COMBINATION OF CIPHER BLOCK CHAIN AND FEISTEL CIPHERS IN HMAC CONSTRUCTION

Alana Jocelyn Natania Massie	(22/496239/PA/21331)
Khalisha Fadiya Khansa	(22/496155/PA/21313)
Vian Sebastian Bromokusumo	(22/496698/PA/21355)

DEPARTMENT OF COMPUTER SCIENCE AND ELECTRONICS
FACULTY OF MATHEMATICS AND NATURAL SCIENCE
GADJAH MADA UNIVERSITY
YOGYAKARTA

2024

CHAPTER I
INTRODUCTION

1.1 Introduction

A hash function is a function that accepts an input (or ‘message’) and produces a fixed-size string of bytes. The resulting output, usually a number, is referred to as the hash code or hash value. The primary objective of a hash function is to efficiently map data of variable sized to fixed-size values, which are frequently utilized as indexes in hash tables [1]. A hash function, depending on the algorithm, typically needs to divide the input data into fixed-sized blocks, known as data blocks, to produce a hash value of a set length. This is because hash functions process data in fixed-length segments, and the size of these data blocks varies across different algorithms. If the blocks are not sufficiently large, padding may be added to complete them. However, regardless of the hashing method used, the output or hash value always maintains a consistent fixed length. The hash function is applied repeatedly, once for each data block [2].

Hash-based Message Authentication Code (HMAC) is a method of message encryption that combines a cryptographic key with a hash function. This approach gives both the server and the client a private key that is exclusively known to them, offering a more secure way of encrypting data compared to a basic Message Authentication Code (MAC) [3]. The HMAC process involves two main stages: the inner hashing and the outer hashing. The inner hash combines the secret key with inner padding and the message, while the outer hash combines the secret key with outer padding and the inner hash result. This double application of the hash function provides a high level of security, ensuring that any changes to the message or the key will result in a different HMAC value.

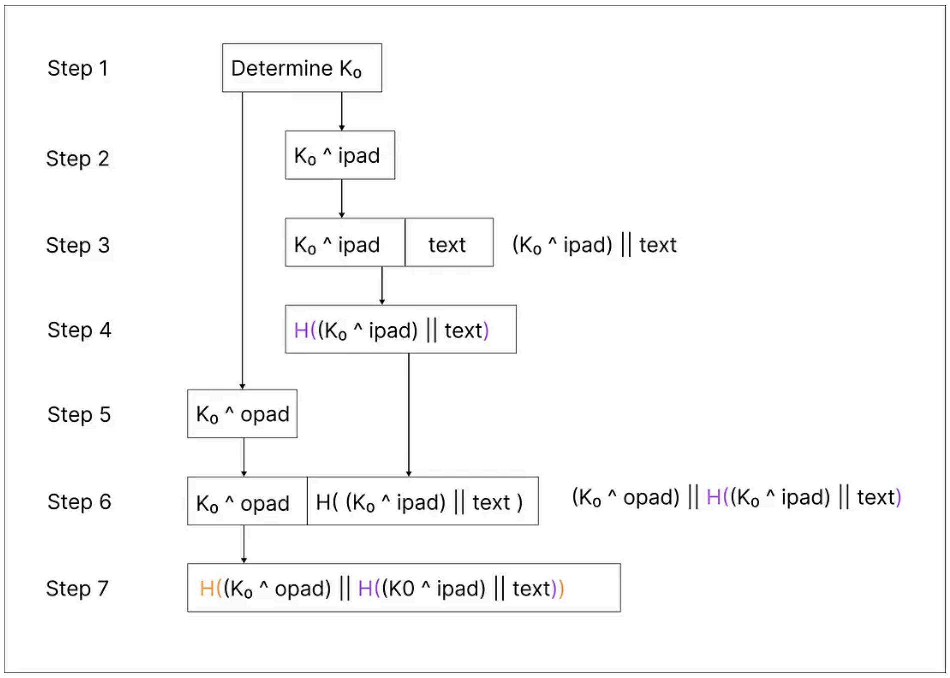

Fig. 1. HMAC Diagram [4]

A collision occurs when two or more files, despite having different contents and behaviors, produce the same value. Following the discovery of an MD5 collision by Wang et al. [5], cryptanalysts have increasingly sought to find additional collisions in both MD5 and SHA hashes, striving for greater efficiency in their methods. The Brute Force algorithm can be used to find a collision, however it is challenging to do computationally [6].

In this report, we aim to adapt the algorithm from the previous assignment to create a 16-bit hash function that can be used in HMAC (Hash-based Message Authentication Code). This involves ensuring the hash function's process is 16 bits and integrating it with the HMAC construction. Subsequently, we will identify two distinct messages that produce the same hash value, and two other messages that result in the same HMAC.

1.2 Objective

The objective of this report is to document the following:

1. Modifications to the Previous Algorithm: Report the changes made to convert the existing algorithm into a 16-bit hash function suitable for HMAC.
2. Identical Hash Values: Identify the two different messages (M1 and M2) that produce the same hash value and elaborate the process used to find them.

3. Identical HMAC Values: Find two different messages (M3 and M4) that generate the same HMAC and explain the method used to discover them.
4. Source Code: Provide the source codes containing the modified 16-bit hash function and the HMAC implementation.
5. Results and Discussion: Present the findings and analysis of the identified messages and their hash or HMAC values, along with comparisons between the previous assignment and the modified algorithm.

CHAPTER II
IMPLEMENTATION

2.1 Explanation

The first objective is to modify the previous algorithm to convert it into a 16-bit hash function suitable for HMAC. Some of the points from the previous algorithm that have been implemented in this algorithm are a Feistel structure with 16 rounds and CBC mode. Below is a step-by-step diagram of the implementation:

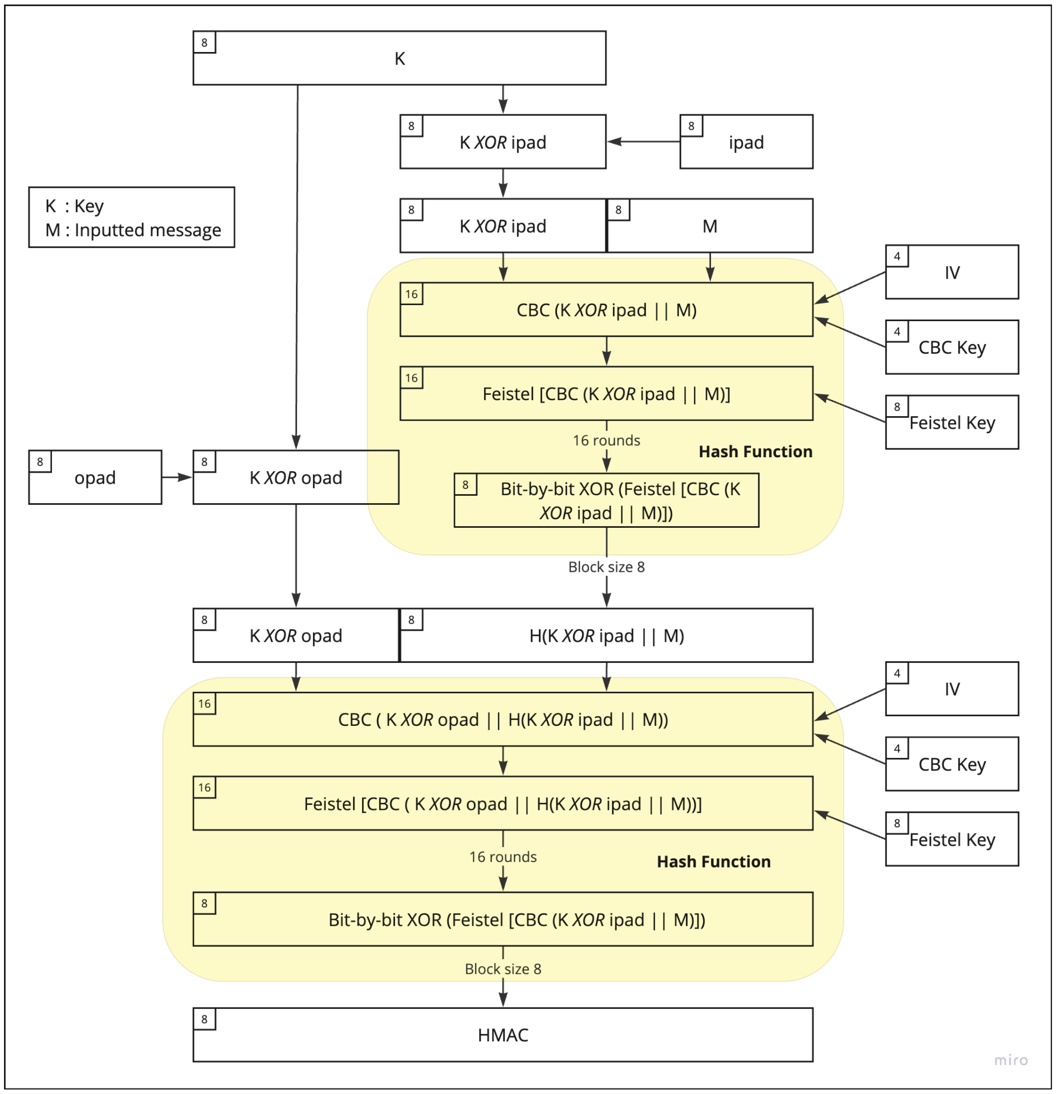

Fig. 2. Modified Algorithm Diagram

1. Input: 8-bit message.
2. Key XOR with ipad: The key is XORed with ipad (both 8 bits). This results in a 8-bit intermediate value.
3. Concatenation with Message: The intermediate value from the previous step is concatenated with the 8-bit input message. This results in a 16-bit value.
4. Initial Hash Function (CBC, Feistel Cipher, and Bit-by-bit XOR):
    - CBC Mode:
        - An Initialization Vector (IV) and a CBC key, both 4 bits, are used.
        - The 16-bit value is processed using CBC mode, producing a 16-bit output.
    - Feistel Cipher:
        - The 16-bit output from CBC mode is then processed using a Feistel cipher.
        - A Feistel key of 8 bits is used.
        - The Feistel cipher runs for 16 rounds, resulting in a 16-bit output.
    - Bit-by-bit XOR:
        - To maintain a constant 16-bit hash function, the final result is XORed with a block size of 8.
        - This results in a final 8-bit output.
5. Key XOR with opad:
    - The original key is XORed with opad (both are 8 bits).
    - This results in another 8-bit intermediate value.
6. Second Concatenation:
    - The 8-bit result from the first Hash Function is concatenated with the 8-bit result from the key XOR opad operation.
    - This creates a new 16-bit value.
7. Final Hash Function (CBC, Feistel Cipher, and Bit-by-bit XOR): The final result is the 8-bit HMAC output.

2.2 Source Code [7]

The source code of our modification is as follows:

2.2.1 Modified Hash Function

```
def xor(a, b):
    return ''.join(str(int(x) ^ int(y)) for x, y in zip(a, b))
```

Code snippet to perform XOR operation on two binary strings of equal length.

```
def cbc_encrypt_4bit(key, iv, bit_input):
    assert len(key) == 4, "Key must be 4 bits."
    assert len(iv) == 4, "IV must be 4 bits."
    assert len(bit_input) == 16, "Input must be 16 bits."

    parts = [bit_input[i:i+4] for i in range(0, len(bit_input), 4)]

    encrypted_parts = []
    current_iv = iv

    for part in parts:
        xor_with_iv = xor(part, current_iv)

        encrypted_part = xor(xor_with_iv, key)

        encrypted_parts.append(encrypted_part)

        current_iv = encrypted_part

    encrypted_message = ''.join(encrypted_parts)
    return encrypted_message

encrypted_message = cbc_encrypt_4bit(key, iv, bit_input)

print(f'Original bit input: {bit_input}')
print(f'Encrypted message: {encrypted_message}')
```

Code snippet to encrypt a 16-bit binary input using a 4-bit key and 4-bit IV in CBC mode.

```
def feistel_round(L, R, key):
    new_L = R
    new_R = format(int(L, 2) ^ (int(R, 2) ^ int(key, 2)), '08b')
    return new_L, new_R

def feistel_block(bit_input, key, rounds=16):
    if len(bit_input) != 16:
        raise ValueError("Input bitstring must be exactly 16 bits")

    L = bit_input[:8]
    R = bit_input[8:]
    if len(key) != 8:
        raise ValueError("Key bitstring must be exactly 8 bits")

    for i in range(rounds):
        L, R = feistel_round(L, R, key)

    output_bitstring = L + R
    return output_bitstring
```

Code snippet for XOR operation for the round function, with bit strings and for Feistel block encryption with 16-bit input in bitstring form and 16-bit output.

```
def xor_8bit(input_16bit):
    assert len(input_16bit) == 16, "Input must be 16 bits."

    part1 = input_16bit[:8]
    part2 = input_16bit[8:]

    xor_result = ''.join(str(int(x) ^ int(y)) for x, y in zip(part1, part2))

    return xor_result

xor_result = xor_8bit(input_16bit)

print(f'Original 16-bit input: {input_16bit}')
print(f'XOR result of the two 8-bit parts: {xor_result}')
```

Code to perform Bit-by-bit XOR on the two 8-bit parts of a 16-bit input from the Feistel cipher.

```
def hash_encrypt(key, iv, message, feistel_key):

    cbc_result = cbc_encrypt_4bit(key, iv, message)
    feistel_result = feistel_block(cbc_result, feistel_key, rounds=16)
    xor_res = xor_8bit(feistel_result)

    return xor_res
```

Code to perform the Hash Function, combining all the functions from CBC, Feistel, and Bit-by-bit XOR.

```
def hmac_custom(message, hash_key, key, feistel_key, iv):
    ipad = '00110110'
    opad = '01011100'

    inside_hash_bin = xor(hash_key, ipad)
    outside_hash_bin = xor(hash_key, opad)
    concat_message_inside_hash = message + inside_hash_bin

    if len(concat_message_inside_hash) < 16:
        concat_message_inside_hash = concat_message_inside_hash.ljust(16, '0')

    xor_res = hash_encrypt(key, iv, concat_message_inside_hash, feistel_key)

    concat_feistel_outside_hash = xor_res + outside_hash_bin
    if len(concat_feistel_outside_hash) < 16:
        concat_feistel_outside_hash = concat_feistel_outside_hash.ljust(16, '0')

    final_xor_res = hash_encrypt(key, iv, concat_feistel_outside_hash, feistel_key)

    return final_xor_res
```

Code to perform the algorithm and acquire the HMAC value.

2.2.2 Hash Collision

2.2.2.1 Hash Value after the first Hash Function in the HMAC Algorithm

```
import itertools

def find_collision(hash_key, key, iv, feistel_key, func):
    messages = [''.join(seq) for seq in itertools.product('01', repeat=8)]
    results = {}

    for message in messages:
        result = func(message, hash_key, key, feistel_key, iv)
        if result in results:
            return results[result], message
        results[result] = message

    return None, None

def hash_block_1(message, hash_key, key, feistel_key, iv):
    ipad = '00110110'

    inside_hash_bin = xor(hash_key, ipad)
    concat_message_inside_hash = message + inside_hash_bin

    if len(concat_message_inside_hash) < 16:
        concat_message_inside_hash = concat_message_inside_hash.ljust(16, '0')

    xor_res = hash_encrypt(key, iv, concat_message_inside_hash, feistel_key)
    return xor_res

feistel_key = '01100110'
hash_key = '11001100'
key = '1100'
iv = '1100'

msg1, msg2 = find_collision(hash_key, key, iv, feistel_key, hash_block_1)

if msg1 and msg2:
    print(f'Found collision:\nMessage 1: {msg1}\nMessage 2: {msg2}\nXOR result:
{hash_block_1(msg1, hash_key, key, feistel_key, iv)}')
else:
    print('No collision found.')
```

2.2.2.2 Hash Value outside of the HMAC Algorithm

```
def find_collision_trunc(key, iv, feistel_key):
    messages = [''.join(seq) for seq in itertools.product('01', repeat=16)]
    results = {}

    for message in messages:
        result = hash_encrypt(key, iv, message, feistel_key)
        if result in results:
            return results[result], message
        results[result] = message

    return None, None

msg1, msg2 = find_collision_trunc(key, iv, feistel_key)

if msg1 and msg2:
    print(f'Found collision:\nMessage 1: {msg1}\nMessage 2: {msg2}\nXOR result:
{hash_encrypt(key, iv, msg1, feistel_key)}')
else:
    print('No collision found.')
```

2.2.3 HMAC Collision

```
def find_collision(hash_key, key, iv, feistel_key, func):
    messages = [''.join(seq) for seq in itertools.product('01', repeat=8)]
    results = {}
    for message in messages:
        result = func(message, hash_key, key, feistel_key, iv)
        if result in results:
            return results[result], message
        results[result] = message

    return None, None

msg1, msg2 = find_collision(hash_key, key, iv, feistel_key, hmac_custom)

if msg1 and msg2:
    print(f'Found collision:\nMessage 1: {msg1}\nMessage 2: {msg2}\nXOR result:
{hmac_custom(msg1, hash_key, key, feistel_key, iv)}')
else:
    print('No collision found.')
```

CHAPTER III
RESULTS AND DISCUSSION

3.1 Output

3.1.1 HMAC

Example Input:

```
message = '11001100'
feistel_key = '01100110'
hash_key = '11001100'
key = '1100'
iv = '1100'
```

Output:

```
HMAC Result (8 bits): 11001010
```

3.1.2 Hash Collision

3.1.2.1 Hash Collision Hash Value after the first Hash Function in the HMAC Algorithm

Output:

```
No collision found.
```

3.1.2.1 Hash Value outside of the HMAC Algorithm

Output:

```
Found collision:
Message 1: 0000000000000000
Message 2: 0000000000000001
XOR result: 01101010
```

3.1.3 HMAC Collision

Output:

```
No collision found.
```

3.2 Comparison with Previous Assignment

The following are the comparisons between the previous assignment and the modified algorithm.

	Assignment 3	Assignment 4
Plaintext length	Uses CBC if longer than 32-bits.	The length must be 8 bits.

CBC	Manual.	Manual.
Feistel	Output was generated and displayed for each round along with the process.	No outputs are generated each round.
Subkey	Subkeys were used in each round.	No subkeys.
Bit-by-bit XOR	No bit-by-bit XOR.	Implements bit-by-bit XOR after each Feistel round.
Algorithm	Made to encrypt, using Feistel or CBC.	Made to create HMAC value, using both Feistel and CBC with an extra step of bit-by-bit XOR.

3.3 Discussion

The outcomes from our implementation of the HMAC algorithm and hash collision checks reveal several important aspects of our custom Hash Function and its integration with the HMAC process. The successful computation of the HMAC value demonstrates that our custom Hash Function, derived from the ciphers developed in the previous assignment, works effectively within the HMAC framework.

Creating the hash function posed significant challenges, primarily because the objective was to develop a 16-bit hash. This challenge was intensified by the requirements of the HMAC algorithm, which involves appending various components during its process. Maintaining the hash function at a strict 16-bit length became problematic under these conditions. To resolve this issue, a bit-by-bit XOR operation was implemented, enabling the hash to stay within the desired 16-bit limit while still incorporating the necessary components within the HMAC structure.

The absence of hash collisions after the first hash function within the HMAC algorithm indicates that our custom Hash Function maintains a sufficient level of uniqueness when used with HMAC. This is a positive indicator of the algorithm's security, as it suggests that the custom hash function can effectively differentiate between different inputs within the HMAC structure. The detection of a hash collision outside the HMAC algorithm highlights a critical observation about our custom Hash Function: while it performs adequately within the HMAC context, it may show vulnerabilities when used independently. This collision reveals that the custom hash function can produce identical outputs for different inputs, indicating a potential weakness in its design. This insight is crucial for understanding the limitations of our hash function and emphasizes the need for further improvement to enhance its collision resistance and overall security.

Although a hash value collision is detected, the absence of collisions in the HMAC output shows the strength of the HMAC mechanism in ensuring collision resistance. This result shows that the HMAC process effectively reduces the risk of collisions, even if the underlying hash function has some weaknesses. This algorithm’s capability to generate unique and secure outputs for different inputs confirms its effectiveness as a reliable tool for message authentication, enhancing the security of our custom hash function.

CHAPTER IV
CONCLUSION

In this report, we successfully modified the algorithm from the previous assignment and transformed it into a 16-bit hash function suitable for use in HMAC. This transformation requires significant adjustments to ensure the hash function's output was constantly 8 bits. The lack of hash collisions after the first hash function within HMAC shows that our modified algorithm maintains sufficient uniqueness when used with HMAC, indicating it is secure and can distinguish between different inputs. Additionally, the lack of collisions in the HMAC output highlights HMAC's effectiveness in ensuring collision resistance.

However, detecting a hash collision outside the HMAC algorithm reveals that our algorithm may have some weaknesses when used independently, emphasizing the need for further improvements.

REFERENCES

[1] Khinchi, R. (2024). Hash Functions and Types of Hash Functions. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/

[2] Loo, A. Hash Function. Corporate Finance Institute. Retrieved from https://corporatefinanceinstitute.com/resources/cryptocurrency/hash-function/#:~=by%20Andrew%20Loo-,What%20is%20a%20Hash%20Function%3F,or%20simply%2C%20a%20hash

[3] Awati, R. Hash-based Message Authentication Code (HMAC). TechTarget. Retrieved from https://www.techtarget.com/searchsecurity/definition/Hash-based-Message-Authentication-Code-HMAC

[4] Volkov, O. How HMAC Works, Step-by-Step Explanation with Examples. Medium. Retrieved from https://medium.com/@short_sparrow/how-hmac-works-step-by-step-explanation-with-examples-f4aff5efb40e

[5] Wang X, Feng D, Lai X, Yu H. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Int Assoc Cryptologic Res. 2004;5:5–8.

[6] Rasjid, Z. E., Soewito, B., Witjaksono, G., & Abdurachman, E. "A Review of Collisions in Cryptographic Hash Functions Used in Digital Forensic Tools." 2nd International Conference on Computer Science and Computational Intelligence 2017 (ICCSCI 2017), 13-14 October 2017, Bali, Indonesia. Bina Nusantara University.

[7] Modified_HMAC_Final. Google Colaboratory Jupyter File. https://colab.research.google.com/drive/1F5byANyX4w5VFyKO0QtHrSafIrae_yIw?usp=sharing