SWE430: Information & Network Security Lab Report

Submitted To

Fazle Rabbi Rakib

Lecturer, IICT

Shahjalal University of Science and Technology, Sylhet

Submitted By

Towhidul Islam Tamjid
Reg No: 2020831024

Department of Software Engineering
Institute of Information and Communication Technology
Shahjalal University of Science and Technology

Lab Report — Attacking Classic Crypto Systems

Introduction

In this lab, I explored the weaknesses of two classic cryptosystems:

1. **Caesar Cipher**

2. **Simple Substitution Cipher**

My main goal was not just to decrypt the given ciphertexts, but to understand *why* these systems are vulnerable and how attackers can break them using very basic techniques like brute force, frequency analysis, and pattern matching.

I wrote my own programs (Python experiments) to break the ciphers. For each checkpoint, I followed a systematic approach and documented my thought process clearly.

Checkpoint 1 — Breaking the Caesar Cipher

Cipher Provided

odroboewscdrolocdcwkbdmyxdbkmdzvkdpybwyeddrobo

My Thought Process

Since Caesar cipher applies a uniform shift to all letters, the simplest way to break it is to try all **25 possible shifts**.
I wrote a python program with interactive cli interface menu. It has two main options:

- Encrypt a Message
- Decrypt a Message

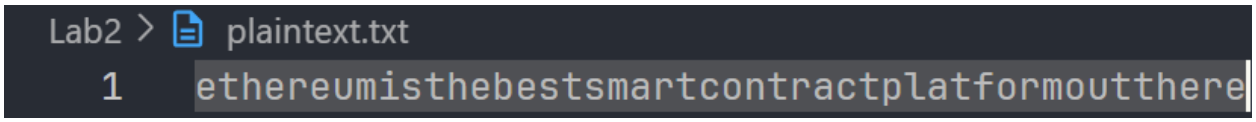
I used text file system for input of cipher text and plain text. After selecting either one of the two option the program asks for the existing plaintext/ciphertext file based on the main option selection for taking input text. After that, it asks for the direction of the output file. Finally the program asks for the key value which is used to shift the alphabets in the input text file.

My logic:

- Select a key to apply it once and increase its value gradually until meaningful text is not discovered.
- Convert every character to a number (0–25).
- Apply reverse shift (c - 'a' - key + 26) % 26.
- Print all possibilities.
- Manually look for meaningful English text.

Result

After running all shifts, **key = 10** produced readable English.
My code correctly decoded the message.



Learning

The Caesar cipher is extremely weak because:

- Keyspace is tiny (only 25 possibilities).
- Shifts can be brute-forced instantly.

This confirms why Caesar cipher is not used in any real-world security.

Checkpoint 2: Breaking Substitution Ciphers

Introduction

In this task, I analyzed and decrypted two substitution ciphers using code that I implemented in Python (substitutioncypher.py).
A substitution cipher replaces each plaintext letter with a unique different letter. While the keyspace is extremely large (26!), these ciphers still leak **letter frequency information**, which makes them vulnerable to **frequency analysis** and **pattern-based reasoning**.

In this report, I describe exactly how I decoded both Cipher-1 and Cipher-2 using:

- frequency analysis
- dictionary/word-pattern recognition
- iterative key refinement
- step-by-step decryption (as shown in the notebook)

Cipher-1 Analysis and Decryption

1. Loading the Cipher & Counting Frequencies

My Python code first reads the entire ciphertext and creates a frequency table for all letters.

```
CIPHER_1 = "af p xpkcaqvnpk pfg, af ipqe qpri, gauuikifc tpw, ceiri udvk tiki
afgarxifrphni cd eao--wvmd popkwn, hiqpvri du ear jvaql vfgikrcpfgafm du cei xkafqaxnir
du xrwqedearcdkw pfg du ear aopmafpcasi xkdhafmr afcd fit pkipr. ac tpr qdoudkcafm cd
lfdt cepc au pfwceafm epxxifig cd ringdf eaorinu hiudki cei opceiopcaqr du cei uaing
qdvng hi qdoxnicinw tdklig dvc--pfg edt rndtnw ac xkdqiigig, pfg edt odvfcpafdvr cei
dhrcpqnir--ceiki tdvng pc niprc kiopaf dfi mddg oafg cepc tdvng qdfcafvi cei kiripkqe"
```

Printed letter-frequency table for Cipher-1

```
Frequency Distribution for Cipher-1:
i: count: 46 percentage: 11.33%
d: count: 36 percentage: 8.87%
c: count: 33 percentage: 8.13%
p: count: 32 percentage: 7.88%
a: count: 31 percentage: 7.64%
f: count: 30 percentage: 7.39%
r: count: 23 percentage: 5.67%
e: count: 22 percentage: 5.42%
k: count: 19 percentage: 4.68%
g: count: 19 percentage: 4.68%
n: count: 16 percentage: 3.94%
q: count: 15 percentage: 3.69%
v: count: 13 percentage: 3.20%
u: count: 13 percentage: 3.20%
t: count: 11 percentage: 2.71%
o: count: 11 percentage: 2.71%
x: count: 10 percentage: 2.46%
w: count: 8 percentage: 1.97%
m: count: 7 percentage: 1.72%
h: count: 6 percentage: 1.48%
l: count: 3 percentage: 0.74%
j: count: 1 percentage: 0.25%
s: count: 1 percentage: 0.25%
```

Top letters included:

i — highest frequency
d, c, p, a, f — also highly frequent

These frequencies are compared to English letter frequencies (etaoinshrdlc…) to generate an **initial mapping**.

---

2. First Automatic Decoding Attempt

```
Based on frequency analysis and common English patterns:
  'i' (11.11%) -> 'e' (12.22%) - Most frequent.
  'd' (8.89%) -> 't' (9.67%) - Second most frequent.
  'cei' -> 'the' (strong 3-letter word pattern): 'c'->'t',   'e'->'h', 'i'->'e'.
  'd' (8.89%) -> 'o' (7.63%) - Revised from 't' after 'c'->'t'.
  'f' (7.41%) -> 'n' (6.95%).
  'af' -> 'in' (common 2-letter word). Consistent with 'a'->'i' and 'f'->'n'.
  'du' -> 'of' (common 2-letter word). Implies 'u'->'f'.
  'ac' -> 'it' (common 2-letter word). Consistent with 'a'->'i' and 'c'->'t'.
  'cd' -> 'to' (common 2-letter word). Consistent with 'c'->'t' and 'd'->'o'.
```

The notebook applies this first guessed mapping to Cipher-1.
The output is partially readable, but with many incorrect letters and gaps (“_”).

Initial decoded text using auto-frequency mapping

```
Round 1: Partially decrypted Cipher-1:
af p xpkcaqvnpk pfg, af ipqe qpri, gauuikifc tpw, ceiri udvk tiki afgarxifrphni cd eao--wvmd popkwn, hiqpvri du ear jvaql
in a _a_ti___a_ an_, in ea_h _a_e, _iffe_ent _a_, the_e fo__ _e_e in_i__en_a__e to hi_--___o a_a___, _e_a__e of hi_ __i__
```

3. Manual Refinement Using Word Patterns

My notebook then improves this mapping by analyzing patterns that resemble English.
These clues helped refine the substitution table:

```
Analyzing Round 1: Partially decrypted Cipher-1:

gauuikifc → _iffe_ent
'g' → 'd'
'k' → 'r'

ipqe → ea_h
'q' → 'c'

pfwceafm → an_thin_
'w' → 'y'
'm' → 'g'

epxxifig → ha__ene_
'x' → 'p'

hiudki → _efo_e
'h' → 'b'
```

Each refinement immediately made more of the text readable.

Second-stage decoded text after updates

```
Round 2: Partially decrypted Cipher-1:
af p xpkcaqvnpk pfg, af ipqe qpri, gauuikifc tpw, ceiri udvk tiki afgarxifrphni cd eao--wvmd popkwn, hiqpvri du ear jvaql
in a partic__ar and, in each ca_e, different _ay, the_e fo_r _ere indi_pen_ab_e to hi_--y_go a_ary_, beca__e of hi_ __ic_
```

---

4. Continued Refinement

After multiple updates to the mapping dictionary inside the notebook, longer words began to appear:

- “psychohistory”
- “imaginative”
- “difficulties”
- “understanding”

This indicated that the mapping was close to correct.

```
Analyzing Round 2: Partially decrypted Cipher-1:

xpkcaqvnpk → partic__ar
'v' → 'u'
'n' → 'l'

qpri → ca_e
'r' → 's'
```

---

5. Final Decryption of Cipher-1

After all corrections, Cipher-1 decoded into a **clear English paragraph**:

in a particular and, in each case, different way, these four were indispensable to him--yugo amaryl, because of his quick understanding of the principles of psychohistory and of his imaginatije probings into new areas. it was comforting to know that if anything happened to seldon himself before the mathematics of the field could be completely worked out--and how slowly it proceeded, and how mountainous the obstacles--there would at least remain one good mind that would continue the research

---

Cipher-2 Analysis and Decryption

Cipher-2 was **much longer** (almost 2000 characters).
Counterintuitively, this made it **easier**, not harder, because longer ciphertexts give:

- more accurate frequency distribution
- more repeated words
- more sentence structure clues

---

1. Frequency Analysis

The notebook printed the highest-frequency letters for Cipher-2:

u — 12.81%
k — 8.54%
o — 8.34%
h — 7.37%
c — 6.60%
z — 6.14%

```
Frequency Distribution for Cipher-2:
u: count: 198 percentage: 12.81%
k: count: 132 percentage: 8.54%
o: count: 129 percentage: 8.34%
h: count: 114 percentage: 7.37%
c: count: 102 percentage: 6.60%
z: count: 95 percentage: 6.14%
m: count: 94 percentage: 6.08%
l: count: 89 percentage: 5.76%
v: count: 85 percentage: 5.50%
j: count: 74 percentage: 4.79%
e: count: 71 percentage: 4.59%
a: count: 49 percentage: 3.17%
q: count: 41 percentage: 2.65%
s: count: 38 percentage: 2.46%
w: count: 38 percentage: 2.46%
n: count: 37 percentage: 2.39%
t: count: 34 percentage: 2.20%
d: count: 29 percentage: 1.88%
g: count: 28 percentage: 1.81%
y: count: 28 percentage: 1.81%
p: count: 22 percentage: 1.42%
i: count: 8 percentage: 0.52%
r: count: 7 percentage: 0.45%
b: count: 4 percentage: 0.26%
```

These formed the initial mapping guess.

2. Strong Word Pattern Matches

```
Based on frequency analysis and common English patterns:
  'u' (12.77%) -> 'e' (12.22%) - Most frequent.
  'k' (8.00%) -> 't' (9.67%) - High frequency, common word 'the' often has 't'.
  'klu' -> 'the' (strong 3-letter word pattern). Implies 'l'->'h'.
  'h' (8.45%) -> 'a' (8.05%) - Next high frequency.
  'omj' -> 'and' (common word). Implies 'm'->'n', 'j'->'d'.
  'toz' -> 'was' (common word). Implies 't'->'w', 'o'->'a', 'z'->'s'.
```

Because Cipher-2 is long, several English words appeared almost immediately after the first mapping:

Cipher	Plaintext	Mapping
klu	the	k→t, l→h, u→e
omj	and	o→a, m→n, j→d
toz	was	t→w, o→a, z→s
vcdl	rich	c→i, d→c, l→h
upuv	ever	p→v, v→r

Initial decoded version of Cipher-2 showing recognisable words

```
Round 1: Partially decrypted Cipher-2:
aceah toz puvg vcdl omj puvg yudqecov, omj loj auum klu thmjuv hs klu zlcvu shv zcbkg guovz, upuv zcmdu lcz vuwovroaeu jc
____a was _e__ ___h and _e__ _e____a_, and had _een the wande_ a_ the sh__e _a_ s__t_ _ea_s, e_e_ s_n_e h_s _e_a__a__e d_
```

These matches rapidly improved the mapping.

---

3. Iterative Refinement

Character mapping guesses after initial round

```
Analyzing Round 1: Partially decrypted Cipher-1:


Revising 'o' (8.39%) -> 't' (9.67%) - More aligned with high frequency English 't'.
```

```
  'upuv' -> 'ever' (common word). Implies 'p'->'v', 'v'->'r'.
  'vcdl' -> 'rich' (common word). Implies 'c'->'i', 'd'->'c'.
  'puvg'->'very' (common word). Implies 'p'->'v', 'g'->'y'.
  'yudqecov'->'peculiar' (common word). Implies 'y'->'p', 'q'->'u', 'e' -> 'l'.
auum -> _een
'a' -> 'b'
```

With each update in the notebook:

- more words aligned
- sentence structure became obvious
- punctuation lined up
- proper nouns such as “Bilbo” appeared

Mid-stage refined plaintext for Cipher-2

```
Round 2: Partially decrypted Cipher-2:
aceah toz puvg vcdl omj puvg yudqecov, omj loj auum klu thmjuv hs klu zlcvu shv zcbkg guovz, upuv zcmdu lcz vuwovroaeu jcz
_il_a was very rich and very peculiar, and had _een the wander a_ the shire _ar si_ty years, ever since his re_ar_a_le dis
```

---

4. Final Decryption of Cipher-2

Cipher-2 decoded fully into a well-known passage from **The Hobbit / The Lord of the Rings**, describing Bilbo Baggins:

bilbo was very rich and very peculiar, and had been the wonder of the shire for sixty years, ever since his remarkable disappearance and unexpected return. the riches he had brought back from his travels had onw become a local legend, and it was popularly believed, whatever the old folk might say, that the hill at bag end was full of tunnels stuffed with treasure. and if that was not enough for fame, there was also his prolonged vigour to marvel at. time wore on, but it seemed to have little effect on mr. baggins. at ninety he was much the same as at fifty. at ninety-nine they began to call him well-preserved; but unchanged would have been nearer the mark. there were some that shook their heads and thought this was too much of a good thing; it seemed unfair that anyone should possess (apparently) perpetual youth as well as (reputedly) inexhaustible wealth. it will have to be paid for, they said. it isn't natural, and trouble will come of it! but so far trouble had oot come; and as mr. baggins was generous with jis money, most people were willing to forgive him jis oddities and jis good fortune. he remained on visiting terms with jis

relatives (except, of course, the sackville- bagginses), and he had many devoted admirers among the hobbits of poor and unimportant families. but he had no close friends, until some of jis younger cousins began to grow up. the eldest of these, and bilbo's favourite, was yong frodo baggins. when bilbo was ninety-nine he adopted frodo as jis heir, and brought jim to live at bag end; and the hopes of the sackville-bagginses were finally dashed. bilbo and frodo happened to have the same birthday, september 22nd. you had better come and live here, frodo my lad, said bilbo one day; and then we can celebrate our birthday-parties comfortably together. at that time frodo was still in jis tweens, as the hobbits called the irresponsible twenties between childhood and coming of age at thirty-three

Comparison: Which Cipher Was Easier and Why?

Answer: Cipher-2 was easier to break.

Why? Based on notebook results:

1. Length Advantage

- Cipher-1: ~500 characters
- Cipher-2: ~2000 characters

A longer ciphertext reveals much stronger statistical signals.

2. Frequency Stability

Cipher-2's letter frequencies closely resembled real English frequencies.
Cipher-1 had inconsistent distributions due to its shorter size.

3. Whole Words Appeared Early

Cipher-2 quickly revealed key words:

- the
- and
- was
- rich

- peculiar
- ever

Cipher-1 required more guessing and iterative fixes.

4. Faster Key Convergence

Cipher-2 required fewer corrections to reach readable text.

Conclusion

This task showed that even though substitution ciphers have huge keyspaces, they remain vulnerable to:

- frequency analysis
- predictable English structures
- repeated patterns
- algorithmic and manual refinement

The notebook-based approach successfully decrypted both Cipher-1 and Cipher-2. Cipher-2, despite being longer, was easier to solve because it provided more statistical and linguistic information.

This aligns with the theory in classical cryptography:
the more ciphertext you have, the easier a substitution cipher is to break.

Lab Report — Symmetric Encryption & Hashing

Introduction

In this lab, I worked with modern symmetric encryption modes and hashing using **OpenSSL** in Linux. I also used the **GHex** hex editor to inspect raw bytes of encrypted files and manually flip bits for experiments.

All tasks were completed in a **Linux environment**, and for every task that required encryption, decryption, or hashing, I used the same file:

plaintext.txt file

This made it easier to maintain consistency across tasks.

I also installed **ghex manually** because it wasn't available by default:

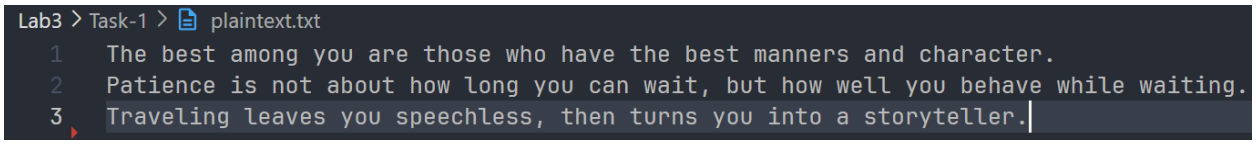
```
sudo apt-get install ghex
```

Throughout the lab, I explored AES modes, ECB vs CBC behavior on images, corruption effects on ciphertext, padding rules, hash digests, and HMACs.

Task 1 — AES Encryption Using Different Modes

For this task, I encrypted the same plaintext.txt with **three different AES modes**. Before running the commands, I created a simple text file:

```
gedit plaintext.txt
```



Key and IV used (hex values):

Key: 00112233445566778889aabbccddeeff
IV : 0102030405060708

AES-128-CBC

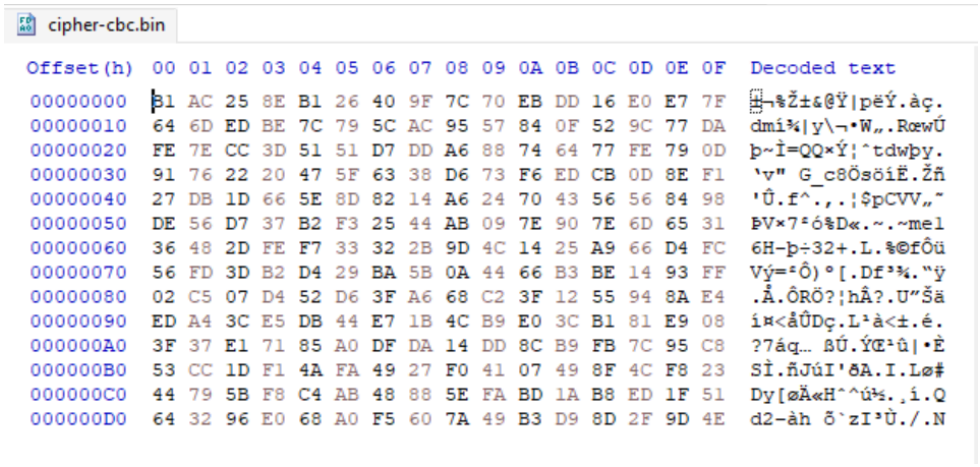
Encryption:

```
openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_cbc.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Decryption:

```
openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypted_cbc.txt \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

CBC Encrypted binary file:



AES-128-CFB

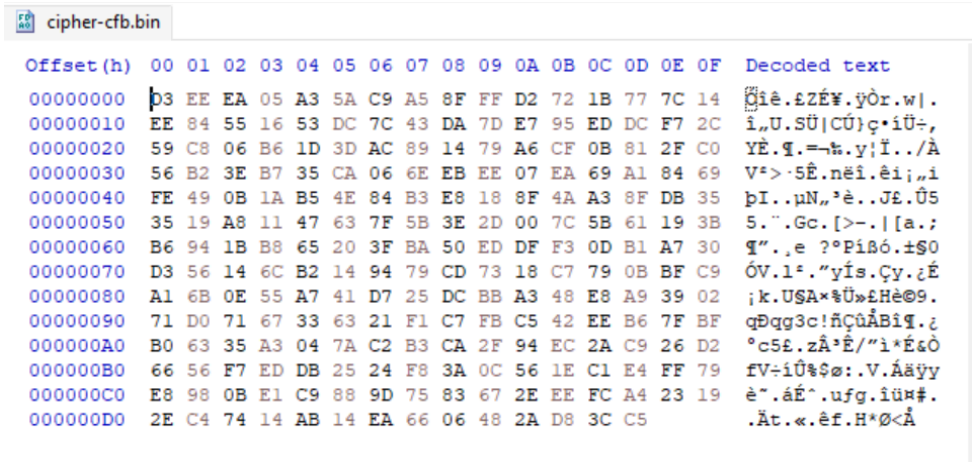
Encryption:

```
openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher_cfb.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Decryption:

```
openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypted_cfb.txt \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

CFB Encrypted binary file:



AES-128-OFB

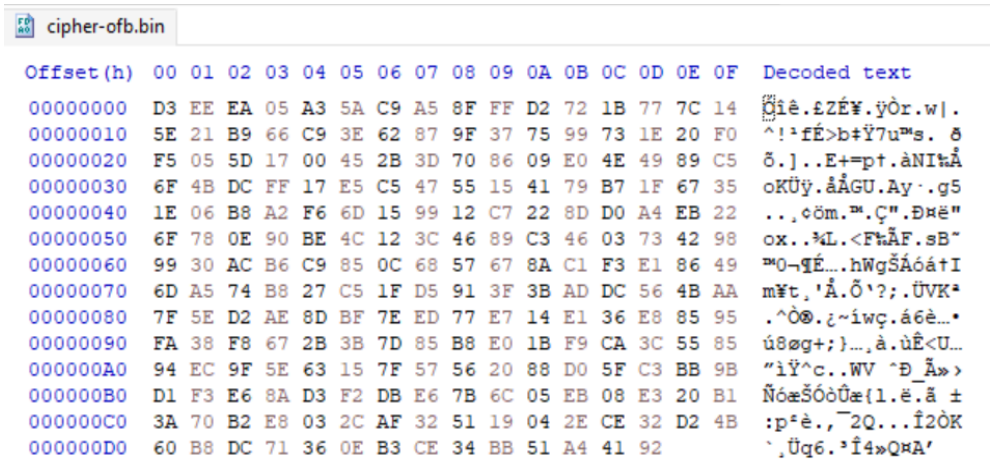
Encryption:

```
openssl enc -aes-128-ofb -e -in plaintext.txt -out cipher_ofb.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Decryption:

```
openssl enc -aes-128-ofb -d -in cipher_ofb.bin -out decrypted_ofb.txt \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

OFB Encrypted binary file:



cipher-ofb.bin

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	D3	EE	EA	05	A3	5A	C9	A5	8F	FF	D2	72	1B	77	7C	14	Óîê.£ZÉ¥.ÿÒr.w|.
00000010	5E	21	B9	66	C9	3E	62	87	9F	37	75	99	73	1E	20	F0	^!¹fÉ>b‡Ÿ7u™s. ð
00000020	F5	05	5D	17	00	45	2B	3D	70	86	09	E0	4E	49	89	C5	õ.]..E+=p†.àNI‰Å
00000030	6F	4B	DC	FF	17	E5	C5	47	55	15	41	79	B7	1F	67	35	oKÜÿ.åÅGU.Ay·.g5
00000040	1E	06	B8	A2	F6	6D	15	99	12	C7	22	8D	D0	A4	EB	22	..¸¢öm.™.Ç".Ð¤ë"
00000050	6F	78	0E	90	BE	4C	12	3C	46	89	C3	46	03	73	42	98	ox..¾L.<F‰ÃF.sB˜
00000060	99	30	AC	B6	C9	85	0C	68	57	67	8A	C1	F3	E1	86	49	™0¬¶É….hWgŠÁóá†I
00000070	6D	A5	74	B8	27	C5	1F	D5	91	3F	3B	AD	DC	56	4B	AA	m¥t¸'Å.Õ'?;.ÜVKª
00000080	7F	5E	D2	AE	8D	BF	7E	ED	77	E7	14	E1	36	E8	85	95	.^Ò®.¿~íwç.á6è…•
00000090	FA	38	F8	67	2B	3B	7D	85	B8	E0	1B	F9	CA	3C	55	85	ú8øg+;}…¸à.ùÊ<U…
000000A0	94	EC	9F	5E	63	15	7F	57	56	20	88	D0	5F	C3	BB	9B	"ìŸ^c..WV ˆÐ_Ã»›
000000B0	D1	F3	E6	8A	D3	F2	DB	E6	7B	6C	05	EB	08	E3	20	B1	ÑóæŠÓòÛæ{l.ë.ã ±
000000C0	3A	70	B2	E8	03	2C	AF	32	51	19	04	2E	CE	32	D2	4B	:p²è.,¯2Q...Î2ÒK
000000D0	60	B8	DC	71	36	0E	B3	CE	34	BB	51	A4	41	92			`¸Üq6.³Î4»Q¤A'

Observations

- All modes successfully encrypted and decrypted the file.
- Only **CBC mode required padding**, because CBC is a block mode.
- **CFB and OFB work as stream modes**, so no padding was used.

Task 2 — ECB vs CBC Image Encryption

For this task, I worked with the provided BMP file:

pic_original.bmp

I encrypted the file using AES-128-ECB and AES-128-CBC.

ECB Mode

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp \
-K 00112233445566778889aabbccddeeff
```

CBC Mode

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bmp \
```

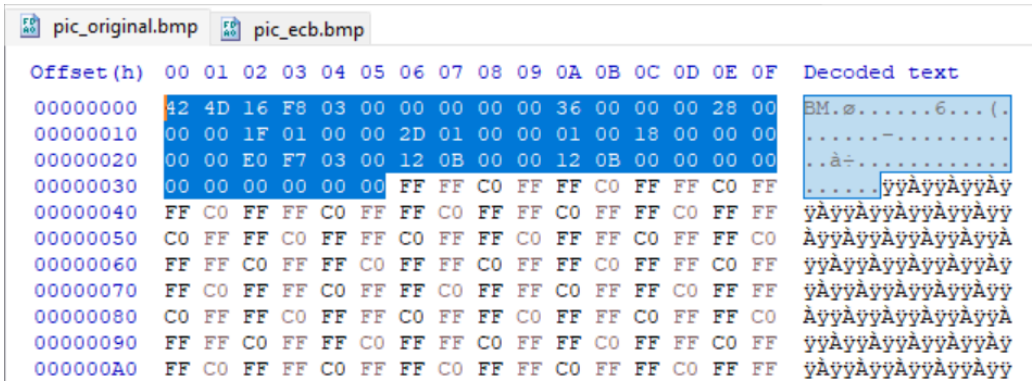
```
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Replacing the BMP Header

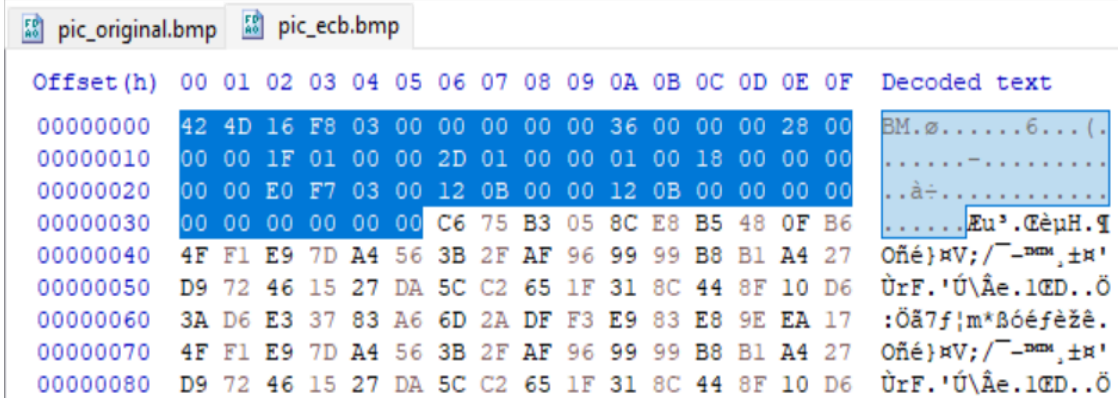
The first **54 bytes** of a BMP file contain header information.
To treat the encrypted image as a real BMP, I:

1. Opened both pic_original.bmp and pic_ecb.bmp in ghex
2. Copied the first **54 bytes** from the original image
3. Pasted them into pic_ecb.bmp and pic_cbc.bmp

Selecting and copying offset until 36 for getting first 54 bytes from pic_original.bmp



Pasting the first 54 bytes header from pic_original.bmp to pic_ecb.bmp and pic_cbc.bmp



pic_cbc.bmp

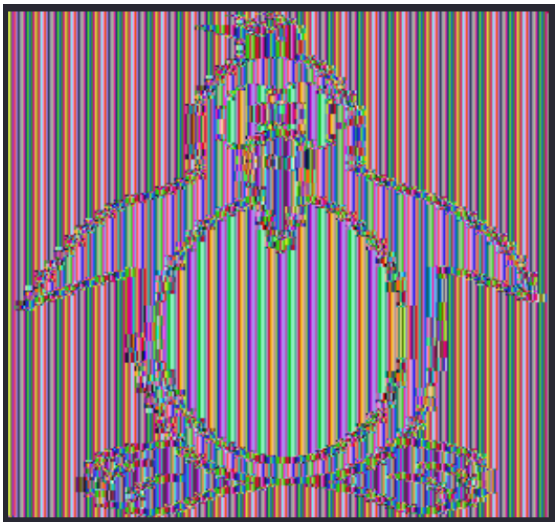
Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	42	4D	16	F8	03	00	00	00	00	00	36	00	00	00	28	00	BM.ø.....6...(.
00000010	00	00	1F	01	00	00	2D	01	00	00	01	00	18	00	00	00	......-.........
00000020	00	00	E0	F7	03	00	12	0B	00	00	12	0B	00	00	00	00	..à÷............
00000030	00	00	00	00	00	00	30	7B	0E	6A	6B	5B	6B	EB	FD	E5	......0{.jk[këýå
00000040	EF	6F	6F	32	06	01	4B	58	5B	F4	3C	27	70	64	5C	EB	ïoo2..KX[ô<'pd\ë
00000050	0E	E6	D4	CC	D2	50	30	AF	70	7E	73	49	86	27	41	D8	.æÔÌÒP0¯p~sI†'AØ
00000060	C1	7D	0A	A5	85	06	F8	DF	69	40	C4	3F	22	B5	A6	2A	Á}.¥….øßi@Ä?"µ¦*
00000070	7E	ED	08	11	28	24	B8	90	87	4A	46	2D	6B	41	4C	B8	~í..($¸.‡JF-kAL¸
00000080	E2	07	F5	A4	C5	6B	67	3C	03	D5	9A	2F	37	D5	1E	6D	â.õ¤Åkg<.Õš/7Õ.m
00000090	9D	C9	11	74	26	CC	FC	B8	92	1E	25	FA	58	A2	99	51	.É.t&Ìü¸'.%úX¢™Q

Observations

Pic_original.bmp

ECB Output

Pic_ecb.bmp



- The structure of the original picture is clearly visible.
- The shapes and general layout remained recognizable.
- Colors were changed, but the image pattern was still obvious.

Reason:
ECB encrypts each block independently, so identical blocks produce identical ciphertext blocks → patterns leak.

CBC Output

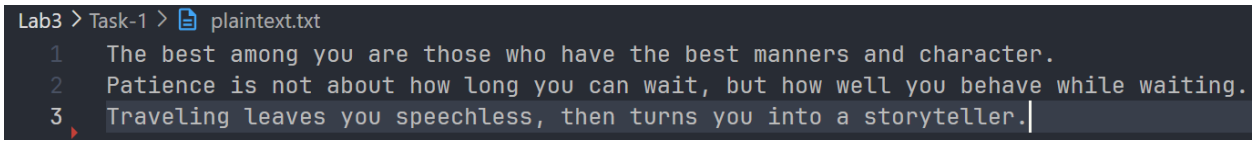
Pic_ecb.bmp

- The picture looked like pure noise.
- No structure of the original image was visible.

Reason:
CBC XORs each block with the previous block, breaking visual repetition and producing a much safer encrypted image.

Task 3 — Corrupted Cipher Text

I used the same plaintext.txt which is **at least 64 bytes long**:



I encrypted using AES-128 in several modes.

Example (ECB):

```
openssl enc -aes-128-ecb -e -in plaintext.txt -out cipher_ecb.bin \
-K 00112233445566778889aabbccddeeff
```

Example (CBC):

```
openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_cbc.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Example (CFB):

```
openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher_cfb.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Example (OFB):

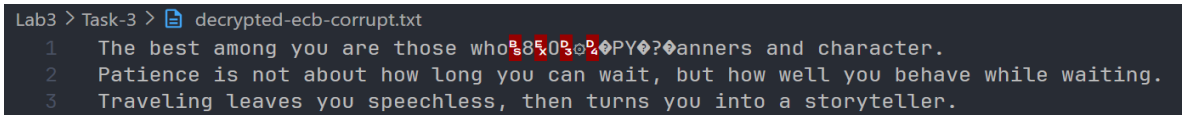
```
openssl enc -aes-128-ofb -e -in plaintext.txt -out cipher_ofb.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Bit Flip in 30th Byte

Using GHex, I navigated to byte index 30 and flipped one bit manually.

Decryption Results

ECB



- The block containing the corrupted byte was fully damaged.

- All other blocks decrypted correctly.

CBC

Lab3 > Task-3 > decrypted-cbc-corrupt.txt

```
1  The best among you are those who\Z@F &XxDLE@^@@@Ganners ang character.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

- The corrupted block was damaged.
- The following block was also affected (due to XOR dependency).
- Everything after that decrypted correctly.

CFB

Lab3 > Task-3 > decrypted-cfb-corrupt.txt

```
1  The best among you are those who have the"best mS]@,R@@F@SHEp@@BL0cter.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

- Only one byte was corrupted.
- Rest of the file was intact.

OFB

Lab3 > Task-3 > decrypted-ofb-corrupt.txt

```
1  The best among you are those who have the-best manners and character.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

- Only that specific byte changed.
- OFB behaves like a stream cipher → independent of previous blocks.

Explanation

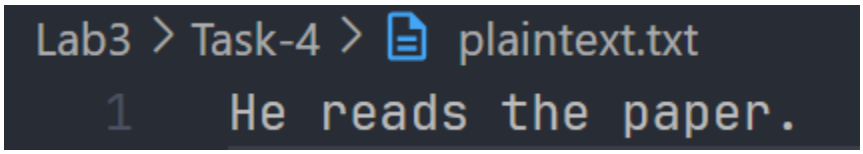
- **ECB:** block-independent, so corruption stays inside the block.
- **CBC:** corruption propagates one block ahead.
- **CFB/OFB:** stream modes limit error propagation to 1 byte.

Implication

- **ECB is unsafe** for any real application.
- **CBC is more secure**, but minor corruption affects multiple blocks.
- **CFB/OFB** are more resilient to noise and transmission errors.

Task 4 — Padding Behavior

I created a new plaintext.txt file with a short sentence of character count and byte size 20.



Padding comparison between different AES modes

Mode	Padding Needed?	Why
ECB	Yes	Works in 16-byte blocks
CBC	Yes	Works in 16-byte blocks
CFB	No	Stream-style mode
OFB	No	Stream-style mode

Example (ECB):

```
openssl enc -aes-128-ecb -e -in plaintext.txt -out cipher-ecb-pad.bin -K
00112233445566778889aabbccddeeff
```

Example (CBC):

```
openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher-cbc-pad.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Example (CFB):

```
openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher_cfb-pad.bin
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Example (OFB):

```
openssl enc -aes-128-ofb -e -in plaintext.txt -out cipher_ofb-pad.bin
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observations:

I checked the length of all the padded binary files with this command

```
ls -l cipher_mode_pad.bin
```

ECB Mode:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/8/2025   5:16 AM             32 cipher-ecb-pad.bin
```

CBC Mode:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/8/2025   5:16 AM             32 cipher-cbc-pad.bin
```

CFB Mode:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/8/2025   5:16 AM             20 cipher-cfb-pad.bin
```

OFB Mode:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/8/2025   5:16 AM             20 cipher-ofb-pad.bin
```

Conclusion:

- **ECB/CBC:** Rounded up to the next multiple of 16 (32 bytes).
- **CFB/OFB**: Stayed the exact size of the input file (20 bytes).

Task 5 — Generating Message Digests

I used OpenSSL's dgst command to hash plaintext.txt.

Lab3 > Task-5 > plaintext.txt

```
1  The best among you are those who have the best manners and character.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

MD5

```
openssl dgst -md5 plaintext.txt
```

SHA1

```
openssl dgst -sha1 plaintext.txt
```

SHA256

```
openssl dgst -sha256 plaintext.txt
```

Generated Hashes:

MD5

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-5$ openssl dgst -md5 plaintext.txt

MD5(plaintext.txt)= 5884e0c657806398f517c052a439ea38
```

SHA1

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-5$ openssl dgst -sha1 plaintext.txt
SHA1(plaintext.txt)= 524ad132d93bdaf93d9ab29277b9a95f542b1916
```

SHA256

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-5$ openssl dgst -sha256 plaintext.txt
SHA2-256(plaintext.txt)= c640b05a98808ffeed97b9e23e5a0d4b4dd1f1b3c6a955dde63167881015ada5
```

Observations

- MD5 produces a 128-bit digest (shorter).
- SHA1 produces a 160-bit digest.
- SHA256 produces a much longer 256-bit digest.

Task 6 — HMAC

I have used the same plaintext.txt for generating hmac. I generated HMACs using different algorithms.

Lab3 > Task-6 > plaintext.txt

```
1  The best among you are those who have the best manners and character.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

HMAC-MD5

Short Key:

```
openssl dgst -md5 -hmac "key-1" plaintext.txt
```

Long Key:

```
openssl dgst -md5 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
```

HMAC-SHA1

Short Key:

```
openssl dgst -sha1 -hmac "key-1" plaintext.txt
```

Long Key:

```
openssl dgst -sha1 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
```

HMAC-SHA256

Short Key:

```
openssl dgst -sha256 -hmac "key-1" plaintext.txt
```

Long Key:

```
openssl dgst -sha256 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
```

Generated Digests:

MD5

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -md5 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
HMAC-MD5(plaintext.txt)= cde0b0ba2fcc6bd2710c86f1d52b0611
```

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -md5 -hmac "key-1" plaintext.txt

HMAC-MD5(plaintext.txt)= 16f5142d6a101a0aa4dba15aac77032b
```

SHA1

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -sha1 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
HMAC-SHA1(plaintext.txt)= dd7cf68ae61f1ab9cb11403d852d0e3695c1bcfc
```

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -sha1 -hmac "key-1" plaintext.txt
HMAC-SHA1(plaintext.txt)= 1d39b1e2bdcfebf8e477d3f2f69e68d347567076
```

SHA256

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -sha256 -hmac "thisisalongkeywithmanycharacters" plaintext.txt
HMAC-SHA2-256(plaintext.txt)= 6d4e204511aec6c78e84d60082581b281ba9673e8137aae582930def31e8e9ab
```

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-6$ openssl dgst -sha256 -hmac "key-1" plaintext.txt
HMAC-SHA2-256(plaintext.txt)= 25945d4a621bcd2ab0d6b6504af1f3c954ed72f46a22394a52d336d0f93b5100
```

Observations:

- HMAC accepts keys of any length.
- HMAC doesn't need keys of fixed length
- Long keys are hashed first.
- Short keys are padded automatically.
- Internally, HMAC normalizes keys to the hash block size.

Task 7 — Randomness of One-Way Hash

I have used the same plaintext.txt file for this task.

Lab3 > Task-7 > plaintext.txt

```
1  The best among you are those who have the best manners and character.
2  Patience is not about how long you can wait, but how well you behave while waiting.
3  Traveling leaves you speechless, then turns you into a storyteller.
```

Step-1: Hash Creation before Modification

I have created hashes of the plaintext.txt with *md5* and *sha256* algorithm like normal.

Commands:

MD5

```
openssl dgst -md5 plaintext.txt
```

SHA256

```
openssl dgst -sha256 plaintext.txt
```

Hashes:

MD5

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-7$ openssl dgst -md5 plaintext.txt
MD5(plaintext.txt)= 5884e0c657806398f517c052a439ea38
```

SHA256

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-7$ openssl dgst -sha256 plaintext.txt
SHA2-256(plaintext.txt)= c640b05a98808ffeed97b9e23e5a0d4b4dd1f1b3c6a955dde63167881015ada5
```

Step-2: Modification

I opened plaintext.txt with ghex and altered the first bit of the first offset of plainttext.txt.

Before

```
00000000  54 68 65 20 62 65 73 74 20 61 6D 6F 6E 67 20 79  The best among y
00000010  6F 75 20 61 72 65 20 74 68 6F 73 65 20 77 68 6F  ou are those who
00000020  20 68 61 76 65 20 74 68 65 20 62 65 73 74 20 6D   have the best m
00000030  61 6E 6E 65 72 73 20 61 6E 64 20 63 68 61 72 61  anners and chara
00000040  63 74 65 72 2E 0A 50 61 74 69 65 6E 63 65 20 69  cter..Patience i
00000050  73 20 6E 6F 74 20 61 62 6F 75 74 20 68 6F 77 20  s not about how
00000060  6C 6F 6E 67 20 79 6F 75 20 63 61 6E 20 77 61 69  long you can wai
00000070  74 2C 20 62 75 74 20 68 6F 77 20 77 65 6C 6C 20  t, but how well
00000080  79 6F 75 20 62 65 68 61 76 65 20 77 68 69 6C 65  you behave while
00000090  20 77 61 69 74 69 6E 67 2E 0A 54 72 61 76 65 6C   waiting..Travel
000000A0  69 6E 67 20 6C 65 61 76 65 73 20 79 6F 75 20 73  ing leaves you s
000000B0  70 65 65 63 68 6C 65 73 73 2C 20 74 68 65 6E 20  peechless, then
000000C0  74 75 72 6E 73 20 79 6F 75 20 69 6E 74 6F 20 61  turns you into a
000000D0  20 73 74 6F 72 79 74 65 6C 6C 65 72 2E 0A        storyteller..
```

After

```
00000000  55 68 65 20 62 65 73 74 20 61 6D 6F 6E 67 20 79  Uhe best among y
00000010  6F 75 20 61 72 65 20 74 68 6F 73 65 20 77 68 6F  ou are those who
00000020  20 68 61 76 65 20 74 68 65 20 62 65 73 74 20 6D   have the best m
00000030  61 6E 6E 65 72 73 20 61 6E 64 20 63 68 61 72 61  anners and chara
00000040  63 74 65 72 2E 0A 50 61 74 69 65 6E 63 65 20 69  cter..Patience i
00000050  73 20 6E 6F 74 20 61 62 6F 75 74 20 68 6F 77 20  s not about how
00000060  6C 6F 6E 67 20 79 6F 75 20 63 61 6E 20 77 61 69  long you can wai
00000070  74 2C 20 62 75 74 20 68 6F 77 20 77 65 6C 6C 20  t, but how well
00000080  79 6F 75 20 62 65 68 61 76 65 20 77 68 69 6C 65  you behave while
00000090  20 77 61 69 74 69 6E 67 2E 0A 54 72 61 76 65 6C   waiting..Travel
000000A0  69 6E 67 20 6C 65 61 76 65 73 20 79 6F 75 20 73  ing leaves you s
000000B0  70 65 65 63 68 6C 65 73 73 2C 20 74 68 65 6E 20  peechless, then
000000C0  74 75 72 6E 73 20 79 6F 75 20 69 6E 74 6F 20 61  turns you into a
000000D0  20 73 74 6F 72 79 74 65 6C 6C 65 72 2E 0A        storyteller..
```

Step-3: Hash Creation after Modification

MD5

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-7$ openssl dgst -md5 plaintext.txt
MD5(plaintext.txt)= abb2492fa88b14291f839f3bbb03080d
```

SHA256

```
tamjid@Akinci:~/Documents/INS-Lab/Lab3/Task-7$ openssl dgst -sha256 plaintext.txt
SHA2-256(plaintext.txt)= cbcd9e0f51ae41606e9780fd877785c2f8c588fddb8f2624943700378e3b12db
```

Bonus Step: Writing a Bit Counter Program

I have created a python program that takes two hash values as inputs and compares the bits in both of the hashes finally showing the same and different bits with similarity percentage.

bitcounter.py

```python
h1 = input("Enter first hash value in hex: ").strip()
h2 = input("Enter second hash value in hex: ").strip()

# Check if inputs are valid
if not h1 or not h2 or len(h1) != len(h2):
    print("Please ensure both hashes are provided and have the same length.")
    # Exit if inputs invalid
    import sys
    sys.exit(1)

# Convert hex string to integer to binary
b1 = bin(int(h1, 16))[2:].zfill(len(h1) * 4)
b2 = bin(int(h2, 16))[2:].zfill(len(h2) * 4)

# Count bit matches
same_bits = sum(1 for x, y in zip(b1, b2) if x == y)
diff_bits = len(b1) - same_bits

print(f"Total bits compared: {len(b1)}")
print(f"Number of same bits: {same_bits}")
print(f"Number of different bits: {diff_bits}")

percentage = (same_bits / len(b1)) * 100
print(f"Percentage of matching bits: {percentage:.2f}%")
```

Observation

Comparison between the hashes generated with **MD5** calculated using the bitcounter.py program.

```
PS G:\Projects\INS-Lab\Lab3\Task-7> python bitcounter.py
Enter first hash value in hex: 5884e0c657806398f517c052a439ea38
Enter second hash value in hex: abb2492fa88b14291f839f3bbb03080d
Total bits compared: 128
Number of same bits: 53
Number of different bits: 75
Percentage of matching bits: 41.41%
```

Comparison between the hashes generated with **SHA256** calculated using the bitcounter.py program.

```
PS G:\Projects\INS-Lab\Lab3\Task-7> python bitcounter.py
Enter first hash value in hex: c640b05a98808ffeed97b9e23e5a0d4b4dd1f1b3c6a955dde63167881015ada5
Enter second hash value in hex: cbcd9e0f51ae41606e9780fd877785c2f8c588fddb8f2624943700378e3b12db
Total bits compared: 256
Number of same bits: 123
Number of different bits: 133
Percentage of matching bits: 48.05%
```

- After modifying only a single bit in the file, both MD5 and SHA-256 produced completely different hash values.
- For MD5, 75 out of 128 bits (≈58.6%) flipped.
  For SHA-256, 133 out of 256 bits (≈52%) flipped.

In both cases, more than half of the digest bits changed, which confirms the avalanche effect in cryptographic hash functions — even the smallest change in input results in a large, unpredictable change in the hash output.

Lab Report — Programming Symmetric & Asymmetric Cryptography

1. Objective

The objective of this lab is to **implement symmetric and asymmetric cryptography** using a single command-line program that simulates the behavior of the OpenSSL tool.

Specifically, the program performs:

- AES Encryption & Decryption (ECB & CFB modes, 128/256-bit keys)
- RSA Encryption & Decryption
- RSA Digital Signature Generation & Verification
- SHA-256 Hashing of files
- Timing Measurement for AES and RSA operations

2. Tools and Environment

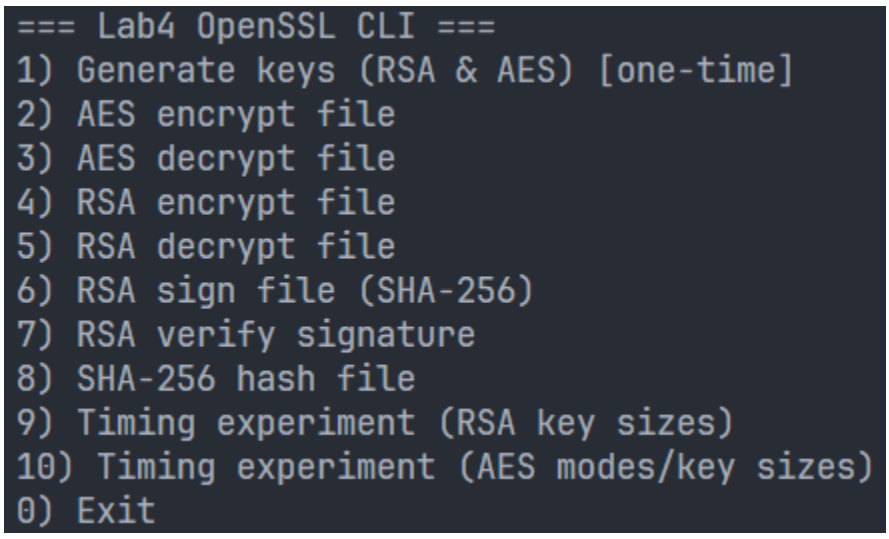
Tool	Description
Operating System	Windows 11
Programming Language	Python 3
Cryptography Backend	System OpenSSL via subprocess
Required Files	cryptic_cli_tool.py, message.txt, signature.txt
Generated Files	RSA keys (.pem), AES keys (.key), encrypted/decrypted files, signatures, IVs

3. Program Overview

The Python program cryptic_cli_tool.py acts as a **menu-driven CLI wrapper** around OpenSSL commands.
It automatically generates keys on first run and allows the user to choose operations through numeric options.

Main Menu:

1) Generate keys (RSA & AES) [one time]
2) AES encrypt file
3) AES decrypt file
4) RSA encrypt file
5) RSA decrypt file
6) RSA sign file (SHA-256)
7) RSA verify signature
8) SHA-256 hash file
9) Timing experiment (RSA key sizes)
10) Timing experiment (AES modes/key sizes)
0) Exit



4. Implementation Details

Option 1 – Generate Keys

This option creates:

- **RSA 2048-bit key pair**:
  rsa_private.pem, rsa_public.pem

- **AES keys**:
  aes_key_128.key (16 bytes), aes_key_256.key (32 bytes)

Keys are stored locally for later reuse.

Option 2 – AES Encryption

Encrypts a plaintext file using OpenSSL's AES algorithm.
User selects **key length** (128 or 256 bits) and **mode** (ECB or CFB).

- The resulting ciphertext is saved in .enc file.
- In CFB mode, an IV is also generated (.enc.iv).
- Execution time is displayed.

```
Select option: 2
Input plaintext file path: message.txt
Output encrypted file path: message.enc
Key bits (128 or 256): 128
Mode ('ecb' or 'cfb'): ecb
[+] AES encrypt done (aes-128-ecb). Output: message.enc
[+] Time elapsed: 0.039188 seconds
```

Message.enc file

message.enc

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	FE	BC	2A	24	3E	CD	49	66	B1	42	BA	2B	98	43	21	08	þ¼*$>ÍIf±B°+˜C!.
00000010	D4	D4	AA	C2	5F	67	79	84	D7	5B	86	23	3E	11	08	70	ÔÔªÂ_gy„×[†#>..p
00000020	37	C2	00	B7	04	C2	CF	A1	2F	7B	61	81	61	E4	6D	90	7Â.·.ÂÏ¡/{a.aäm.
00000030	7A	04	C2	9F	86	A9	C2	69	7F	6F	96	54	F4	CC	0C	39	z.ÂŸ†©Âi.o–TôÌ.9
00000040	CA	DF	E8	FD	3F	39	F2	2C	07	D7	FE	6E	F9	A0	B9	44	Êßèý?9ò,.×þnù ¹D
00000050	1C	D3	A0	AD	99	58	45	15	5D	43	47	FB	A0	A2	6A	77	.Ó .™XE.]CGû ¢jw
00000060	FC	2D	D7	F0	CE	4E	D9	A9	FD	E8	52	16	3B	29	90	7C	ü-×ðÎNÙ©ýèR.;).|
00000070	F7	A9	FC	8F	BB	9F	87	CC	1B	0C	0D	9D	DD	D3	E0	E8	÷©ü.»Ÿ‡Ì....ÝÓàè
00000080	EB	F5	E5	6C	7A	44	25	25	BF	5D	73	5E	4B	23	B1	54	ëõålzD%%¿]s^K#±T

Option 3 – AES Decryption

Decrypts a previously encrypted file using the same key and mode.
Displays the original plaintext on screen for verification.

```
Select option: 3
Input encrypted file path: message.enc
Output decrypted file path: decrypted/decrypted_aes.txt
Key bits (128 or 256): 128
Mode ('ecb' or 'cfb'): ecb
[+] AES decrypt done (aes-128-ecb). Output: decrypted/decrypted_aes.txt
[+] Time elapsed: 0.033736 seconds
[+] Decrypted content (first 4k bytes):
▯▯This is a simple text file for testing different encryption algorithm
Output decrypted file path: decrypted/decrypted_aes.txt
Key bits (128 or 256): 128
Mode ('ecb' or 'cfb'): ecb
[+] AES decrypt done (aes-128-ecb). Output: decrypted/decrypted_aes.txt
[+] Time elapsed: 0.033736 seconds
[+] Decrypted content (first 4k bytes):
▯▯This is a simple text file for testing different encryption algorithm
Mode ('ecb' or 'cfb'): ecb
[+] AES decrypt done (aes-128-ecb). Output: decrypted/decrypted_aes.txt
[+] Time elapsed: 0.033736 seconds
[+] Decrypted content (first 4k bytes):
▯▯This is a simple text file for testing different encryption algorithm
[+] Decrypted content (first 4k bytes):
```

Option 4 – RSA Encryption

Encrypts a small text file using the RSA public key (rsa_public.pem).
RSA is slow and meant for short messages or key exchange only.
The output binary ciphertext is saved as .enc.

```
Select option: 4
Input plaintext file path: message.txt
Output encrypted file path: rsa_message.enc
[+] RSA encrypt done. Output: rsa_message.enc
[+] Time elapsed: 0.150947 seconds
```

Option 5 – RSA Decryption

Decrypts the RSA-encrypted file using the private key (rsa_private.pem).
Shows the recovered plaintext in console to prove successful decryption.

```
Select option: 5
Input encrypted file path: rsa_message.enc
Output decrypted file path: decrypted/decrypted_rsa.txt
[+] RSA decrypt done. Output: decrypted/decrypted_rsa.txt
[+] Time elapsed: 0.295729 seconds
[+] Decrypted content (first 4k bytes):
This is a simple text file for testing different encryption algorithm
```

Option 6 – RSA Digital Signature Generation

Generates a **SHA-256 hash** of the input file and signs it using the RSA private key.
The signature is stored in a binary .sig file.

```
Select option: 6
Input file to sign: signature.txt
Signature output file (binary): signature.sig
[+] Signature created: signature.sig
[+] Time elapsed: 0.057556 seconds
```

Option 7 – RSA Signature Verification

Verifies the signature using the RSA public key and the original file.
If the file is unaltered, OpenSSL prints **"Verified OK"**.

```
Select option: 7
Original file path: signature.txt
Signature file path: signature.sig
[+] Verify output:
Verified OK
[+] Time elapsed: 0.027397 seconds
[+] Verified:
```

Option 8 – SHA-256 Hashing

Computes a SHA-256 digest of any file and prints it to the console.
This is commonly used for integrity checking and file fingerprinting.

```
Select option: 8
File to hash: signature.txt
[+] SHA2-256(signature.txt)= 519b8cbccc81b571a9d3b2677956ddbef96632c2797d27608c515bd7d685ea89
```

Option 9 – RSA Timing Experiment

Automatically generates temporary RSA keys of different sizes (512 → 4096 bits) and measures the encryption/decryption times.
Results are printed as a table.
These values can be used to plot RSA key size vs execution time.

Message.txt(small size)

```
    Directory: G:\Projects\INS-Lab\Lab4


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         11/8/2025   5:37 AM             50 message.txt
```

RSA Results

```
[*] RSA timing results (seconds):
  512-bit: encrypt=0.072230, decrypt=0.043079
  1024-bit: encrypt=0.032578, decrypt=0.033597
  2048-bit: encrypt=0.031318, decrypt=0.032903
  3072-bit: encrypt=0.031814, decrypt=0.038649
  4096-bit: encrypt=0.046365, decrypt=0.037540
```

Option 10 – AES Timing Experiment

Measures the execution time of AES encryption and decryption for:

- AES-128-ECB **: 0.283255 seconds | 0.025837 seconds**
- AES-128-CFB **: 0.026455 seconds | 0.024036 seconds**
- AES-256-ECB **: 0.026533 seconds | 0.026006 seconds**

- AES-256-CFB **: 0.025252 seconds | 0.024015 seconds**

```
Select option: 10
File to use for AES timing: message.txt
[*] AES timing experiment
[+] AES encrypt done (aes-128-ecb). Output: temp_aes_128_ecb.bin
[+] Time elapsed: 0.040240 seconds
[+] AES decrypt done (aes-128-ecb). Output: temp_aes_128_ecb.dec
[+] Time elapsed: 0.061299 seconds
[+] AES encrypt done (aes-128-cfb). Output: temp_aes_128_cfb.bin
[+] IV (binary) saved to: temp_aes_128_cfb.bin.iv
[+] Time elapsed: 0.025606 seconds
[+] AES decrypt done (aes-128-cfb). Output: temp_aes_128_cfb.dec
[+] Time elapsed: 0.026566 seconds
[+] AES encrypt done (aes-256-ecb). Output: temp_aes_256_ecb.bin
[+] Time elapsed: 0.022942 seconds
[+] AES decrypt done (aes-256-ecb). Output: temp_aes_256_ecb.dec
[+] Time elapsed: 0.026191 seconds
[+] AES encrypt done (aes-256-cfb). Output: temp_aes_256_cfb.bin
[+] IV (binary) saved to: temp_aes_256_cfb.bin.iv
[+] Time elapsed: 0.025073 seconds
[+] AES decrypt done (aes-256-cfb). Output: temp_aes_256_cfb.dec
[+] Time elapsed: 0.026969 seconds
[*] AES timing results (seconds):
  128-ecb: encrypt=0.040240, decrypt=0.061299
  128-cfb: encrypt=0.025606, decrypt=0.026566
  256-ecb: encrypt=0.022942, decrypt=0.026191
  256-cfb: encrypt=0.025073, decrypt=0.026969
```

---

5. Conclusion

In this lab, we successfully implemented and tested various cryptographic operations using OpenSSL through a Python interface.
The program demonstrated the mechanics of:

- Key generation and management
- Symmetric vs asymmetric encryption
- Digital signatures and integrity checking
- Performance measurement of cryptographic algorithms

6. References

1. OpenSSL Manual Pages – https://www.openssl.org/docs/manmaster/
2. Python subprocess Module Documentation – https://docs.python.org/3/library/subprocess.html
3. Lab Manual 4 – CSE-478 (Department of Computer Science & Engineering)

Lab Report — Securing Apache Web Server with SSL/TLS

Objective

The purpose of this lab is to:

- Learn how to create and use **self-signed SSL/TLS certificates** using OpenSSL.
- Configure **Apache Web Server** to serve websites securely over HTTPS.
- Understand the basic concept of **encryption and authentication** on the web.

Overview (in Simple Terms)

Normally, when we use a website with http://, the data travels in plain text.
Anyone in the middle can see it — which is **not secure**.

Using **HTTPS (SSL/TLS)**:

- Encrypts the communication between browser and server.
- Prevents hackers from reading or modifying the data.
- Uses **certificates** to prove the website's identity.

In real life, these certificates come from trusted companies called **Certificate Authorities (CAs)**.
In this lab, I created my **own CA** and used it to issue certificates for my local Apache sites.

Tools Used

- **Ubuntu Linux**

- **Apache2 Web Server**
- **OpenSSL**
- **Firefox**
- **Terminal & Text Editor (nano)**

Step-by-Step Procedure

Step 1: Prepare the Workspace

I first created a folder to store all our certificate files:

```
mkdir -p ~/lab5-ca && cd ~/lab5-ca
```

Then copied the OpenSSL config file:

```
cp /etc/ssl/openssl.cnf ./openssl.cnf
```

Step 2: Create a Root Certificate Authority (CA)

A CA is like an ID authority that signs certificates for others.
We created folders for our CA database:

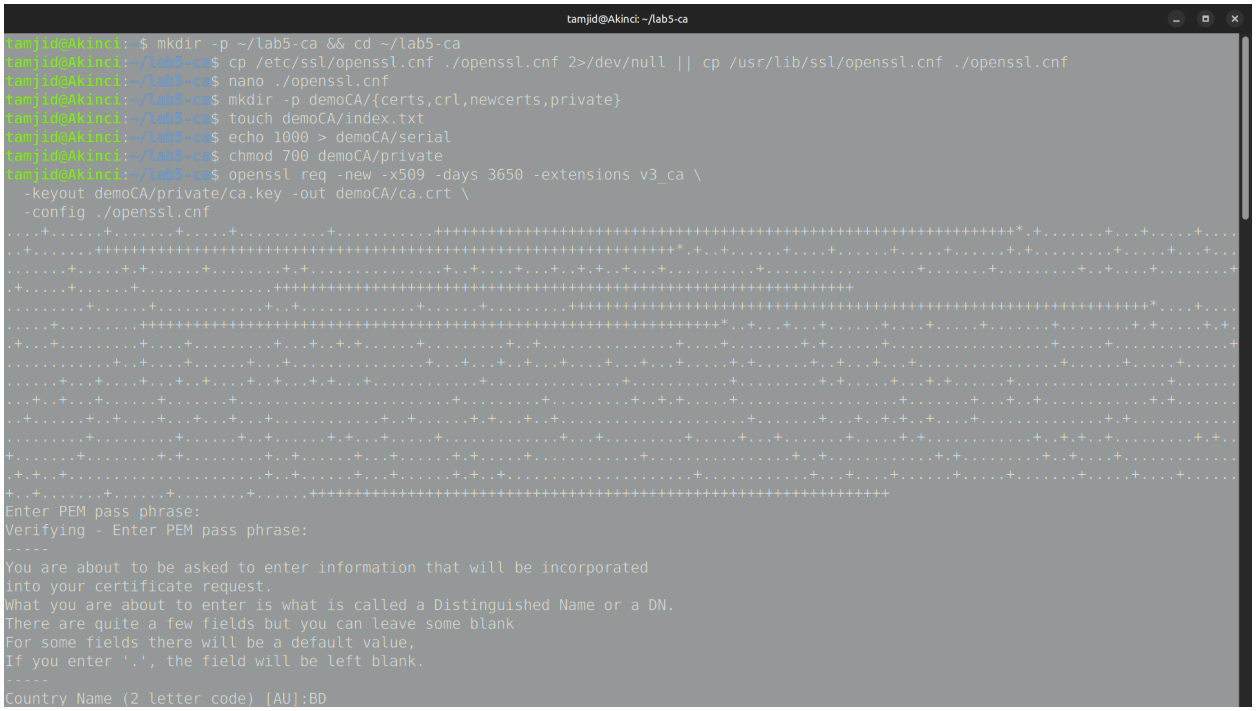
```
mkdir -p demoCA/{certs,crl,newcerts,private}
touch demoCA/index.txt
echo 1000 > demoCA/serial
```

Then I generated a **self-signed CA certificate**:

```
openssl req -new -x509 -days 3650 -extensions v3_ca \
  -keyout demoCA/private/ca.key -out demoCA/ca.crt \
  -config ./openssl.cnf
```

This created two files:

- ca.key → private key (keep it secret)
- ca.crt → your CA certificate



Step 3: Create a Key and Certificate Request for the Server

Next, I generated a private key for our website:

```
openssl genrsa -des3 -out server.key 2048
```

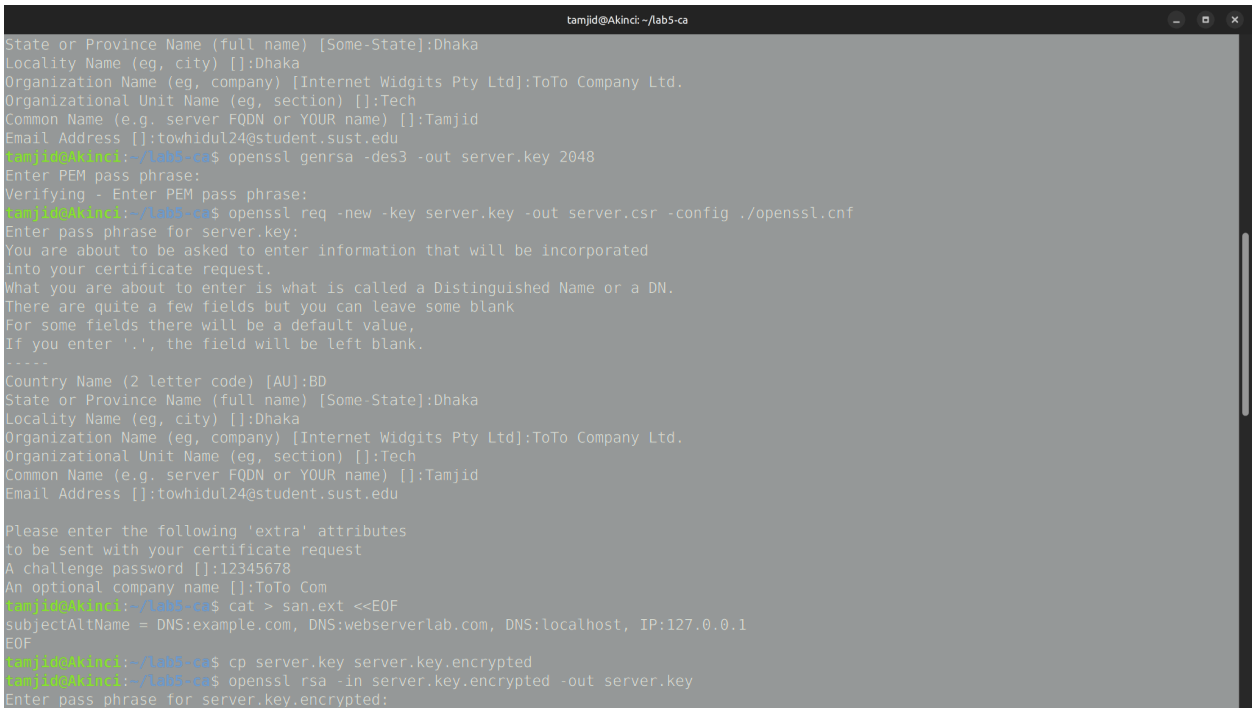
Then created a **CSR (Certificate Signing Request)**:

```
openssl req -new -key server.key -out server.csr -config ./openssl.cnf
```

When asked for “Common Name,” I entered:

example.com

This request will later be signed by our CA.



Step 4: Add SAN (Subject Alternative Names)

To make the certificate valid for multiple domains (like `example.com`, `localhost`), we created a file named `san.ext`:

subjectAltName = DNS:example.com, DNS:webserverlab.com, DNS:localhost, IP:127.0.0.1

This ensures browsers won't complain about mismatched hostnames.

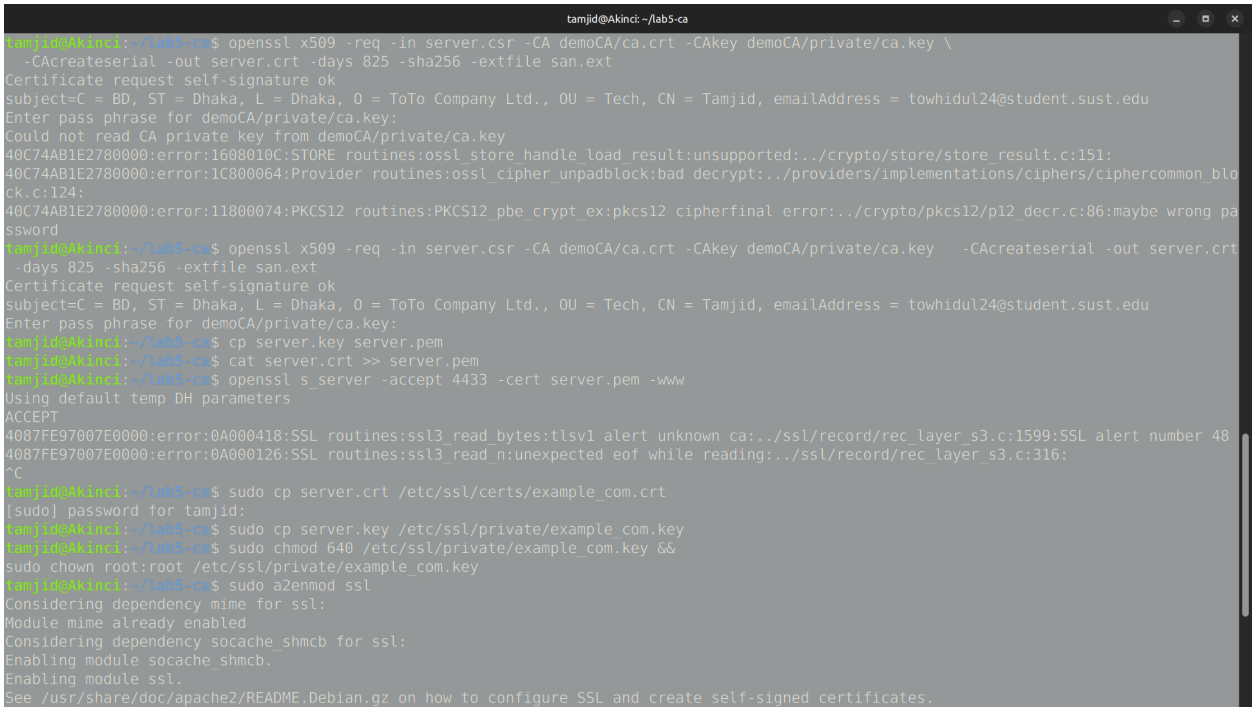
Step 5: Sign the Certificate with Our CA

We used our CA to sign the CSR and generate the actual certificate:

```
openssl x509 -req -in server.csr -CA demoCA/ca.crt -CAkey
demoCA/private/ca.key \
  -CAcreateserial -out server.crt -days 825 -sha256 -extfile san.ext
```

Now we have:

- server.crt → signed certificate
- server.key → server's private key



Step 6: Test HTTPS with OpenSSL

Before using Apache, we tested our certificate with OpenSSL's built-in web server:

```
cp server.key server.pem
cat server.crt >> server.pem
```

```
openssl s_server -accept 4433 -cert server.pem -www
```

Then opened a browser and went to:

https://localhost:4433

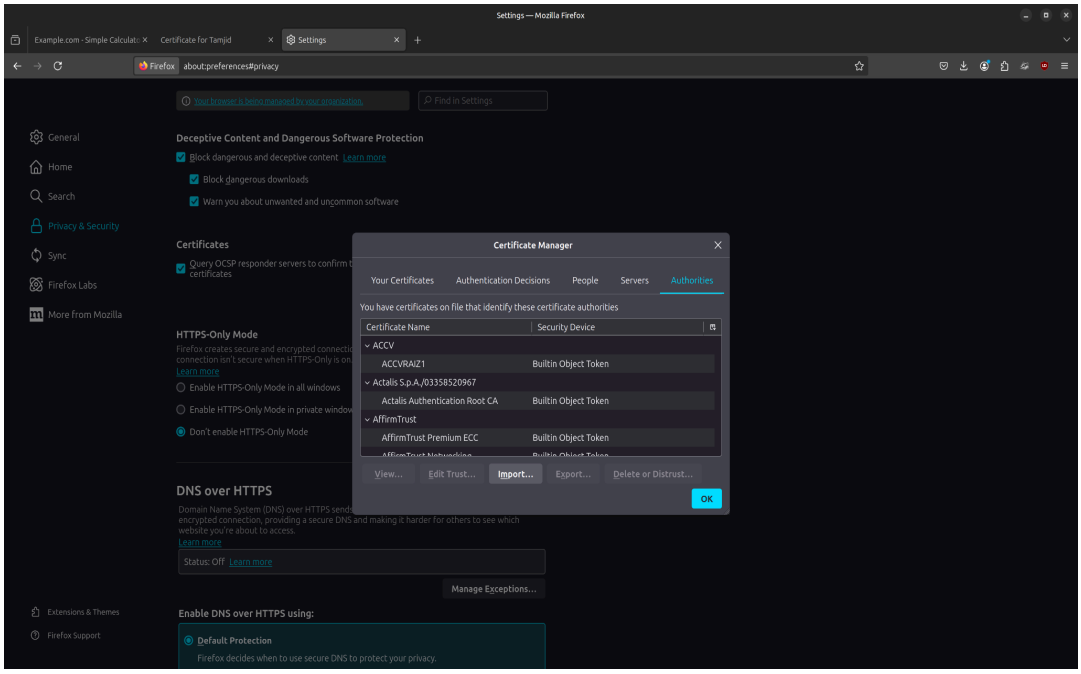
At first, it showed a warning because the browser didn't yet trust our CA.

Step 7: Import Root CA into Browser

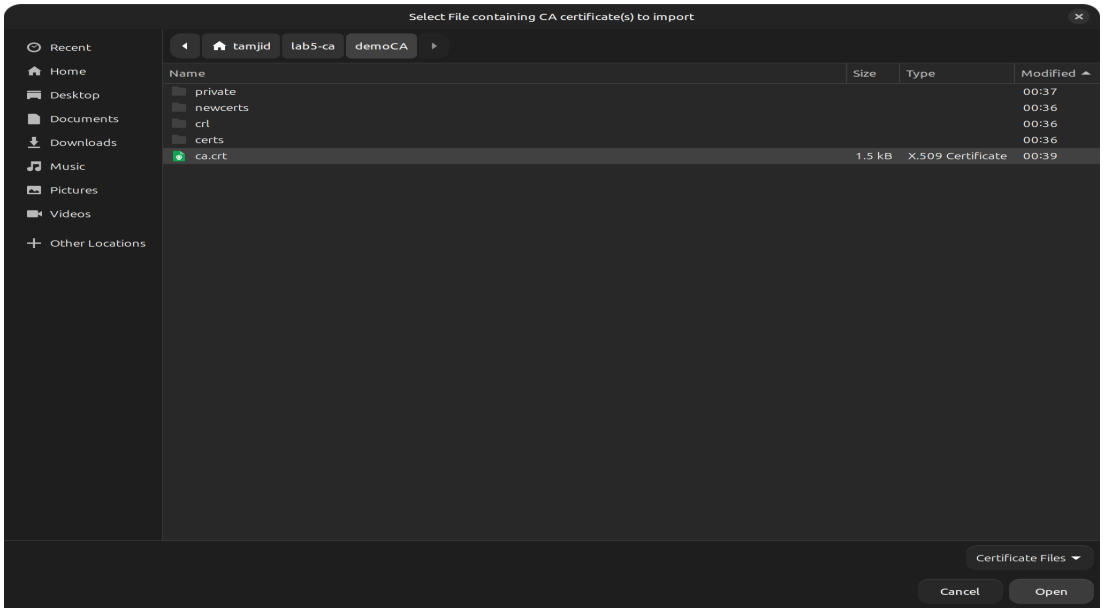
We added our CA to Firefox so it trusts our certificates:

In Firefox:

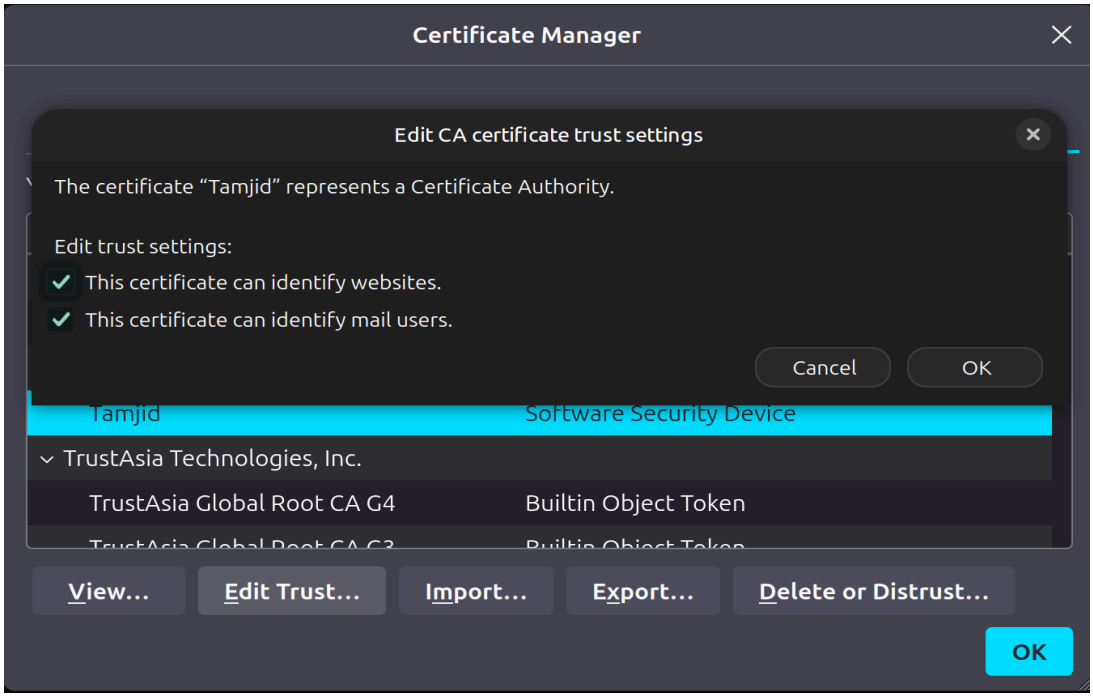
1. Go to **Settings → Privacy & Security → Certificates → View Certificates**



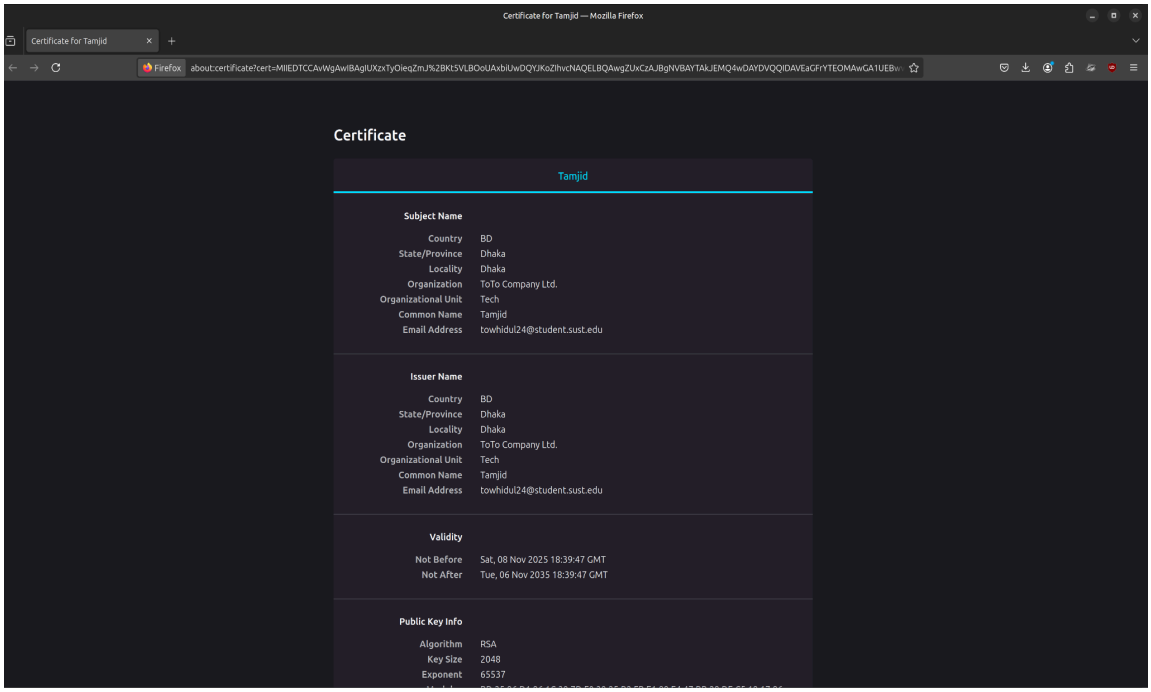
2. Click **Import…** and select demoCA/ca.crt



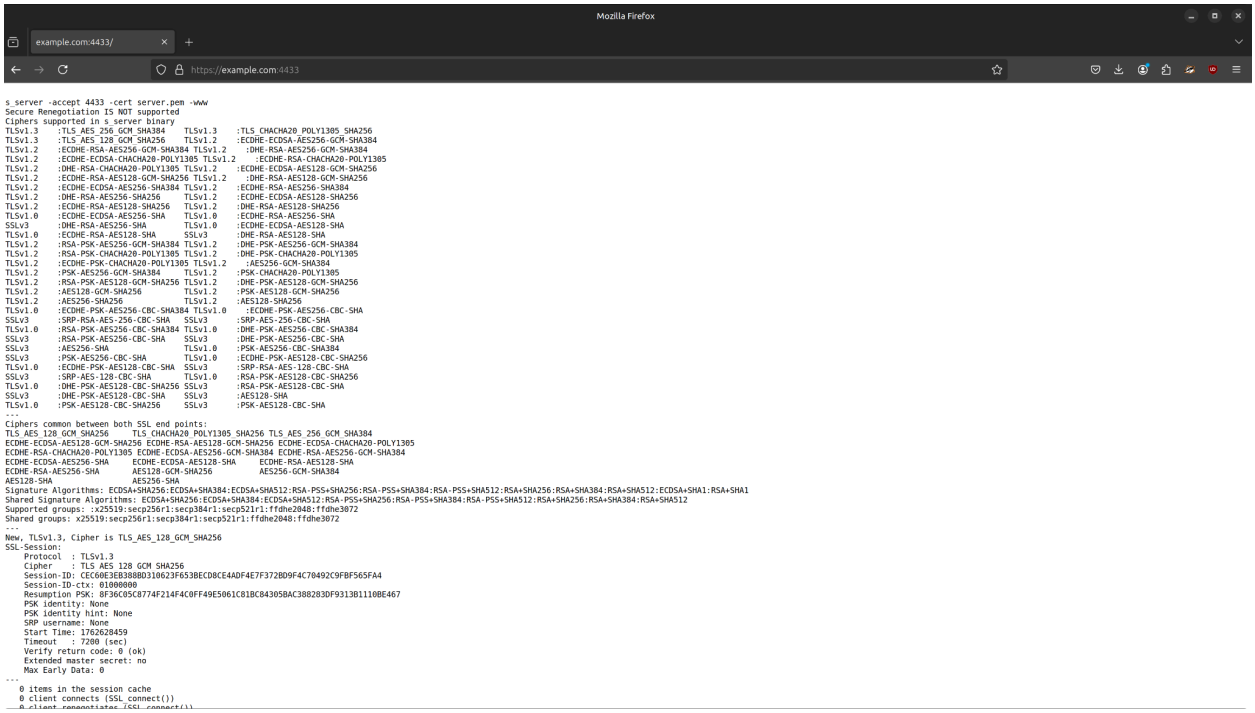
3. Check **Trust this CA to identify websites**



Certificate details in Firefox browser



After that, refreshing the site showed a **padlock icon** — meaning HTTPS is working and trusted.



Step 8: Configure Apache for HTTPS

We copied the certificate and key to system folders:

```
sudo cp server.crt /etc/ssl/certs/example_com.crt
```

```
sudo cp server.key /etc/ssl/private/example_com.key
```

Enabled SSL module:

```
sudo a2enmod ssl
sudo systemctl restart apache2
```

Then edited the Apache virtual host file:

```
sudo nano /etc/apache2/sites-available/example.com.conf
```

Added this block:

```
<IfModule mod_ssl.c>
<VirtualHost *:443>
    ServerAdmin admin@example.com
    ServerName example.com
    DocumentRoot /var/www/example.com/html

    SSLEngine on
    SSLCertificateFile /etc/ssl/certs/example_com.crt
    SSLCertificateKeyFile /etc/ssl/private/example_com.key

    ErrorLog ${APACHE_LOG_DIR}/example_ssl_error.log
    CustomLog ${APACHE_LOG_DIR}/example_ssl_access.log combined
</VirtualHost>
</IfModule>
```

Tested configuration:

```
sudo apache2ctl configtest
sudo systemctl restart apache2
```

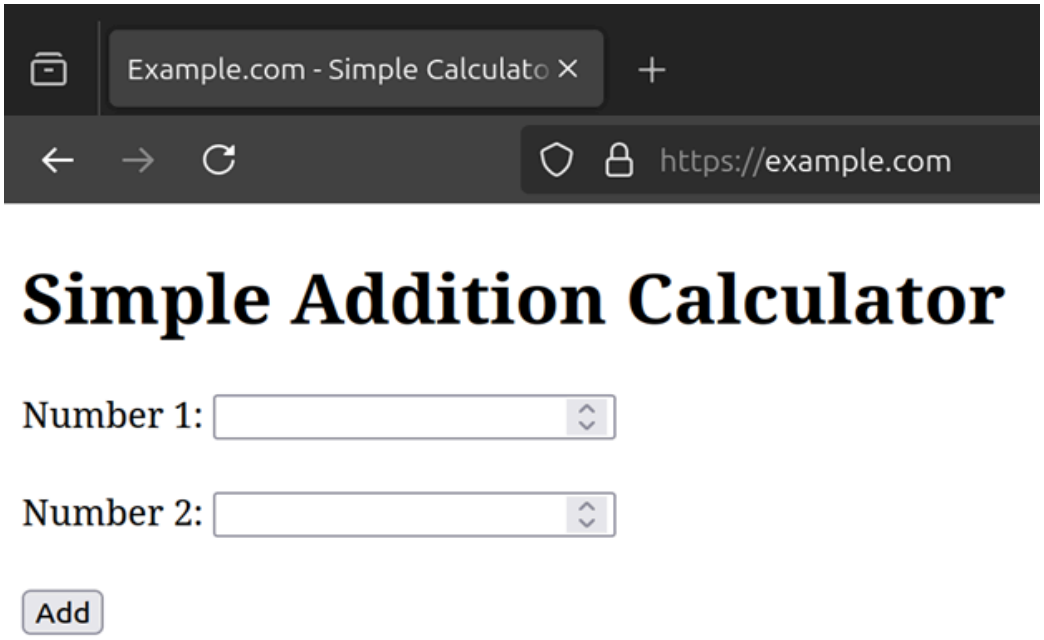
```
tamjid@Akinci:~/lab5-ca$ sudo cp server.crt /etc/ssl/certs/example_com.crt
[sudo] password for tamjid:
tamjid@Akinci:~/lab5-ca$ sudo cp server.key /etc/ssl/private/example_com.key
tamjid@Akinci:~/lab5-ca$ sudo chmod 640 /etc/ssl/private/example_com.key &&
sudo chown root:root /etc/ssl/private/example_com.key
tamjid@Akinci:~/lab5-ca$ sudo a2enmod ssl
Considering dependency mime for ssl:
Module mime already enabled
Considering dependency socache_shmcb for ssl:
Enabling module socache_shmcb.
Enabling module ssl.
See /usr/share/doc/apache2/README.Debian.gz on how to configure SSL and create self-signed certificates.
To activate the new configuration, you need to run:
  systemctl restart apache2
tamjid@Akinci:~/lab5-ca$ sudo systemctl restart apache2
tamjid@Akinci:~/lab5-ca$ nano /etc/apache2/sites-available/example.com.conf
tamjid@Akinci:~/lab5-ca$ sudo nano /etc/apache2/sites-available/example.com.conf
tamjid@Akinci:~/lab5-ca$ sudo a2ensite example.com.conf && sudo apache2ctl configtest
Site example.com already enabled
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 127.0.1.1. Set the 'ServerName' directive g
lobally to suppress this message
Syntax OK
```

Step 9: Verify HTTPS Website

Finally, we opened:

https://example.com

and saw the page load successfully with a **secure padlock** in the address bar.



Results

Task	Description	Status
1	Created self-signed CA	Completed
2	Generated server certificate	Completed
3	Tested HTTPS using OpenSSL	Completed
4	Imported CA into browser	Completed
5	Configured Apache for HTTPS	Completed

Observations

- HTTPS encrypts communication, making it safer than HTTP.
- Self-signed CAs can be used for learning and local development.
- Once the CA is imported, browsers treat our certificates as trusted.
- Apache needs SSL enabled (mod_ssl) to handle HTTPS requests.
- The padlock icon proves encryption and certificate trust.

Web Lab Report — Apache Web Server

Objective

To install and configure Apache Web Server on Ubuntu, create and manage multiple virtual hosts, and understand how hostname resolution works using the /etc/hosts file.

Checkpoint-1: Apache Installation and Default Page

Step 1.1 – Install Apache

```
sudo apt update
```

```
sudo apt install apache2 -y
```

Step 1.2 – Allow Apache Through Firewall

```
sudo ufw allow 'Apache'
```

```
sudo ufw status
```

Step 1.3 – Verify Apache Service

```
sudo systemctl status apache2
```

Confirm it shows **active (running)**.

Step 1.4 – Add Local Domain

Edit /etc/hosts and add:

```
127.0.0.1   localhost webserverlab.com
```

Step 1.5 – Test in Browser

Visit:

http://webserverlab.com

You should see the **Apache2 Ubuntu Default Page**.

Checkpoint-2: Create and Configure First Virtual Host (example.com)

Step 2.1 – Create Directory Structure and Provide Privilege for Ownership and Execution

```
sudo mkdir -p /var/www/example.com/html
```

```
sudo chown -R $USER:$USER /var/www/example.com/html
```

```
sudo chmod -R 755 /var/www/example.com
```

Step 2.2 – Create Website Content

```
nano /var/www/example.com/html/index.html
```

```
<html>
  <head><title>Welcome to Example.com!</title></head>
  <body><h1>Success! example.com virtual host is working!</h1></body>
</html>
```

Step 2.3 – Create Virtual Host Config

```
sudo nano /etc/apache2/sites-available/example.com.conf
```

```
<VirtualHost *:80>
    ServerAdmin admin@example.com
    ServerName example.com
    ServerAlias www.example.com
    DocumentRoot /var/www/example.com/html

    ErrorLog ${APACHE_LOG_DIR}/example_error.log
    CustomLog ${APACHE_LOG_DIR}/example_access.log combined
</VirtualHost>
```

Step 2.4 – Enable Site and Disable Default

```
sudo a2ensite example.com.conf
```

```
sudo a2dissite 000-default.conf
```

```
sudo systemctl restart apache2
```

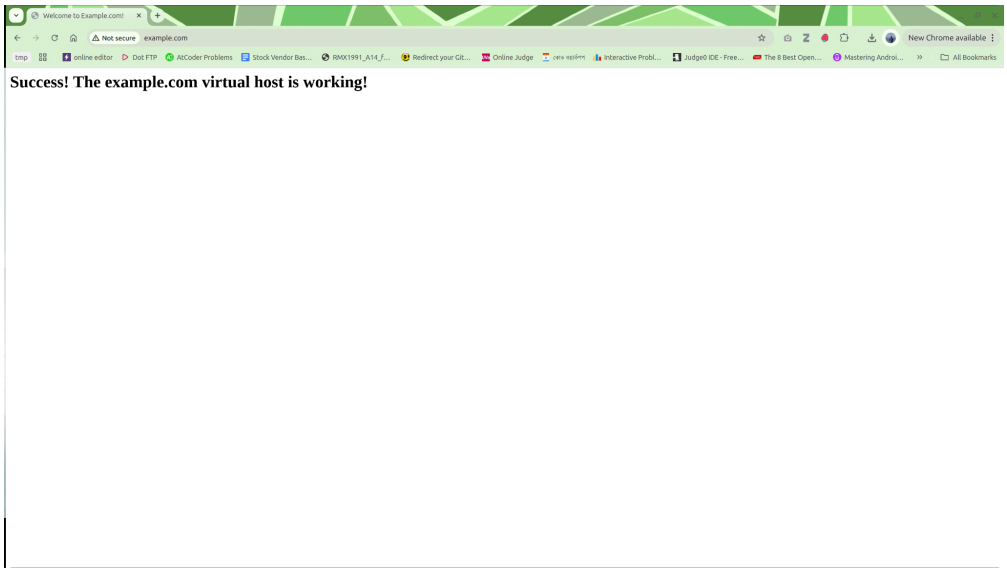
Step 2.5 – Update /etc/hosts

```
127.0.0.1   localhost webserverlab.com example.com
```

Step 2.6 – Test Site

Visit:

http://example.com



Checkpoint-3: Observing Default Host Behavior

Now visit:

http://webserverlab.com
http://anothervhost.com

Both URLs will load the same content as example.com.

Reason:
When 000-default.conf is disabled, Apache uses the **first active virtual host** as the default for any unmatched request.

Checkpoint-4: Creating a Second Virtual Host (anothervhost.com)

Step 4.1 – Create Directory Structure and Provide Privilege for Ownership and Execution

```
sudo mkdir -p /var/www/anothervhost.com/html
```

```
sudo chown -R $USER:$USER /var/www/anothervhost.com/html
```

```
sudo chmod -R 755 /var/www/anothervhost.com
```

Step 4.2 – Create HTML Content

```
nano /var/www/anothervhost.com/html/index.html
```

```
<html>
  <head><title>Welcome to AnotherVHost!</title></head>
  <body><h1>This is another virtual host site.</h1></body>
</html>
```

Step 4.3 – Create Configuration File

```
sudo nano /etc/apache2/sites-available/anothervhost.com.conf
```

```
<VirtualHost *:80>
    ServerAdmin admin@anothervhost.com
    ServerName anothervhost.com
    ServerAlias www.anothervhost.com
    DocumentRoot /var/www/anothervhost.com/html

    ErrorLog ${APACHE_LOG_DIR}/another_error.log
    CustomLog ${APACHE_LOG_DIR}/another_access.log combined
</VirtualHost>
```

Step 4.4 – Enable Site and Restart Apache

```
sudo a2ensite anothervhost.com.conf
```

```
sudo apache2ctl configtest
```

```
sudo systemctl restart apache2
```

Step 4.5 – Update Hosts File

```
127.0.0.1   localhost webserverlab.com example.com anothervhost.com
```

Step 4.6 – Test Both Sites

Visit:

http://example.com
http://anothervhost.com

Each should load its own distinct page.

Checkpoint-5: Hosting Two Dynamic JavaScript-Based Websites

Step 5.1 – Verify Existing Virtual Hosts

Ensure both virtual hosts are already configured and working:

```
sudo a2ensite example.com.conf
```

```
sudo a2ensite anothervhost.com.conf
```

```
sudo systemctl restart apache2
```

Verify in /etc/hosts:

```
127.0.0.1   localhost webserverlab.com example.com anothervhost.com
```

Step 5.2 – Dynamic Website 1: example.com (Simple Calculator)

a. Create/Replace HTML file

```
nano /var/www/example.com/html/index.html
```

Paste the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Example.com - Simple Calculator</title>
  </head>
  <body>
    <h1>Simple Addition Calculator</h1>
    <form id="calcForm">
      <label>Number 1:
        <input type="number" id="num1" required />
      </label><br /><br />
      <label>Number 2:
```

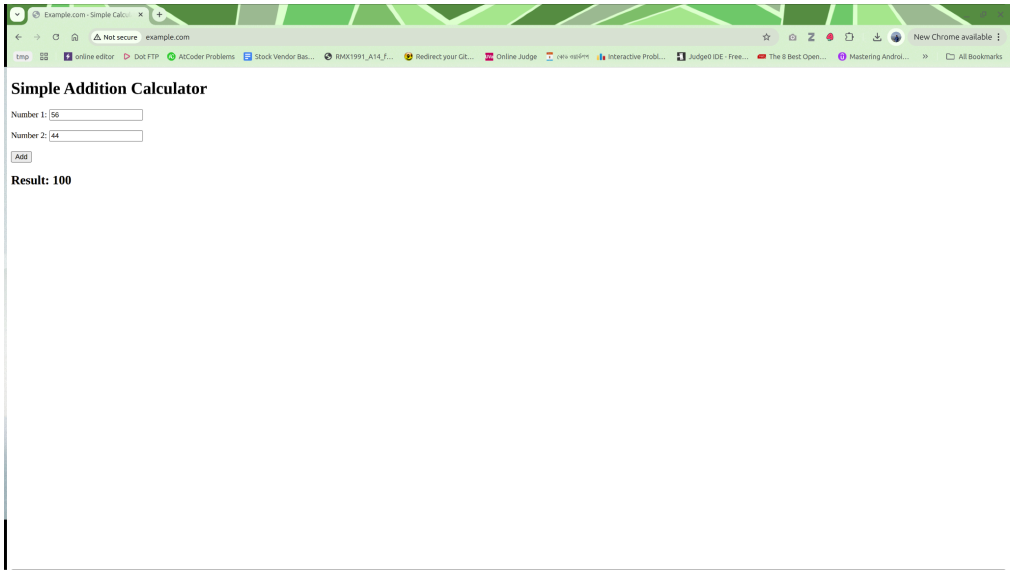
```
      <input type="number" id="num2" required />
    </label><br /><br />
    <button type="submit">Add</button>
  </form>
  <h2 id="result"></h2>

  <script>
    const form = document.getElementById("calcForm");
    const result = document.getElementById("result");
    form.addEventListener("submit", function (e) {
      e.preventDefault();
      const a = parseFloat(document.getElementById("num1").value);
      const b = parseFloat(document.getElementById("num2").value);
      if (isNaN(a) || isNaN(b)) {
        result.textContent = "Please enter valid numbers.";
      } else {
        result.textContent = "Result: " + (a + b);
      }
    });
  </script>
 </body>
</html>
```

b. Test in Browser

Visit:

http://example.com

Enter two numbers and press **Add**.
The result should appear dynamically.



Step 5.3 – Dynamic Website 2: anothervhost.com (Grade Calculator)

a. Create/Replace HTML file

```
nano /var/www/anothervhost.com/html/index.html
```

Paste:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Anothervhost.com - Grade Calculator</title>
  </head>
  <body>
    <h1>Grade Percentage Calculator</h1>
    <form id="gradeForm">
      <label>Quiz (out of 20): <input type="number" id="quiz" required
/></label><br /><br />
      <label>Midterm (out of 30): <input type="number" id="midterm"
required /></label><br /><br />
      <label>Final (out of 50): <input type="number" id="final" required
/></label><br /><br />
```

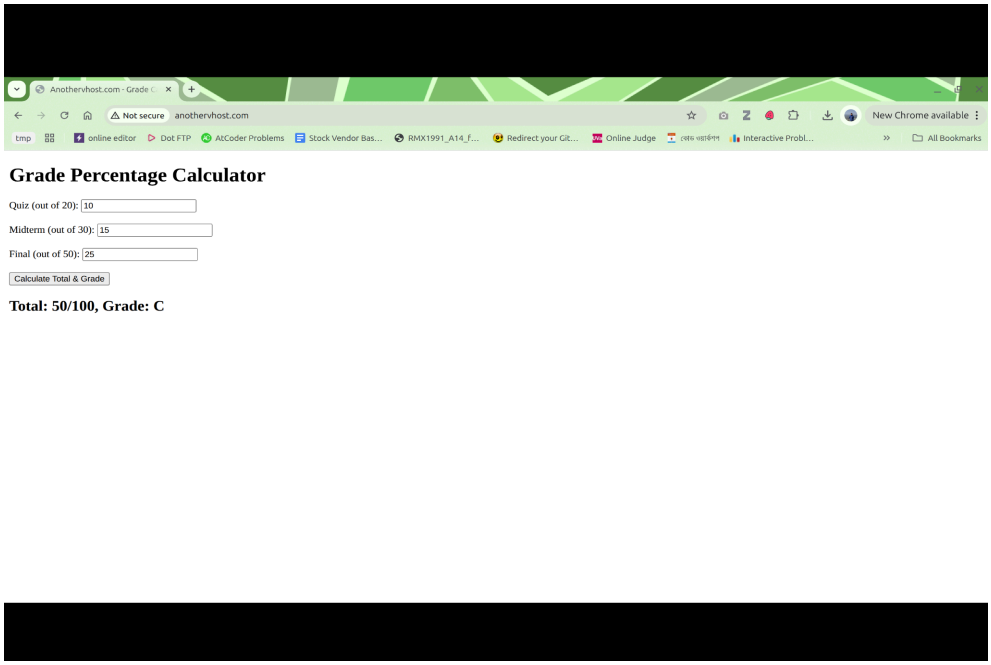
```
    <button type="submit">Calculate Total & Grade</button>
  </form>
  <h2 id="gradeResult"></h2>

  <script>
    const form = document.getElementById("gradeForm");
    const output = document.getElementById("gradeResult");
    form.addEventListener("submit", function (e) {
      e.preventDefault();
      const quiz = parseFloat(document.getElementById("quiz").value);
      const mid = parseFloat(document.getElementById("midterm").value);
      const fin = parseFloat(document.getElementById("final").value);
      if (isNaN(quiz) || isNaN(mid) || isNaN(fin)) {
        output.textContent = "Please enter all scores.";
        return;
      }
      const total = quiz + mid + fin;
      let grade;
      if (total >= 80) grade = "A+";
      else if (total >= 70) grade = "A";
      else if (total >= 60) grade = "B";
      else if (total >= 50) grade = "C";
      else grade = "F";
      output.textContent = "Total: " + total + "/100, Grade: " + grade;
    });
  </script>
 </body>
</html>
```

b. Test in Browser

Visit:

http://anothervhost.com

Input quiz, midterm, and final marks → click **Calculate** → total and grade appear instantly.



Step 5.4 – Validation

Both sites work independently and dynamically:

Virtual Host	Functionality	Dynamic Feature
example.com	Addition Calculator	JavaScript form + instant result
anothervhost.com	Grade Calculator	JavaScript form + instant grade output

Observations

- Apache successfully serves multiple **dynamic** front-end websites.
- No server-side language was used; all logic handled via **client-side JavaScript**.

- Each virtual host uses its own document root and configuration file.

- Demonstrates the concept of **hosting multiple applications/domains** on a single local machine.