Padding-Oracle Attack Report

Assignment: Breaking A CBC Block Cipher Mode

SUNWOO PARK (Student ID: 2021428629)
Department of Computer Science Engineering, Kyungpook National University

November 12, 2025

Contents

1 Introduction

In this work, we implement and evaluate a padding-oracle attack against a block cipher in Cipher-Block-Chaining (CBC) mode. Our objective is to recover the original plaintext from each given ciphertext-block pair without any knowledge of the encryption key. To accomplish this, we leverage an oracle that reveals whether the decrypted ciphertext has valid padding.

The remainder of this report is structured as follows. First, we provide the theoretical background on CBC mode and padding mechanisms. Then, we outline the specific attack algorithm and its implementation in Python. We then present the decryption results and their ASCII representations for multiple ciphertext pairs. Finally, we conclude by discussing encountered challenges, performance metrics, and key lessons learned.

2 Background

2.1 CBC Mode and Padding

In cipher-block chaining (CBC) mode, each plaintext block P_i is first XORed with the previous ciphertext block C_{i-1} (or with the initialization vector IV when $i = 0$), and then encrypted using the key K. This process can be expressed as:

$$C_i = E_K\big(P_i \oplus C_{i-1}\big), \quad C_0 = IV.$$

Decryption simply reverses this operation:

$$P_i = D_K(C_i) \oplus C_{i-1}, \quad C_0 = IV.$$

Because block ciphers operate on fixed-length blocks (for example, 8 bytes in our assignment), if the final block of plaintext is shorter than one full block, we must apply padding. A common scheme is PKCS #7 (or PKCS #5), in which k bytes of value k are appended. For example, if 4 bytes of padding are needed, the padding would be `0x04 0x04 0x04 0x04`.

What makes this especially relevant in a padding-oracle attack is that the padding follows a predictable structure. An adversary can modify a ciphertext block and query an oracle that answers whether the decrypted plaintext ends with valid padding. By gradually exploiting the “valid/invalid” padding responses, the attacker can recover the original plaintext—**without ever knowing the encryption key**.

2.2 Padding-Oracle Attack

Suppose we got two ciphertext blocks (C_{i-1}, C_i). And we want to figure out the plaintext blocks P_i. I don’t know the secret key, but there’s something powerful about it. It’s the oracle that tells you if a decrypted message ends in a valid padding.

1. Create a modified version of the first block, name it C'_{i-1}, and submit (C'_{i-1}, C_i) to Oracle.

2. The oracle decrypts C_i, XORs the result with C'_{i-1}, verifies that padding is valid, and simply responds with “Yes” or “No.” A “Yes” means that the last byte of the decrypted block matches a valid padding value such as `0x01` or `0x02 0x02`. This gives away a clue about the median value $D_K(C_i)$, because:

$$P_i = D_K(C_i) \oplus C_{i-1}$$

3. Repeat this process one byte at a time from the end of the block and moving backwards—you can gradually recover the entire $D_K(C_i)$. Knowing that, you then compute:

$$P_i = D_K(C_i) \oplus C_{i-1}.$$

Since padding always follows a known, fixed pattern, each oracle response reduces the uncertainty by one byte at a time. In practice, recovering entire b-byte blocks often takes $256 \times b$ queries, much faster than attacking keys with brute force attacks.

A variant of this method is applied in an assignment scenario using an 8-byte block size. To address problems in real-world environments, we introduce response normalization to interpret non-standard oracle responses and backtracking logic to recover from ambiguous cases, increasing the robustness and reliability of our attacks.

3 Environment & Methodology

3.1 Environment Setup

- Language: Python 3.13
- Provided oracle function: `pad_oracle(C0_hex, C1_hex)`
- Block size: 8 bytes (64 bits)

3.2 Attack Algorithm Overview

Explain the step-by-step algorithm:

1. Start from the last byte of the plaintext block.
2. Modify bytes of the previous block to force padding values.
3. Use oracle responses to deduce intermediate value and plaintext.
4. Repeat for each byte from right to left.

4 Implementation

Implementations have chosen Python (version 3.13) as the programming language. Below, we outline the main components of the script, discuss important features, and highlight adjustments made for its robustness.

4.1 Key Components

- `normalize_oracle_response(ok)` — In our step, Oracle always returned `b'1'` or `b'0'` (in the form `byte`), not the regular integer `1` or string `"1"`. Because of this, we included a dedicated regularization function normalize_oracle_response() that handles all these variations and allows consistent interpretation of Oracle's output as an integer `1` (for effective padding) or `0` (for invalid padding) Without this step, subtle differences in return types can lead to false padding valid detection and disruptions to the plain text recovery process.

- Processing hexadecimal strings — Functions such as `hex_to_bytes_safe(s)` remove the "0x" prefix, prepend '0' to handle odd-length hexadecimal strings, and then convert the result to `bytes` objects. Verifying that ciphertext and IV blocks are of correct length (8 bytes) prevents incorrect input.

- `recover_block(C0_bytes, C1_bytes)` — Implement a core padding-oracle attack with back-tracking, as follows:

  - Initializes the arrays for the median value and recovered plaintext.
  - Defines a helper `build_prefix_for_index(idx, pad_len)` to create a modified version of the previous ciphertext block so that the suffix bytes mimics the padding of length `pad_len`.
  - Uses a depth-first search (DFS) starting from the last byte to collect all candidate intermediate bytes for a given index. If the candidate does not find a complete solution, it backtracks to try an alternative candidate.

- `attack(C0_hex, C1_hex)` — Parses the input hex strings, verify the accuracy of the block length, use `remove_pkcs7_padding()` to remove PKCS padding, and outputs the recovered plaintext (to hex and ASCII if printable).

4.2 Key Adjustments for Robustness

- **Response Normalization**: In an actual integrations (or oracle of this assignment), the return value can be `b'1'`, `'1'`, or even `True`. The `normalize_oracle_response()` function integrates all cases into one integer 1 or 0.

- **Back-tracking Logic**: Some ciphertext modifications can accidentally generate valid padding early (false), especially if the original block had valid padding. DFS with back-tracking addresses this problem by listing all valid candidates and committing only to the path that reaches the full solution.

- **Hex Handling Safeguards**: Input hex strings are blanked, odd length corrected, and block-size enforced. This avoids runtime errors caused by incorrect input formats.

4.3 Usage Example

Here's how to run a script from the command line:

```
$ python p_S2021428629.py 0xb71faf5190b8519a 0x46c68113cc01daeb
Starting padding oracle attack ...
Raw recovered bytes: 4544474152030303
Recovered plaintext  EDGAR
```

5 Results

Below are the decryption results for each ciphertext.

Ciphertext Index	`Plaintext (hex)`	Plaintext (ASCII)
1	`0xE584DEBD2ABAD5B3 0xCBD746544CDADF30`	"FRANCIS"
2	`0x56D3216564AAA6B4 0x7069B55CC9F69DDF`	"FREDDIE"
3	`0xB71FAF5190B8519A 0x46C68113CC01DAEB`	"EDGAR"
4	`0x4400A04574707149 0x15B8CF27546DA944`	"GERALD"
5	`0x59EABFB68676A258 0x20B26249E4E511E3`	"NORMAN"
6	`0x8E0E0363EADEB819 0x8B48D45F6BBFC361`	"STEPHEN"
7	`0x8325313861CCAE5D 0xA99FFD44412B9902`	"DON"
8	`0xF6AF1A2C85CD46BE 0x3F8F0B1D3A5B4AE4`	"WILLIAM"
9	`0x1DE8549FC6FF4A77 0xB2475C16FF15B180`	"CALVIN"
10	`0xF3E7A7327F1FC500 0xD2F8A91F2B61B980`	"THOMAS"

Table 1: Summary of decryption results

6 Discussion

6.1 Challenges Faced

Several problems were noticed during the experiment. First, Oracle consistently returned only `b'1'` or `b'0'` (in byte format) and not a simple integer such as 1, or a string such as "1". Failure to handle this variable may lead the code to misinterpret the "valid" padding response. To address this issue, we introduced a function `normalize_oracle_response(ok)` that translates Oracle's returns to a safely uniform integer 0 or 1. Another significant challenge was the occurrence of false positives. Since the original ciphertext can already end up as a valid padding, certain modified ciphertexts can incorrectly trigger a "valid padding" response even if they don't disclose new information. To counter that, we introduced a backtracking approach: Instead of immediately generating a "yes" to the first candidate byte, we collect and iterate *all* valid candidates for a given location, and accept the path only when it leads to a full recovery of the block. Finally, while our block size was only 8 bytes (smaller than many real systems), the volume of Oracle queries remains significant. Network latency, Oracle's speed limits, and the handling of misspelled ciphertexts (e.g., Oracle refuses to decrypt) all consider practical possibilities.

6.2 Performance Observation

Since our challenge used 8-byte blocks and a fairly controlled environment, we observed much lower recovery times and query volumes than the brute force required by the key. In the literature, prior research estimates approximately $128 \times b$ queries for b-byte blocks under ideal conditions, but our implementation handled an average of approximately hundreds of queries per block. Primarily was affected by the frequency with which ambiguous candidates (false positive) appeared and the degree to which backtracking was required. In the remote Oracle scenario, it is worth noting that each query added network latency and potentially server throttling overhead. Thus, while the number of theoretical queries remains the same, *real-time* per block can scale locally to minutes in a live environment.

6.3 Limits and Future Work

Although our attacks worked reliably under laboratory conditions, some limitations remain. First, we assume uninterrupted oracle access and the ability to submit many modified ciphertexts. Real-world systems can either limit requests or return common error messages that block inference. Second, we focused on fixed 8-byte block sizes and simple PKCS-style padding, and further work was required to adapt to different block sizes, different padding schemes, or Authenticated encryption with Associated Data (AEAD) modes. Third, although we included normalization and backtracking, we did not build a defense for timing side channels or active countermeasures (e.g., oracle delay responses to prevent navigation).

7 Conclusion

In this project, we performed a padding oracle attack on a CBC mode block cipher to directly demonstrate how a simple "yes/no" oracle can reveal plaintexts without knowing the encryption key. While this may seem like a clever trick, it reveals real-world vulnerabilities in systems that rely solely on block ciphers using standard padding. A Python implementation designed to handle 8-byte blocks has successfully exploited this attack. Significant improvements such as response normalization to handle Oracle singularities and backtracking logic to handle ambiguous cases have been introduced, making the attack not only functional but also robust even in laboratory settings. The results show that plaintext recovery is significantly faster than indiscriminate key substitution when the environment and oracle are in control. However, the experiment also highlighted some important caveats. The attack requires complete and iterative access to oracle and conditions that are often not met in real-world systems. Access can be limited in real-world deployments, and padding errors may not be revealed or authenticated encryption can be used. These factors greatly reduce the viability of such attacks. From a defense perspective, our findings highlight a clear lesson once again: If plaintext confidentiality is important, the encryption system should be used in conjunction with integrity checks. For example, instead of relying solely on padding for error signals, use Encrypt-Then-MAC, or Authenticated Encryption with Associated Data (AEAD). Smooth combination of confidentiality and integrity is the most reliable way to prevent padding oracle attacks. In summary, vulnerabilities exist, and these methods are only effective under the appropriate conditions.

8 References

[1] Juliano Rizzo and Thai Duong, *Practical Padding Oracle Attacks*, USENIX Workshop on Offensive Technologies (WOOT), 2010.

Source Code

Below is the full Python implementation used for this project.

Listing 1: Padding Oracle Attack Implementation (p_S2021428629.py)

```python
#!/usr/bin/env python3

import sys
from oracle_python_v1_2 import pad_oracle

def normalize_oracle_response(ok):

        if isinstance(ok, bytes):
                try:
                        s = ok.decode(errors='ignore').strip()
                except Exception:
                        return 0
                return 1 if s == '1' else 0
if isinstance(ok, str):
        s = ok.strip()
        return 1 if s == '1' else 0
if isinstance(ok, bool):
        return 1 if ok else 0
try:
        return 1 if int(ok) == 1 else 0
except Exception:
        return 0

BLOCKSIZE = 8

def hex_to_bytes_safe(s: str) -> bytes:
        if not isinstance(s, str):
                raise TypeError("Input must be a string")
        s = s.strip()
        if s.startswith(("0x", "0X")):
                s = s[2:]
        s = s.replace(" ", "")

        if len(s) % 2 != 0:
                s = '0' + s
        try:
                return bytes.fromhex(s)
        except ValueError as e:
                raise ValueError("Invalid hex string") from e

def hex_to_bytes_block(s:str, expected_len_bytes: int = BLOCKSIZE) -> bytes:
        b = hex_to_bytes_safe(s)
        if len(b) != expected_len_bytes:
                raise ValueError(f"Input must be {
                        expected_len_bytes} bytes long")
        return b

def bytes_to_hex_prefixed(b: bytes) -> str:
        return '0x' + b.hex()

def recover_block(C0_bytes, C1_bytes):

        assert len(C0_bytes) == BLOCKSIZE
        assert len(C1_bytes) == BLOCKSIZE

        C0 = bytearray(C0_bytes)
        recovered_intermediate = [None] * BLOCKSIZE
        recovered_plaintext = [None] * BLOCKSIZE

        C1_hex = bytes_to_hex_prefixed(bytes(C1_bytes))

        def build_prefix_for_index(idx, pad_len):
                prefix = bytearray(C0)
                for j in range(BLOCKSIZE - 1, idx, -1):
                        prefix[j] = recovered_intermediate[j] ^ pad_len
                return prefix

        def dfs(idx):
                if idx < 0:
                        return True

                pad_len = BLOCKSIZE - idx
                prefix_base = build_prefix_for_index(idx, pad_len)

                candidates = []
                for guess in range(256):
                        prefix_try = bytearray(prefix_base)
                        prefix_try[idx] = C0[idx] ^ guess ^ pad_len
                        C0_try_hex = bytes_to_hex_prefixed(bytes(prefix_try))
                        try:
                                ok = pad_oracle(C0_try_hex, C1_hex)
                        except Exception as e:
                                print("Oracle call error:", e)
                                print("Launch bridge process if not running.\nType > java -cp pad_oracle.jar:bcprov-jdk15-130.jar:python_interface_v1_2.jar python_interface_v1_2")
                                sys.exit(1)
                        if normalize_oracle_response(ok) == 1:
                                intermediate_byte = C0[idx] ^ guess
                                candidates.append(intermediate_byte)

                if not candidates:
                        return False

                for intermediate_byte in candidates:
                        recovered_intermediate[idx] = intermediate_byte
                        recovered_plaintext[idx] = intermediate_byte ^ C0[idx]

                        if dfs(idx - 1):
                                return True


                recovered_intermediate[idx] = None
                recovered_plaintext[idx] = None
                return False

        success = dfs(BLOCKSIZE - 1)
        if not success:
                raise RuntimeError("Failed to recover block (no valid padding candidates lead to full solution).")

        return bytes(bytearray(recovered_plaintext))

def remove_pkcs7_padding(p: bytes) -> bytes:
        if len(p) == 0:
                return p
        pad = p[-1]
        if pad < 1 or pad > BLOCKSIZE:
                raise ValueError("Invalid padding value when stripping.")
        if p[-pad:] != bytes([pad]) * pad:
                raise ValueError("Invalid padding bytes when stripping")
        return p[:-pad]

def attack(C0_hex: str, C1_hex: str):
        C0 = hex_to_bytes_block(C0_hex, BLOCKSIZE)
        C1 = hex_to_bytes_block(C1_hex, BLOCKSIZE)

        print("Starting padding oracle attack ...")
        recovered = recover_block(C0, C1)
        print("Raw recovered bytes", recovered.hex())

        try:
```

```python
                plain = remove_pkcs7_padding(recovered)
                try:
                        printable = plain.decode('ascii')
                except Exception:
                        printable = repr(plain)
                print("Recovered plaintext ", printable)
        except ValueError as e:
                print("Warning: could not strip padding safely", e)
                try:
                        print("Recovered plain text as ASCII attempt: ", recovered.decode('ascii', errors='replace'))
                except Exception:
                        print("Recovered plaintext bytes: ", recovered)

        return recovered

if __name__ == "__main__":
        if len(sys.argv) != 3:
                print("Usage: python p_S2021428629.py 0xC0HEX 0xC1HEX")
                sys.exit(1)

        C0_hex = sys.argv[1]
        C1_hex = sys.argv[2]

        try:
                attack(C0_hex, C1_hex)
        except Exception as exc:
                print("Attack failed:", exc)
                sys.exit(1)
```