 2 , 1 , 1 , 1 , 1 , 1 , 2 , 2 , 1 , 2 , 2 , 1 , 1 , 1 , 3 , 2 , 1]

While we could scan the plaintext manually, we preferred to have the computer do it and score its intelligibility. So, we loaded an English dictionary (the Ubuntu dictionary is excellent; it even contains Hellman). For each set of 10 plaintext messages, we checked how many valid English words from the dictionary appeared in it and gave it |word| 2 points for each word that did. This scoring function strongly favors longer words like cryptography. Lastly, we outputted the best-scoring messages and pad.

```
# Recursively expand the pad
def recursively_expand_pad(cur_index, cur_pad, words):
    if cur_index == msglen:  # msglen starts as 9 instead of 60
        texts = [bytes_to_text(ben_decrypt(ciph[:msglen], cur_pad)) for ciph in
tenciphs]
        text_to_score = '\n'.join(texts).lower()
        score = sum(len(word)**2 for word in words if word in text_to_score)
        return (score, texts, cur_pad)
    else:
        best_score = 0
        best_texts = None
        best_pad = None
        for p in possible_pad_bytes[cur_index]:
            score, texts, pad = recursively_expand_pad(cur_index + 1, cur_pad +
[p], words)
            if best_score < score:
                best_score = score
                best_texts = texts
                best_pad = pad
        return (best_score, best_texts, best_pad)

# Load the English dictionary
with open('/usr/share/dict/words', 'r') as words_file:
    words = set([word.lower() for word in words_file.read().split()])

# Start the recursive expansion
_, texts, pad = recursively_expand_pad(0, [], words)
print(f'messages = {texts}, pad = {pad}')

# Output example:
# messages = ['We stand ', 'Probabili', 'Secure So', 'This docu', 'MIT devel',
'NIST anno', 'Diffie-He', 'Public-ke', 'The keys ', 'We hope t'],
# pad = [119, 75, 116, 51, 85, 113, 72, 105, 76]

# Continue expanding to decrypt the full 60 characters
```

We could have searched these plaintext prefixes on Google, but we decided to continue running our code to choose p9, . . . , p18 (often the messages we want to decrypt won't be available online). First we wrote the pad above to possible pad bytes[0:9] (so there is only 1 choice for pi , $0 \leq i < 9$), and then we increased msglen to 18. Output of rerunning the code:

messages = ['We stand today on ', 'Probabilistic encr', 'Secure Sockets Lay', 'This document will', 'MIT developed Kerb', 'NIST announced a c', 'Diffie-Hellman est', 'Public-key cryptog', 'The keys used to s', 'We hope this inspi'], pad = [119, 75, 116, 51, 85, 113, 72, 105, 76, 78, 114, 79, 84, 49, 71, 101, 71, 88]

Repeating this process 4×, we eventually got the desired output (pasted at the beginning) for all 60 characters. At this point, we manually checked it with online articles to be confident in our decryption. For instance, We stand today. . . appears in a well-cited 1976 paper by Whitfield Diffie and Martin Hellman. 7

Extra Credit 3

In the previous problem, we saw how to attack a scheme in which a one-time pad, or a scheme like it, is reused. This problem will walk you through how the rare reuse of the pad for the standard one-time pad (not the scheme of the previous part) can be efficiently detected. We will look at the extreme case in which a pad is reused just once.

The following fact may be useful in this problem: Let $R_p(n)$ be the longest run of heads in n coin flips, each of which is heads with probability p. With high probability as n grows, $R_p(n)$ is between $\log_{1/p} n - \log_{1/p} \ln \ln n$ and $\log_{1/p} n + \log_{1/p} \ln n$.

a) Suppose you are given poly(n) n-bit ciphertexts $c_1, \ldots, c_q$, each of which is properly encrypted with an independent uniformly randomly chosen one-time pad. Show that with high probability, the length of the longest repeated bitstring (i.e. a substring of both c_i and c_j for some $i \neq j$) is less than $\log_2 n + \log_2 \ln n$.

the goal is to determine the upper bound on the longest run of matching bits between any two n-bit ciphertexts encrypted with unique plaintext/pad pairs. The analysis shows that, with high probability, this upper bound is log 2 (nlogn).

First, the longest run of matching bits between two ciphertexts without any offset is modeled using the probability of matching bits, which is 0.5. It is shown that the longest run is less than log 2(nlogn) with high probability.

Next, considering offsets, the upper bound for matching bits within n-k bits of two ciphertexts remains log 2(nlogn). This extends to all possible pairs and offsets, amounting to a polynomial number of pairs.

Combining these results, the analysis concludes that for f(n)=poly(n) ciphertexts, the upper bound on the longest run of matching bits is log2 (nlogn) with high probability for all pairs and offsets.

b) We have collected data on the average length of the longest run of identical characters in identical positions in passages of English text by randomly sampling the collected works of the American writer Winston Churchill. You can find a graph of run length (in characters) versus log of the message length in characters at https://courses.csail.mit.edu/6.857/2014/files/run lengths.png. How do these run-lengths

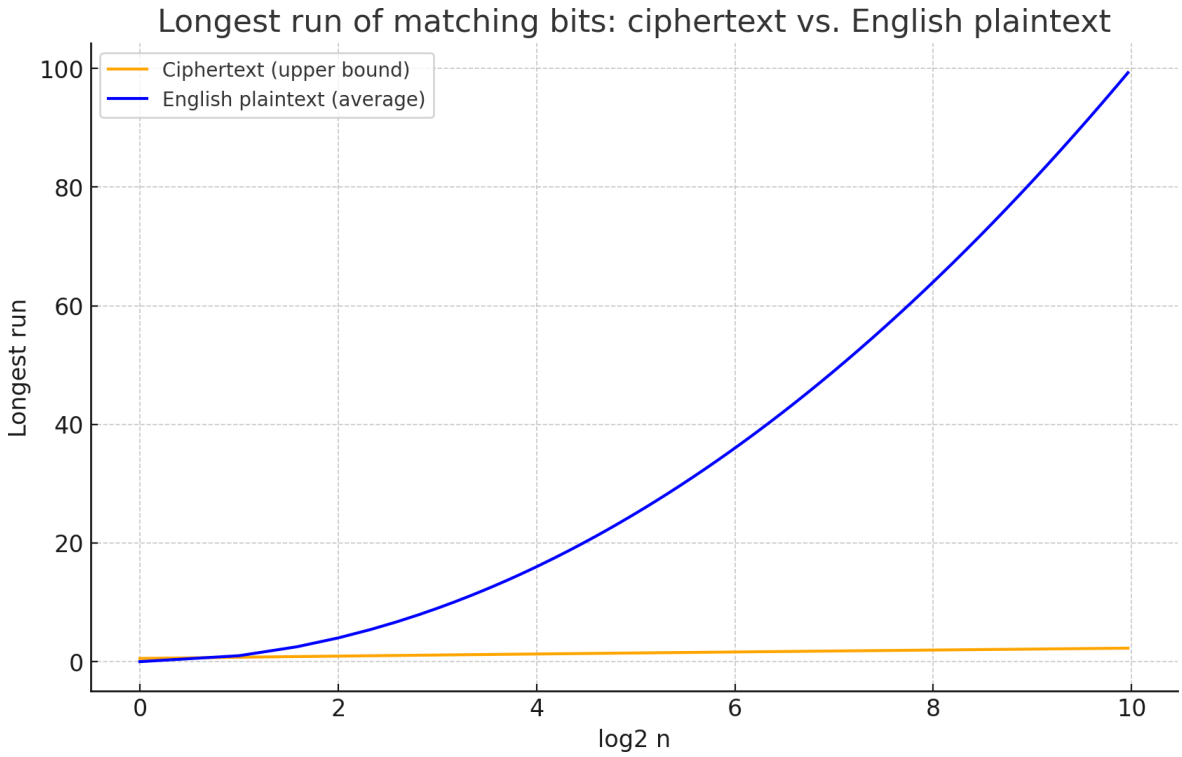
compare to the upper bound of the previous part on the longest repeated bit string if the English plaintext is US-ASCII encoded?

To begin, the upper bound from part (a) is plotted and superimposed on a graph that shows the longest runs for English plaintext. Here, n represents the text length in characters, and the upper bound of the longest run in ciphertext is calculated as $1/7 \log_2(7n\log 7n)$.

The analysis highlights that at the beginning, the random ciphertext has more flexibility in matching bits than the structured 7-bit US-ASCII characters of English plaintext. This is due to several factors: bits can match across character boundaries, there are more characters to match in the plaintext, and the ciphertext bound is an upper limit while the plaintext is an average.

However, as the text length increases ($\log_2 n > 9$), the average longest runs in English plaintext start to exceed the upper bound for ciphertext. This occurs because the English language has inherent redundancy, such as high frequencies of certain letters and common letter pairings. This redundancy leads to longer runs of matching bits in plaintext compared to ciphertext.

The graph illustrates this difference, where the ciphertext upper bound appears linear while the plaintext average appears exponential. Thus, for sufficiently large text lengths, the English plaintext's average longest runs are larger than the upper bound on the longest runs in ciphertext.



c) Assume that each pair of plaintexts shares a long common run of identical characters, and no pair of ciphertexts with independently chosen pads does. Show that given poly(n) n-bit ciphertexts with total length N and one instance of pad reuse, you can find the two ciphertexts that share a key in time which is $O(N)$ in the total ciphertext length. This means that your solution should run in $O(N \log N)$ time for some constant c.

Hint: You may wish to read about and use suffix trees in your solution. They are a type of tree that stores all suffixes of a string of n characters, can be constructed in $O(n \log n)$ time and can be used, among other things, to find the longest repeated substring of a string in linearithmic time by finding the deepest non-leaf node. Note that the longest repeated substring is not exactly what you need to solve this problem. If you are not familiar with suffix trees, you may or may not find suffix arrays simpler and easier to understand.

the task is to detect pad reuse between pairs of n-bit ciphertexts in quasilinear time, where
f(n)=poly(n). If two ciphertexts share a substring longer than $\log_2(n\log n)$, it indicates pad reuse.

Algorithm
Concatenate Ciphertexts: Concatenate the ciphertexts with unique terminators into a single string S.

Suffix Tree Construction: Build a suffix tree for S.
Longest Repeated Substring: Traverse the suffix tree to find the deepest internal node, representing the longest repeated substring.

Detect Pad Reuse: If the length of this substring is greater than $\log_2$ (nlogn), trace paths from the node to find the corresponding ciphertexts ci and cj that reuse the same pad.

Correctness and Efficiency
The algorithm correctly identifies pad reuse based on substring length, leveraging suffix trees for efficient traversal. The total running time is quasilinear, O(NlogN), where $N=n \cdot f(n)$.