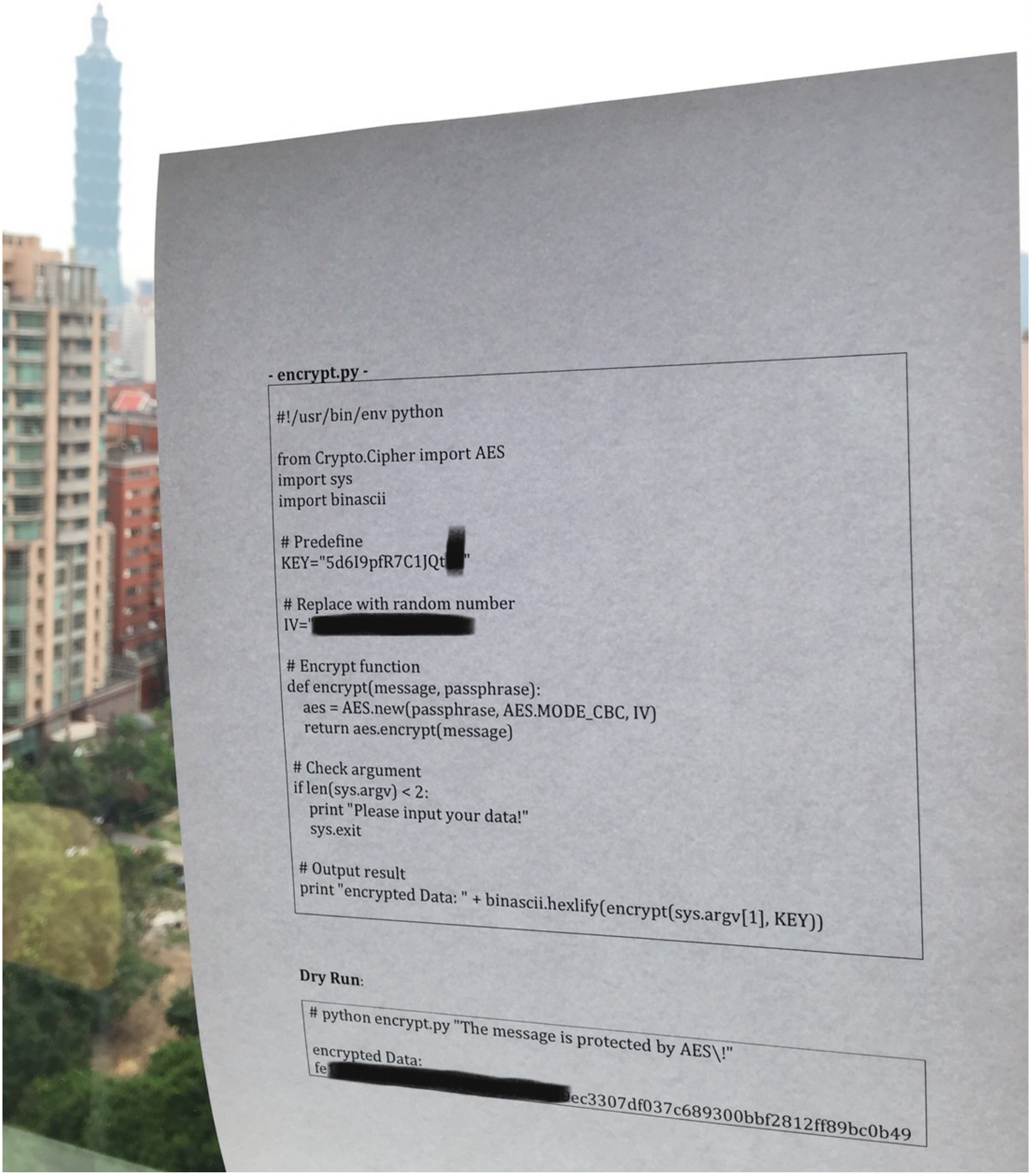
```
iv_result = ""
encrypted = "fe1199011d45c87d10e9e842c1949ec3"
for position in range(16):
        iv = list(IV)
        for missing in range(0, 256):
                iv[position] = chr(missing) # IV with single byte set to tested value
                decrypted = decrypt(real_key, "".join(iv), binascii.unhexlify(encrypted))
                if decrypted[position] == message[position]:
                        print("%d %d" % (position, missing))
                        iv_result += chr(missing)
print(iv_result)
```

Co daje nam `Key:rVFvN9KLeYr6` więc zgodnie z treścią zadania flagą jest `TMCTF{rVFvN9KLeYr6}`

ENG Version

The task was to recover initialization vector IV for AES cipher based on knowledge of the message, part of the key and part of ciphertext. The data were proviede as a photo of crossed-out code:



```
- encrypt.py -
#!/usr/bin/env python

from Crypto.Cipher import AES
import sys
import binascii

# Predefine
KEY="5d6I9pfR7C1JQt[redacted]"

# Replace with random number
IV="[redacted]

# Encrypt function
def encrypt(message, passphrase):
    aes = AES.new(passphrase, AES.MODE_CBC, IV)
    return aes.encrypt(message)

# Check argument
if len(sys.argv) < 2:
    print "Please input your data!"
    sys.exit

# Output result
print "encrypted Data: " + binascii.hexlify(encrypt(sys.argv[1], KEY))
```

Dry Run:

```
# python encrypt.py "The message is protected by AES\!"

encrypted Data:
fe[redacted]ec3307df037c689300bbf2812ff89bc0b49
```

From this we can get:

- Part of the key: `5d6I9pfR7C1JQt` with missing 2 bytes
- Message: `The message is protected by AES!`
- Part of ciphertext: `fe000000000000000000000000009ec3307df037c689300bbf2812ff89bc0b49` (0s in the first block are missing part)

First step, after reading about AES in given configuration, was to extract the whole ciper key. It is worth noting that our message is separated into 2 blocks for this cipher, each with 16 bytes:

```
The message is p
rotected by AES!
```

And the cipher works on blocks, so our ciphertext can also be split into blocks:

```
fe000000000000000000000000009ec3
307df037c689300bbf2812ff89bc0b49
```

For encoding the first block AES uses IV vector and the key, but to encode second block only previous block and the key is used. On top of that the cipher works byte-by-byte which means that deciphering 1 byte of 2 block requires knowledge only of the key and of the 1 byte of 1 block.

It means that for input:

```
XX000000000000000000000000000000
YY000000000000000000000000000000
```

Deciphering usign a proper key will give us properly decoded 16th byte (counting from 0), regardless of IV vector used. Therefore, we test all possible values for the missing 2 key characters, testing for which of them the decipered text has proper values in the second block on the positions where in the first block we have proper values (first byte and last two bytes):

```
KEY = "5d6I9pfR7C1JQt"
IV = "0000000000000000"

def valid_key(correct_bytes, decrypted):
for byte_tuple in correct_bytes:
    if decrypted[byte_tuple[0]] != byte_tuple[1]:
        return False
return True

def break_key(key_prefix, encoded_message_part, correct_bytes):
        final_key = ""
        encrypted = encoded_message_part
        for missing1 in range(0, 256):
                key = key_prefix + chr(missing1)
                for missing2 in range(0, 256):
                        real_key = key + chr(missing2)
                        decrypted = decrypt(real_key, IV, binascii.unhexlify(encrypted))
                        if valid_key(correct_bytes, decrypted):
                                final_key = real_key
        return final_key

real_key = break_key(KEY, "fe000000000000000000000000009ec3307df037c689300bbf2812ff89bc0b49", [(16, "r"), (30, "S"),
```

This way we get the key: `5d6I9pfR7C1JQt7$`

IV vector we are looking for is used to encode 1 block and it is used on the same principle as encoding next blocks decribed above - encoded 1 byte of 1 block depends on 1 byte of 1 block of IV vector, 2 depends on 2 etc. Therefore, to be able to get the IV vector we need to know the whole first encoded block. To get it we use a very similar approach as the one we used to get the key, but this time we test bytes of the encoded 1 block, checking which value after decoding gives us properly decoded byte from 2 block:

```
IV = "0000000000000000"
message = "The message is protected by AES!"
ciphertext = ""
```

```
encrypted = "00000000000000000000000000000000307df037c689300bbf2812ff89bc0b49"
data = binascii.unhexlify(encrypted)
for position in range(16): # going through first block
        encrypted_sub = list(data)
        for missing in range(0, 256):
                encrypted_sub[position] = chr(missing) #encrypted message with single byte in first block set to test
                decrypted = decrypt(real_key, IV, "".join(encrypted_sub))
                if decrypted[position + 16] == message[position + 16]:
                        print("%d %d" % (position, missing))
                        print(decrypted[position + 16])
                        ciphertext += chr(missing)
print(binascii.hexlify(ciphertext))
```

Which gives us: `fe1199011d45c87d10e9e842c1949ec3` and this is the encoded 1 block.

Last step is to recover IV vector. We use the same principle, this time testing IV vector bytes, checking when deciphering gives us properly decoded values from 1 block:

```
iv_result = ""
encrypted = "fe1199011d45c87d10e9e842c1949ec3"
for position in range(16):
        iv = list(IV)
        for missing in range(0, 256):
                iv[position] = chr(missing) # IV with single byte set to tested value
                decrypted = decrypt(real_key, "".join(iv), binascii.unhexlify(encrypted))
                if decrypted[position] == message[position]:
                        print("%d %d" % (position, missing))
                        iv_result += chr(missing)
print(iv_result)
```

Which gives us: `Key:rVFvN9KLeYr6` so according to the task rules the flag is `TMCTF{rVFvN9KLeYr6}`