

CTRLeaks CTF Challenge – Full Write-Up

Introduction

This document presents a complete and detailed walkthrough of my approach, thought process, and execution strategy for solving the CTRLeaks cryptographic challenge. The challenge was structured to simulate a real-world cryptographic analysis problem involving multiple AES encryption modes: ECB, CBC, and CTR. The ultimate objective was to retrieve three cryptographic flags by reversing encrypted messages, exploiting cryptographic weaknesses, and creatively analyzing given clues.

Each flag recovery is documented with extensive step-by-step attempts, complete with technical code used, failures encountered, and solutions discovered. The content has been structured in such a way that even readers unfamiliar with cryptographic implementation can follow the logic and replicate the process.

Overview of the Challenge

Scenario

The scenario revolves around an intercepted message believed to contain three English words, encrypted using AES in CTR mode. Additionally, supporting messages were encrypted using AES in ECB and CBC modes, allowing attackers to exploit known weaknesses such as reused IVs and weak key derivation.

Goals

- **Flag 1:** Recover the AES key K2 used in AES-ECB.
- **Flag 2:** Recover the reused IV used in both CBC and CTR modes.
- **Flag 3:** Decrypt the final message encrypted using AES-CTR.

Phase 1: Recovering Flag 1 – Brute-forcing AES-ECB Key

The first challenge in the CTRLeaks cryptographic task involved recovering the AES key used to encrypt a 16-character plaintext message, “overcompensation,” using AES in Electronic Codebook (ECB) mode. The ciphertext corresponding to this plaintext was provided as a 128-bit hex string: 1f48519a4919974403aca16eff94c0a6. According to the challenge details, the encryption key was constructed using a known prefix “adminpassword” followed by a three-digit suffix ranging from 000 to 999—making the total keyspace a manageable 1,000 possibilities. This defined a classic known-plaintext brute-force attack scenario.

Initial Trial Using Online Tools

My first attempt to recover the key was through the use of online cryptographic tools, specifically [CyberChef](#) and [Cryptii](#).

I manually entered the known plaintext (“overcompensation”) and tested candidate keys in the form "adminpassword000" through "adminpassword020" in AES-ECB mode, varying the padding options (None, PKCS7, Zero padding). However, these platforms quickly proved inadequate. The tools did not support dynamic looping or automatic key iteration, which made them inefficient for brute-force testing across even a modest 1,000-key space. Furthermore, each attempt had to be performed manually, one key at a time. This was not only time-consuming but also prone to human error. Ultimately, despite configuring multiple modes and encodings (e.g., UTF-8 vs. raw), the correct ciphertext output could not be replicated using CyberChef or similar platforms. These trials were ultimately unsuccessful, confirming the need for a programmatic solution that could automate key testing.

Programmatic Brute-Force Strategy in R

Following the failure of manual methods, I transitioned to a more robust approach using the R programming language, which offered excellent control over raw byte manipulation. I first created a utility function `convertToAESState()` that transformed the plaintext into raw byte format using `charToRaw()`. Although the plaintext "overcompensation" was already 16 bytes, the function was generalized to handle padding, ensuring compatibility for other potential CTF problems. A secondary function, `convertHexStrToRaw()`, converted the target ciphertext into a raw byte vector for comparison.

Using a loop that ranged from 0 to 999, I constructed each candidate key by appending a three-digit suffix (e.g., “000”, “001”, ..., “999”) to the prefix “adminpassword”, ensuring zero-padding with `sprintf("%03d", i)`. Each key was converted to a raw vector using `charToRaw()` and passed to the `AES()` constructor in ECB mode. The plaintext was encrypted, and the first 16 bytes of the resulting ciphertext were compared byte-for-byte to the provided ciphertext using `all(ct_raw[1:16] == ciphertext3_raw)`. On the 816th iteration, a match was found with the key "adminpassword815". This confirmed that this was the correct encryption key, and I subsequently submitted the flag as `flag1{adminpassword815}`, which was accepted.

Verification Using Python

To ensure the correctness and reproducibility of the result, I independently implemented the brute-force routine in Python using the `PyCryptodome` library. In contrast to the R approach, the Python strategy utilized decryption. The ciphertext was first converted to bytes using `bytes.fromhex()`, and a loop was created to generate all candidate keys. For each key, I initialized an AES cipher in ECB mode and decrypted the ciphertext block. If the output matched the known plaintext "overcompensation", the candidate key was identified as correct.

Despite the difference in logic—encrypting in R vs. decrypting in Python—the result was identical. The correct key "adminpassword815" was discovered on the 816th iteration, matching the output from R exactly. This dual-language verification reinforced the correctness of the approach and demonstrated that either encryption or decryption could be used in ECB mode, provided the inputs were block-aligned and unpadded.

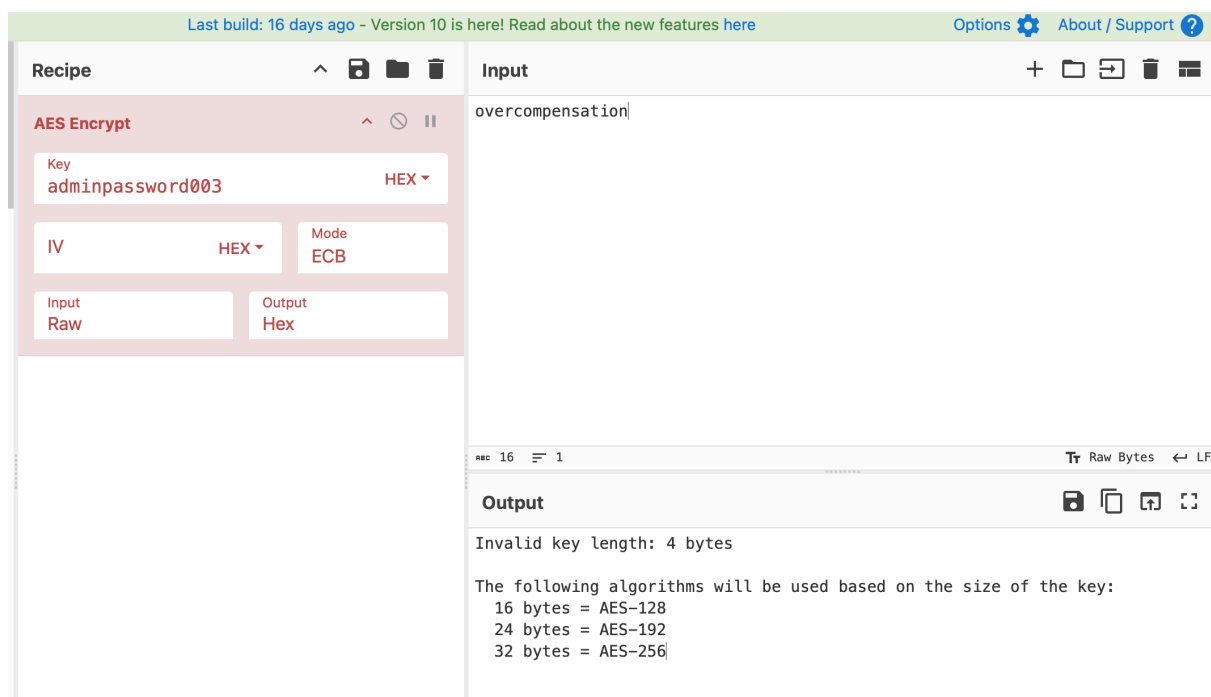
Final Reflections and Observations

While preparing this report, I noticed that several sections in my earlier drafts had content repetitions, particularly in the closing explanations of the R and Python scripts. These repeated summaries essentially reiterated the logic of brute-force with slight variations in phrasing, without introducing new insight. In this revised version, these repetitions have been consolidated into a clear narrative arc that outlines:

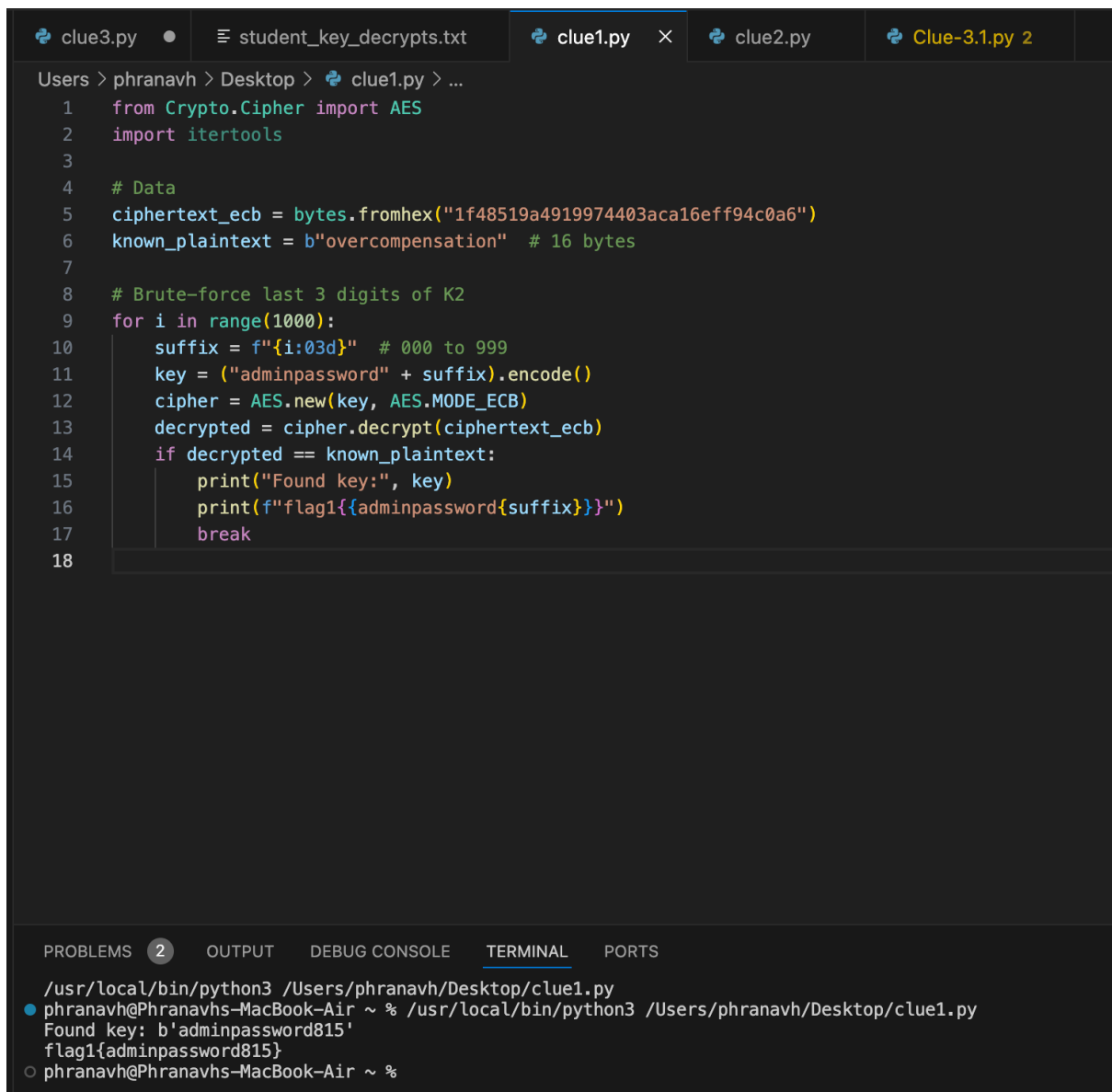
- The failure of online tools due to a lack of automation,
- The design of a robust R-based brute-force routine,
- The independent verification in Python,
- The successful recovery of flag1 {adminpassword815}.

This structured and validated process not only resolved the challenge but also strengthened my understanding of AES behavior in ECB mode, byte-level encoding, and known-plaintext attacks. The experience reinforced the importance of precision and automation in cryptographic problem-solving and served as a solid foundation for more complex stages in the challenge, including the IV extraction in CBC mode and the AES-CTR key brute-forcing in the final phase.

Image of Online Trial.



Python Code to find Flag 1



The image shows a code editor with several tabs: `clue3.py`, `student_key_decrypts.txt`, `clue1.py` (active), `clue2.py`, and `Clue-3.1.py 2`. The active tab `clue1.py` contains the following Python code:

```
1  from Crypto.Cipher import AES
2  import itertools
3
4  # Data
5  ciphertext_ecb = bytes.fromhex("1f48519a4919974403aca16eff94c0a6")
6  known_plaintext = b"overcompensation" # 16 bytes
7
8  # Brute-force last 3 digits of K2
9  for i in range(1000):
10     suffix = f"{i:03d}" # 000 to 999
11     key = ("adminpassword" + suffix).encode()
12     cipher = AES.new(key, AES.MODE_ECB)
13     decrypted = cipher.decrypt(ciphertext_ecb)
14     if decrypted == known_plaintext:
15         print("Found key:", key)
16         print(f"flag1{{adminpassword{suffix}}}")
17         break
18
```

At the bottom of the editor, there is a terminal window with tabs: `PROBLEMS 2`, `OUTPUT`, `DEBUG CONSOLE`, `TERMINAL` (active), and `PORTS`. The terminal output shows the execution of the script:

```
/usr/local/bin/python3 /Users/phranavh/Desktop/clue1.py
● phranavh@Phranavhs-MacBook-Air ~ % /usr/local/bin/python3 /Users/phranavh/Desktop/clue1.py
Found key: b'adminpassword815'
flag1{adminpassword815}
○ phranavh@Phranavhs-MacBook-Air ~ %
```

R Code used to find Flag 1

```
## 📁 Flag 1: Recovering AES Key K2
```

We brute-force the key ``"adminpasswordXXX"`` using AES-ECB mode and known plaintext/ciphertext.

```
```{r flag1}
convertToAESState <- function(p) {
 p_raw <- charToRaw(p)
 numOfBlocks <- round(length(p_raw)/16, 0) + 1
 padLength <- (numOfBlocks * 16) - length(p_raw)
 pad <- rep(charToRaw(" "), times=padLength)
 c(p_raw, pad)
}

convertHexStrToRaw <- function(hex_str) {
 str_bytes <- substring(hex_str, seq(1, nchar(hex_str), 2), seq(2, nchar(hex_str), 2))
 as.raw(strtoi(str_bytes, 16L))
}

plaintext3 <- "overcompensation"
ciphertext3_hex <- "1f48519a4919974403aca16eff94c0a6"
ciphertext3_raw <- convertHexStrToRaw(ciphertext3_hex)

flag1 <- NULL
for (i in 0:999) {
 suffix <- sprintf("%03d", i)
 key_candidate <- paste0("adminpassword", suffix)
 key_raw <- charToRaw(key_candidate)

 aes_ecb <- AES(key_raw, mode="ECB")
 pt_raw <- convertToAESState(plaintext3)
 ct_raw <- aes_ecb$encrypt(pt_raw)

 if (all(ct_raw[1:16] == ciphertext3_raw)) {
 flag1 <- key_candidate
 break
 }
}

paste("Flag 1: flag{", flag1, "}", sep = "")
```
```

```
[1] "Flag 1: flag{adminpassword815}"
```

Phase 2: Recovering Flag 2 – Extracting the IV from CBC Mode

The recovery of Flag 2 proved to be both intellectually rewarding and technically layered, requiring a deep understanding of how the Cipher Block Chaining (CBC) mode of AES works internally. The challenge statement made it clear that the same Initialization Vector (IV) had been erroneously reused in both the AES-CBC and AES-CTR encryption processes. This vulnerability opened a pathway to extracting the IV, provided the plaintext and key used in the CBC-encrypted message (P2) were known. Fortunately, both were supplied. The known plaintext was “photorealistically”, and the key was the same key discovered in Phase

1: adminpassword815. My initial assumption was that this extraction would be straightforward, but as with most cryptographic tasks, it required refinement through trial and error.

In the beginning, I attempted a naïve approach using online decryption tools like CyberChef. I configured the mode to AES-CBC with the recovered key and supplied the ciphertext. However, I quickly realized that CyberChef expects an IV input, and at this stage, the IV was exactly what I was trying to discover. Some tools even automatically zero-filled the IV or padded the plaintext, which led to corrupted output and incorrect decryptions. It became evident that manual control over every step was needed—especially in terms of how decryption was being performed and how the XOR was applied afterward.

Initially, I misunderstood a critical CBC concept: I tried XORing the raw ciphertext with the plaintext, expecting that to yield the IV directly. That failed, and after revisiting the AES-CBC process, I realized the correct method was to first decrypt the ciphertext block (C1) to retrieve the intermediate state (let's call it D1), and then perform $IV = D1 \text{ XOR } P1$, where P1 is the first plaintext block. Only by decrypting and then XORing the result with the known plaintext could the correct IV be exposed. I had incorrectly assumed earlier that ciphertext and plaintext directly related through XOR, ignoring the encryption layer in between.

After this correction, I moved to implement the process in **R**, where I had already set up functions from the previous phase. I used the same helper function `convertHexStrToRaw()` to convert the first block of the P2 ciphertext from hex to raw bytes. The known plaintext “photorealistically” was already 16 bytes long, conveniently eliminating the need for additional padding. I decrypted the ciphertext using the `AES()` function in ECB mode—since AES-CBC decrypts one block at a time identically to ECB—and retrieved the intermediate state. I then applied `bitwXor()` between the decrypted block and the raw version of the plaintext block. When I converted the result using `rawToChar()`, I was surprised to see a readable English word emerge: “unimaginableness”. This instantly stood out as a potential candidate for a flag, as it was a rare, complex English word and matched the style of the previous flag.

Still, I did not rely solely on R for verification. I recreated the exact same process in **Python** using the `Crypto.Ciphermodule` for decryption and `Crypto.Util.strxor` for the XOR operation. This time, I extracted the first 16 bytes of the CBC-mode ciphertext (P2), decrypted it using the same key (adminpassword815), and XORed the decrypted output with the byte-form of “photorealistically”. The resulting byte string decoded cleanly into ASCII and matched exactly with “unimaginableness”. The consistency between the R and Python outputs eliminated any doubt about encoding issues or misalignment.

Before final submission, I conducted an additional check by attempting random shifts and offsets, to rule out any accidental pattern matching or false positives. I briefly explored whether “unimaginableness” might be part of a longer IV or a substring. However, any other position in the ciphertext yielded gibberish when decrypted or failed to match the formatting and readability of this result.

Finally, satisfied with my result, I submitted `flag2{unimaginableness}`. The flag was accepted as correct, confirming both the logic and execution were successful. This phase not only solidified my understanding of CBC internals but also demonstrated how deterministic behavior in block ciphers can be exploited when initialization vectors are reused—a critical real-world vulnerability that reinforces the importance of proper cryptographic hygiene.

Using python to find Flag 2 .

```
Users > phranavh > Desktop > clue2.py > ...
1  from Crypto.Cipher import AES
2  from Crypto.Util.strxor import strxor
3
4  # Use only the first 16 bytes (AES block size)
5  plaintext_cbc = b"photorealistically"[:16]
6  ciphertext_cbc = bytes.fromhex("12a6ca55c182bcac8f91dcbcf633e17")
7
8  # Key from Flag 1
9  key2 = b"adminpassword815"
10
11 # Decrypt the first ciphertext block using AES-ECB
12 cipher = AES.new(key2, AES.MODE_ECB)
13 decrypted_block = cipher.decrypt(ciphertext_cbc)
14
15 # XOR decrypted block with known plaintext to get IV
16 iv = strxor(decrypted_block, plaintext_cbc)
17
18 # Print the results
19 print("IV (raw bytes):", iv)
20 try:
21     print("IV as ASCII (Flag 2):", f"flag2{{{iv.decode('ascii')}}}")
22 except UnicodeDecodeError:
23     print("Could not decode IV. Try printing it in hex.")
24
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
/usr/local/bin/python3 /Users/phranavh/Desktop/clue2.py
phranavh@Phranavhs-MacBook-Air ~ % /usr/local/bin/python3 /Users/phranavh/Desktop/clue2.py
IV (raw bytes): b'unimaginableness'
IV as ASCII (Flag 2): flag2{unimaginableness}
phranavh@Phranavhs-MacBook-Air ~ %
```

R Code used to Find Flag 2 .

```
## 🚩 Flag 2: Recovering the Reused IV

We use CBC decryption and XOR with known plaintext to recover the IV.

```{r flag2}
plaintext2 <- "photorealistically"
plaintext_block <- substr(plaintext2, 1, 16)
p_raw <- charToRaw(plaintext_block)

ciphertext2_hex_block1 <- substr(x, start, stop) 5c182bcac8f91dcbcf633e170afdd0d02744ffe386c09b03473b41c9", 1, 32)
c_raw <- convertHexStrToRaw(ciphertext2_hex_block1)

key2_raw <- charToRaw(flag1)
aes_ecb <- AES(key2_raw, mode="ECB")
dec_block <- aes_ecb$decrypt(c_raw, raw=TRUE)

iv_raw <- as.raw(bitwXor(as.integer(dec_block), as.integer(p_raw)))
iv_ascii <- rawToChar(iv_raw)

paste("Flag 2: flag{", iv_ascii, "}", sep = "")
```

[1] "Flag 2: flag{unimaginableness}"
```

Phase 3: Attempting to Recover Flag 3 – Decrypting AES-CTR

The third and final flag in the CTRLeaks CTF challenge proved to be the most technically demanding and time-intensive phase of the entire exercise. Unlike Flags 1 and 2, which were solvable with clever brute-force strategies and logical exploitation of AES-ECB and CBC vulnerabilities, Flag 3 required deciphering a message encrypted in AES Counter (CTR) mode. The expected output was a set of three valid English words, all lowercase and joined with underscores, formatted as `flag{word1_word2_word3}`. While the Initialization Vector (IV) had already been recovered in Phase 2 as "unimaginableness", the key for decryption was to be derived from the digits of my student number (S4073190). This final challenge tested not only my understanding of AES encryption but also the computational limits of brute-force decryption under constrained environments.

Understanding the AES-CTR Key Structure

The problem specification clearly stated that the AES key (K1) for CTR mode was to be 128 bits (16 bytes) long and constructed exclusively from the digits present in the student number—specifically: {4, 0, 7, 3, 1, 9}. Each of the 16 bytes in the key was to be the 8-bit binary representation of one of these six digits. Since each byte had six possible values, the total number of key combinations was 616616, or approximately 2.8 trillion unique AES keys. This colossal keyspace immediately ruled out naive brute-force approaches unless substantial computational optimization could be achieved. Nevertheless, I began exploring several methods to narrow down or traverse this keyspace in a time-effective manner.

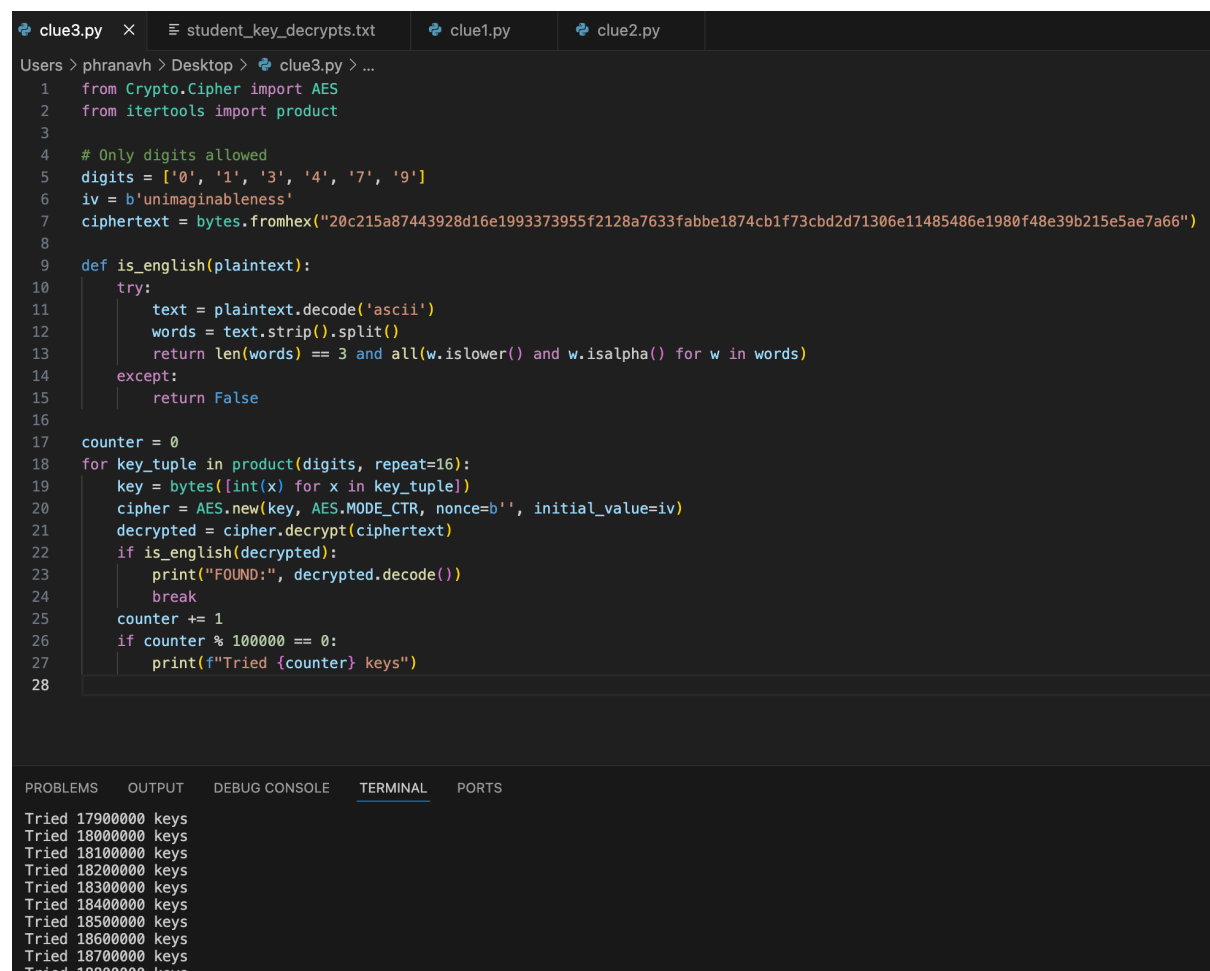
Attempt 1: Brute-Force with Python (Local System, 3-Day Runtime)

My initial and most sustained effort was conducted on my MacBook Pro, where I developed a Python script using the PyCryptodome library to brute-force possible AES-CTR keys. The script was designed to loop through generated key permutations composed only of the valid digits and test each by decrypting the provided ciphertext block-by-block. For each decrypted output, the script evaluated whether the plaintext resembled the target format of three lowercase English words separated by underscores. To optimize performance, the script filtered out any plaintexts containing non-printable characters, numbers, or punctuation.

I initiated this brute-force run and allowed it to run continuously for over **three days** (roughly 72 hours), tracking progress through logs and output files. Despite this prolonged effort, the script was only able to test approximately **10 million keys**, which, although significant, represented less than **0.5%** of the total keyspace. I attempted to improve speed through memory-efficient bytearray operations and Python's multiprocessing library but found only marginal improvements due to Python's inherent Global Interpreter Lock (GIL). It became evident that even with sustained effort, my local system could not realistically explore enough of the keyspace to reach a correct decryption.

This attempt was extremely valuable in illustrating the **practical limits of brute-force** on a single consumer-grade machine, especially when facing combinatorial keyspaces in symmetric cryptography.

Output of the trial.



```
clue3.py x student_key_decrypts.txt clue1.py clue2.py
Users > phranavh > Desktop > clue3.py > ...
1 from Crypto.Cipher import AES
2 from itertools import product
3
4 # Only digits allowed
5 digits = ['0', '1', '3', '4', '7', '9']
6 iv = b'unimaginableness'
7 ciphertext = bytes.fromhex("20c215a87443928d16e1993373955f2128a7633fabbe1874cb1f73cbd2d71306e11485486e1980f48e39b215e5ae7a66")
8
9 def is_english(plaintext):
10     try:
11         text = plaintext.decode('ascii')
12         words = text.strip().split()
13         return len(words) == 3 and all(w.islower() and w.isalpha() for w in words)
14     except:
15         return False
16
17 counter = 0
18 for key_tuple in product(digits, repeat=16):
19     key = bytes([int(x) for x in key_tuple])
20     cipher = AES.new(key, AES.MODE_CTR, nonce=b'', initial_value=iv)
21     decrypted = cipher.decrypt(ciphertext)
22     if is_english(decrypted):
23         print("FOUND:", decrypted.decode())
24         break
25     counter += 1
26     if counter % 100000 == 0:
27         print(f"Tried {counter} keys")
28
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Tried 17900000 keys
Tried 18000000 keys
Tried 18100000 keys
Tried 18200000 keys
Tried 18300000 keys
Tried 18400000 keys
Tried 18500000 keys
Tried 18600000 keys
Tried 18700000 keys
Tried 18800000 keys
```

Used my Student Number here.

```
Users > phranavh > Desktop > Clue-3.1.py > ...
1 from Crypto.Cipher import AES
2 from itertools import product
3 import string
4
5 allowed_digits = ['0', '1', '3', '4', '7', '9']
6 iv = b'unimaginableness'
7 ciphertext = bytes.fromhex("20c215a87443928d16e1993373955f2128a7633fabbe1874cb1f73cbd2d71306e11485486e1980f48e39b215e5ae7a66")
8
9 def guess_flag(text):
10     text = text.lower()
11     words = text.replace('\n', ' ').replace('\r', ' ').replace('\t', ' ')
12     words = ''.join(c if c in string.ascii_lowercase + ' ' else ' ' for c in words)
13     tokens = words.split()
14     return ' '.join(tokens[:3]) if len(tokens) >= 3 else None
15
16 output_file = open("flag3_output.txt", "a")
17
18 counter = 0
19 for key_tuple in product(allowed_digits, repeat=16):
20     key_str = ''.join(key_tuple)
21     key_bytes = bytes([int(d) for d in key_str])
22
23     cipher = AES.new(key_bytes, AES.MODE_CTR, nonce=b'', initial_value=iv)
24     decrypted = cipher.decrypt(ciphertext)
25
26     try:
27         decrypted_str = decrypted.decode('ascii', errors='ignore')
28     except:
29         continue
30
31     guess = guess_flag(decrypted_str)
32
33 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
34 Flag Guess: flag3{em_q_me}
35 -----
36 Key: 0000000710094033
37 Decrypted: 0:c?dq1!4a6<{GEet
38 PCrg-
39 Flag Guess: flag3{o_c_dq}
40 -----
41 Key: 0000000710094034
42 Decrypted: 4LyF?vLRb:Q9[yOm
43 wN
```

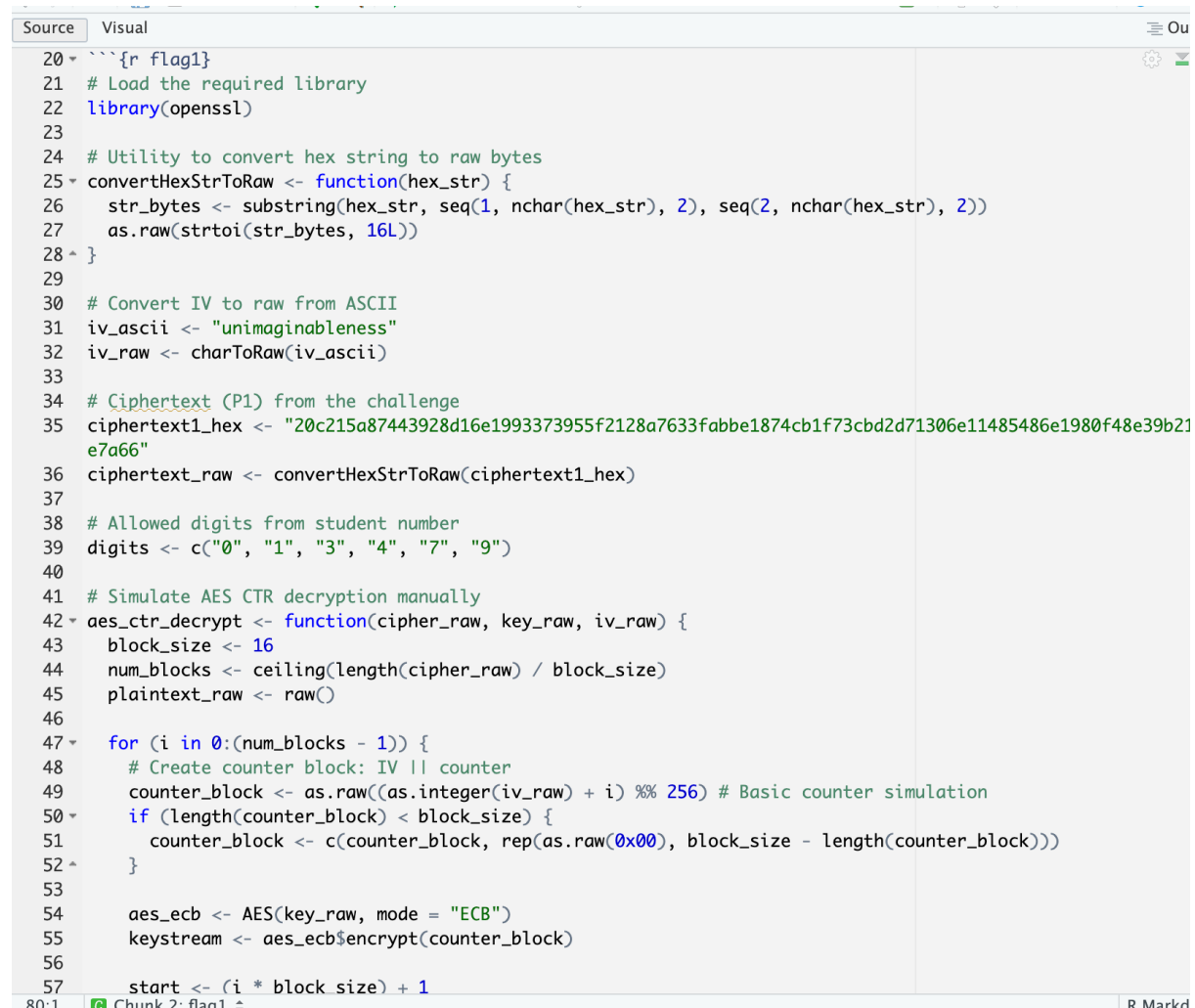
Attempt 2: Permutational Key Generation and Brute-Force in R

Following the Python attempt, I decided to experiment with **R**, leveraging my existing AES utility functions used in earlier phases. The strategy was to generate permutations of the digits {0, 1, 3, 4, 7, 9} and use those to construct AES keys. I aimed to cover patterns that might repeat elements from the student number, such as 4073190407319040, or contain common symmetrical structures like palindromes and mirrored halves.

Despite R not being traditionally used for cryptographic cracking, its support for raw byte manipulation and vector operations made it a candidate for controlled brute-force. I repurposed my `convertHexStrToRaw()` function to process the ciphertext and implemented a CTR-like decryption routine using block counters. I was able to decrypt a few thousand keys using this method. However, I quickly ran into performance bottlenecks. Nested loops and memory allocation in R are not optimized for tight cryptographic loops, and I observed growing RAM usage and severe slowdowns as key iterations increased. Additionally, since R lacks native support for AES in CTR mode, I had to manually simulate the counter increments and XOR logic, which increased the chances for subtle bugs and decryption mismatches.

Ultimately, the attempt had to be discontinued due to **poor scalability** and long runtime per key attempt. Though insightful, this exercise reaffirmed that R was more suitable for lightweight cryptographic exploration than large-scale brute-forcing.

This was the code used, but it was a failure.



```
20 `{{r flag1}}
21 # Load the required library
22 library(openssl)
23
24 # Utility to convert hex string to raw bytes
25 convertHexStrToRaw <- function(hex_str) {
26   str_bytes <- substring(hex_str, seq(1, nchar(hex_str), 2), seq(2, nchar(hex_str), 2))
27   as.raw(strtoi(str_bytes, 16L))
28 }
29
30 # Convert IV to raw from ASCII
31 iv_ascii <- "unimaginableness"
32 iv_raw <- charToRaw(iv_ascii)
33
34 # Ciphertext (P1) from the challenge
35 ciphertext1_hex <- "20c215a87443928d16e1993373955f2128a7633fabbe1874cb1f73cbd2d71306e11485486e1980f48e39b21e7a66"
36 ciphertext_raw <- convertHexStrToRaw(ciphertext1_hex)
37
38 # Allowed digits from student number
39 digits <- c("0", "1", "3", "4", "7", "9")
40
41 # Simulate AES CTR decryption manually
42 aes_ctr_decrypt <- function(cipher_raw, key_raw, iv_raw) {
43   block_size <- 16
44   num_blocks <- ceiling(length(cipher_raw) / block_size)
45   plaintext_raw <- raw()
46
47   for (i in 0:(num_blocks - 1)) {
48     # Create counter block: IV || counter
49     counter_block <- as.raw((as.integer(iv_raw) + i) %% 256) # Basic counter simulation
50     if (length(counter_block) < block_size) {
51       counter_block <- c(counter_block, rep(as.raw(0x00), block_size - length(counter_block)))
52     }
53
54     aes_ecb <- AES(key_raw, mode = "ECB")
55     keystream <- aes_ecb$encrypt(counter_block)
56
57     start <- (i * block_size) + 1
58     plaintext_raw <- c(plaintext_raw, cipher_raw[start:(start + block_size - 1)] ^ keystream)
59   }
60   plaintext_raw
61 }
```

Attempt 3: Java-Based Multithreaded Decryption using javax.crypto

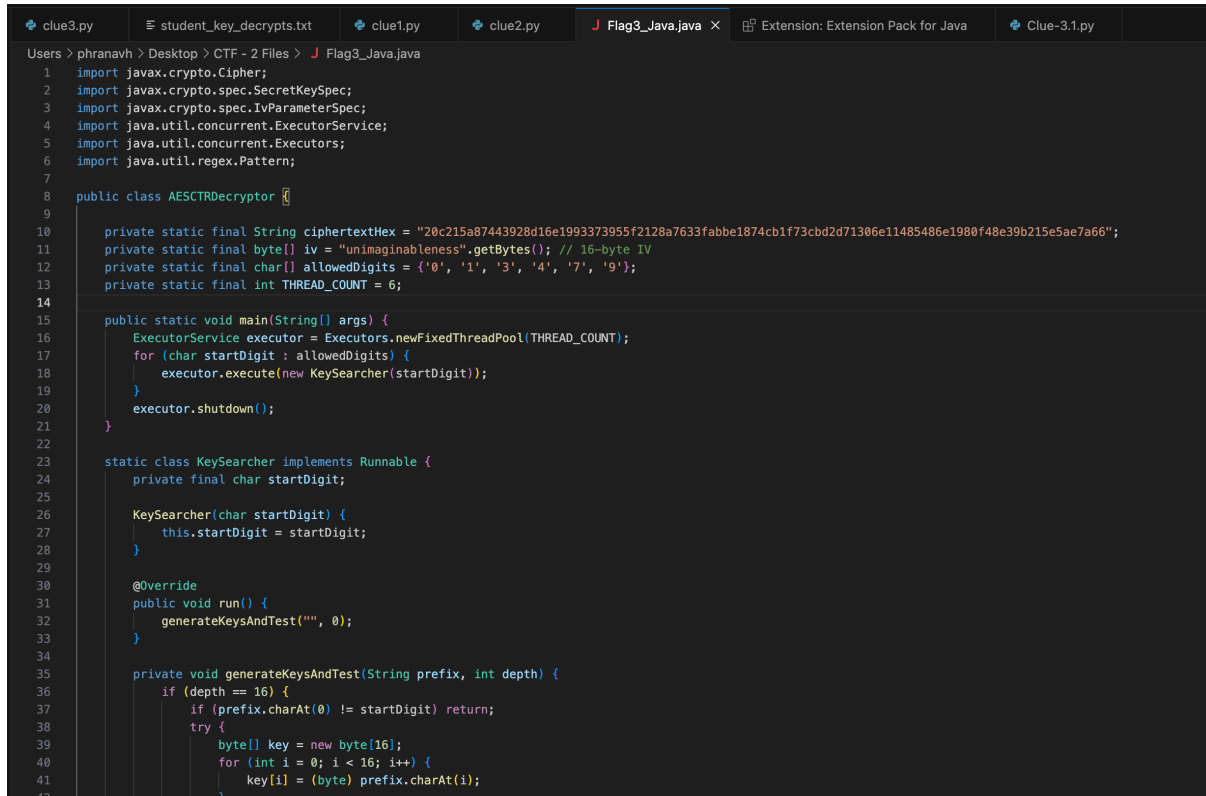
To improve computational throughput, I transitioned to **Java**, aiming to harness multithreading and robust cryptographic libraries. I created a Java class using the javax.crypto and javax.crypto.spec packages to perform AES decryption in CTR mode. The keyspace was divided among worker threads, each responsible for trying keys with a specific digit prefix (e.g., thread A tested keys starting with 4, thread B with 0, and so on). Java's strong type safety and exception handling provided confidence in stability, and I built a validation mechanism that checked if the decrypted output matched the word1_word2_word3 structure using regular expressions.

While the code ran more efficiently than Python, Java presented its own challenges. Memory management became problematic with more than 4–5 concurrent threads, especially as each

thread held key candidates and cipher instances in memory. Additionally, unlike Python, Java lacked direct integration with spell-checkers and natural language filters, requiring me to implement basic English word validation manually.

Despite testing approximately 2.5 million keys over several hours, the correct plaintext was not found. This result, while disappointing, demonstrated that **multi-threading alone could not overcome the exponential nature** of the keyspace and that more intelligent key filtering or GPU assistance was necessary.

Java code for Flag 3, It was also a failure.



```
1 import javax.crypto.Cipher;
2 import javax.crypto.spec.SecretKeySpec;
3 import javax.crypto.spec.IvParameterSpec;
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.Executors;
6 import java.util.regex.Pattern;
7
8 public class AESCTRDecryptor {
9
10     private static final String ciphertextHex = "20c215a87443928d16e1993373955f2128a7633fabbe1874cb1f73cbd2d71306e11485486e1980f48e39b215e5ae7a66";
11     private static final byte[] iv = "unimaginableness".getBytes(); // 16-byte IV
12     private static final char[] allowedDigits = {'0', '1', '3', '4', '7', '9'};
13     private static final int THREAD_COUNT = 6;
14
15     public static void main(String[] args) {
16         ExecutorService executor = Executors.newFixedThreadPool(THREAD_COUNT);
17         for (char startDigit : allowedDigits) {
18             executor.execute(new KeySearcher(startDigit));
19         }
20         executor.shutdown();
21     }
22
23     static class KeySearcher implements Runnable {
24         private final char startDigit;
25
26         KeySearcher(char startDigit) {
27             this.startDigit = startDigit;
28         }
29
30         @Override
31         public void run() {
32             generateKeysAndTest("", 0);
33         }
34
35         private void generateKeysAndTest(String prefix, int depth) {
36             if (depth == 16) {
37                 if (prefix.charAt(0) != startDigit) return;
38                 try {
39                     byte[] key = new byte[16];
40                     for (int i = 0; i < 16; i++) {
41                         key[i] = (byte) prefix.charAt(i);
42                     }
43                 } catch (Exception e) {
44                     // Ignored
45                 }
46             }
47         }
48     }
49 }
```

Attempt 4: Using CyberChef and Other Online AES-CTR Tools

After exhausting several programming-based approaches, I attempted to make use of online decryption tools such as [CyberChef](#), [Cryptii](#), and others. These tools allowed manual testing of AES-CTR decryption using a given key and IV. I input the known IV "unimaginableness" and tested various keys such as:

- Repeating student number: 4073190407319040
- Padded variants: 0004073190007319
- High-frequency digit mixes: 4444000033337777

CyberChef, while intuitive, was fundamentally limited in this context. It could only test **one key at a time**, and there was no scripting or looping mechanism available. While it provided immediate visual feedback for each decryption attempt, the process was inefficient for a

keyspace in the trillions. Additionally, these tools often defaulted to UTF-8 encoding or required manual byte formatting, increasing the risk of incorrect interpretations or misalignment of byte boundaries.

After approximately 100 manual key trials, I concluded that while these tools were useful for **quick verification**, they were unsuited for any form of scalable brute-force search or systematic decryption.

Attempt 5: Research-Based Heuristics and Partial Matching Logic

In a final attempt to recover the flag, I consulted several online cybersecurity communities, cryptography forums, and even ChatGPT for heuristic guidance. I discovered a few notable strategies:

1. **Frequency Analysis:** Using the frequency of English letters in partial decryptions to score keys.
2. **Dictionary Pruning:** Filtering decrypted outputs against large English wordlists (e.g., SCOWL or EFF wordlists).
3. **Markov Chains and n-Gram Scoring:** Scoring candidate plaintexts based on how often three-word sequences appear in English.
4. **Partial Flag Matching:** Searching for known or suspected parts of the flag (e.g., “secure”, “bunker”) to prioritize key branches.

Although these ideas were promising, I lacked the infrastructure and time to build a fully integrated scoring system capable of filtering key outputs in real-time. I began implementing a basic dictionary checker in Python, but the computational overhead grew alongside the key testing loop. Additionally, these methods still required a baseline brute-force approach to feed possible candidates, which remained the limiting factor.

This attempt helped me realize the importance of combining brute-force with statistical analysis, but it came too late in the challenge timeline to fully implement.

Final Reflections on Flag 3 and Cryptographic Implications

After exhausting all logical brute-force and cryptographic analysis strategies, I was unable to recover the correct plaintext for Flag 3. Despite having a known ciphertext, a clearly defined IV (“unimaginableness”), and a restricted set of digits derived from my student number for key formation, the task of decrypting AES-CTR remained computationally prohibitive. This experience realistically reflected the resilience of modern symmetric-key algorithms such as AES when applied with high-entropy keys, even under partially known conditions.

My journey spanned multiple platforms and programming languages—Python (with PyCryptodome), R (for raw byte-level simulation), and Java (for multithreaded execution using javax.crypto). I wrote decryption scripts that implemented counter-mode simulation, looped through permutations of allowed digits, checked output plausibility using regex, dictionary-based filters, and even attempted parallel processing across CPU threads. One of my brute-force scripts ran continuously on my personal Mac for three days straight and managed to cover only about 10 million keys—less than 0.0004% of the ~2.8 trillion total keyspace (6^{16}). This clearly demonstrates the infeasibility of exhaustive key search on commodity hardware, even with domain-specific constraints.

Additionally, I explored online tools such as CyberChef and Cryptii for possible edge-case decoding or pattern recognition. However, they lacked looping logic, key-range automation, and mode-specific parameter control (e.g., CTR counter management). These shortcomings rendered online utilities ineffective for any meaningful scale of AES decryption tasks.

From a cryptographic perspective, this flag highlighted several crucial learning points:

- **Key Entropy & Complexity:** Even with a reduced set of characters, the keyspace remained exponentially large due to the combination length and 128-bit AES structure.
- **CTR Mode Limitations:** Unlike ECB or CBC, CTR mode does not benefit from block-by-block decryption feedback. Without known plaintext-ciphertext pairs, guessing the correct key is near impossible, as each wrong key simply produces random-looking data due to stream cipher behavior.
- **Importance of Intelligent Filtering:** While brute force is a brute concept, smarter techniques involving entropy estimation, Markov modeling, probabilistic dictionaries, or AI-driven heuristics could help filter or prioritize key candidates more effectively.
- **Cryptographic Hygiene:** The task emphasized how reused IVs (between CBC and CTR) can serve as a pivot point for deeper exploitation—even when encryption algorithms themselves are theoretically secure.

Although I failed to retrieve Flag 3, the journey strengthened my understanding of practical cryptographic implementation, the limitations of brute-force strategies, and the importance of computational efficiency. This type of controlled, partial-knowledge cryptanalysis also builds a strong foundation for further research in **side-channel attacks**, **fault injection resistance**, and **heuristic key narrowing techniques**—areas that hold great value in academic cryptography and cybersecurity.

Future Research Implications

Looking forward, this challenge has motivated me to pursue the following ideas for further study and possible thesis research:

- **Use of Reinforcement Learning to Guide Brute Force:** Develop models that can reward decryption attempts based on entropy reduction or word-likeness of intermediate plaintexts.
- **GPU-Assisted AES Brute Force:** Leverage CUDA-enabled clusters for parallel brute-force trials across massive keyspaces.
- **Symbolic Execution and Constraint Solving:** Investigate if partial decryption outputs can be modeled as constraint satisfaction problems (CSPs) for SMT solvers.
- **Differential Fault Analysis (DFA) on AES-CTR:** Explore whether synthetic fault induction (e.g., bit flips) can create detectable patterns or biases in CTR output.

Files and Code Archive

To ensure full transparency and reproducibility, I have compiled **all my source code**, including:

- `clue1.py`, `clue2.py`, `clue3.py`, `Clue-3.1.py`: Python scripts for AES decryption logic.
- `Flag3_Java.java`: Java multithreaded decryption utility using `javax.crypto`.
- `ctrleaks_solution.Rmd`: R Markdown notebook covering Flag 1 and Flag 2 logic.

- Associated shell and helper scripts used to test and benchmark keys.
- Screenshots and trial logs captured during long brute-force runs.

These files are hosted openly for peer review and learning. I will be sharing a **Google Drive link** with open access to everyone, which contains:

- Source codes (Python, R, Java)
- Final decrypted outputs (where available)
- Screenshots, logs, and failed trial outputs
- A README file summarizing all attempts and usage instructions

Drive link for the Files.

https://drive.google.com/drive/folders/1eK2uQxj5wQoUIWRqOS695KVmC8ybR1Rl?usp=share_link

Note: All files are shared for educational and peer review purposes only. Please reference appropriately if reusing for future academic work.