Practical-1

Aim: To implement Caesar cipher encryption-decryption.

Solution:

Code:

```
#include <stdio.h>
// Function to encrypt the message
void encrypt(char message[], int key) {
    int i = 0;
    char ch;
    while (message[i] != '\0') {
        ch = message[i];
        // Encrypt uppercase letters
        if (ch >= 'A' && ch <= 'Z') {
            ch = ((ch - 'A') + key) % 26 + 'A';
        }
        // Encrypt lowercase letters
        else if (ch >= 'a' && ch <= 'z') {
            ch = ((ch - 'a') + key) % 26 + 'a';
        }
        message[i] = ch;
        i++;
    }
}
// Function to decrypt the message
void decrypt(char message[], int key) {
    // To decrypt a message, we use the negative value of the key
    key = -key;
    encrypt(message, key);
}
```

```
int main() {
    char message[100];
    int key;
    printf("Enter a message: ");
    fgets(message, sizeof(message), stdin);
    printf("Enter the key (0-25): ");
    scanf("%d", &key);
    encrypt(message, key);
    printf("Encrypted message: %s\n", message);
    decrypt(message, key);
    printf("Decrypted message: %s\n", message);
    return 0;
}
```

Output:

```
Enter a message: John
Enter the key (0-25): 2
Encrypted message: Lqjp

Decrypted message: John


Process returned 0 (0x0)   execution time : 18.395 s
Press any key to continue.
```

Practical-2

Aim: To implement Monoalphabetic cipher encryption-decryption.

Solution:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#define MAX_LENGTH 100
int main() {
    int randomNumbers[26];  // Array to store the random numbers
    int count = 0;          // Counter for unique numbers
    srand(time(0));         // Seed the random number generator with the current time
    while (count < 26) {
        int randomNumber = rand() % 26 + 65;  // Generate a random number between 65 and 90
(inclusive)
        // Check if the number is already generated
        int isDuplicate = 0;
        for (int i = 0; i < count; i++) {
            if (randomNumbers[i] == randomNumber) {
                isDuplicate = 1;
                break;
            }
        }
        if (!isDuplicate) {
            randomNumbers[count] = randomNumber;  // Store the unique number
            count++;
        }
    }
```

```c
char plainText[MAX_LENGTH];
printf("Enter the plain text: ");
fgets(plainText, sizeof(plainText), stdin);
plainText[strcspn(plainText, "\n")] = '\0';  // Remove trailing newline character
int textLength = strlen(plainText);
char cipherText[MAX_LENGTH];
char decryptedText[MAX_LENGTH];
// Generate the cipher text
for (int i = 0; i < textLength; i++) {
    char currentChar = plainText[i];
    if (currentChar >= 'A' && currentChar <= 'Z') {
        int key = randomNumbers[currentChar - 'A'];
        cipherText[i] = key;
    } else {
        cipherText[i] = currentChar;
    }
}
cipherText[textLength] = '\0';
// Decrypt the cipher text
for (int i = 0; i < textLength; i++) {
    char currentChar = cipherText[i];
    if (currentChar >= 'A' && currentChar <= 'Z') {
        int index = 0;
        while (randomNumbers[index] != currentChar) {
            index++;
        }
        decryptedText[i] = 'A' + index;
    } else {
        decryptedText[i] = currentChar;
    }
}
```

```
    decryptedText[textLength] = '\0';
    // Print the generated keys
    printf("Generated Keys: \n");
    for (int i = 0; i < 26; i++) {
        printf("%d: %c\n", i+1, (char)randomNumbers[i]);
    }
    printf("\n");
    // Print the cipher text
    printf("Cipher Text: %s\n", cipherText);
    // Print the decrypted text
    printf("Decrypted Text: %s\n", decryptedText);
    return 0;
}
```

Output:

```
Enter the plain text: JOHN
Generated Keys:
1: F
2: Z
3: Y
4: X
5: D
6: G
7: Q
8: C
9: J
10: P
11: R
12: S
13: E
14: L
15: H
16: T
17: A
18: K
19: N
20: W
21: B
22: I
23: O
24: V
25: U
26: M

Cipher Text: PHCL
Decrypted Text: JOHN

Process returned 0 (0x0)   execution time : 25.793 s
Press any key to continue.
```

Practical-3

Aim: To implement Playfair cipher encryption-decryption.

Solution:

Code:

```
#include <stdio.h>
int check(char table[5][5], char k) {
    int i, j;
    for (i = 0; i < 5; ++i) {
        for (j = 0; j < 5; ++j) {
            if (table[i][j] == k)
                return 0;
        }
    }
    return 1;
}
int main() {
    int i, j, key_len;
    char table[5][5];
    for (i = 0; i < 5; ++i) {
        for (j = 0; j < 5; ++j) {
            table[i][j] = '0';
        }
    }
    printf("**********Playfair Cipher************\n\n");
    printf("Enter the length of the Key: ");
    scanf("%d", &key_len);
    fflush(stdin);

    char key[key_len];
```

```
printf("Enter the Key: ");
for (i = 0; i < key_len; ++i) {
    scanf(" %c", &key[i]);
    fflush(stdin);
    if (key[i] == 'j' || key[i] == 'J')
        key[i] = 'I';
}
int flag;
int count = 0;
// inserting the key into the table
for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
        flag = 0;
        while (flag != 1) {
            if (count >= key_len)
                goto l1;
            flag = check(table, key[count]);
            ++count;
        }// end of while
        table[i][j] = key[(count - 1)];
    }// end of inner for
}// end of outer for
l1: printf("\n");
int val = 65;
//inserting other alphabets
for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
        if (table[i][j] >= 65 && table[i][j] <= 90) {
        } else {
            flag = 0;
            while (flag != 1) {
```

```
                if ('J' == (char) val)
                    ++val;
                flag = check(table, (char) val);
                ++val;
            }// end of while
            table[i][j] = (char) (val - 1);
        }//end of else
    }// end of inner for
}// end of outer for
printf("The table is as follows:\n");
for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
        printf("%c ", table[i][j]);
    }
    printf("\n");
}
int l = 0;
printf("\nEnter the length of the plain text (without spaces): ");
scanf("%d", &l);
fflush(stdin);
printf("\nEnter the Plain text: ");
char p[l];
for (i = 0; i < l; ++i) {
    scanf(" %c", &p[i]);
    fflush(stdin);
    if (p[i] == 'j' || p[i] == 'J')
        p[i] = 'I';
}
printf("\nThe replaced text (J with I): ");
for (i = 0; i < l; ++i)
    printf("%c ", p[i]);
```

```
count = 0;
for (i = 0; i < l; ++i) {
    if (p[i] == p[i + 1])
        count = count + 1;
}
printf("\nThe cipher has to enter %d bogus char. It is either 'X' or 'Z'\n",
    count);
int length = 0;
if ((l + count) % 2 != 0)
    length = (l + count + 1);
else
    length = (l + count);
printf("\nValue of length is %d.\n", length);
char p1[length];
//inserting bogus characters.
char temp1;
int count1 = 0;
for (i = 0; i < l; ++i) {
    p1[count1] = p[i];
    if (p[i] == p[i + 1]) {
        count1 = count1 + 1;
        if (p[i] == 'X' || p[i] == 'x')
            p1[count1] = 'Z';
        else
            p1[count1] = 'X';
    }
    count1 = count1 + 1;
}
//checking for length
char bogus;
```

```
if ((l + count) % 2 != 0) {
    if (p1[length - 1] == 'X' || p1[length - 1] == 'x')
        p1[length] = 'Z';
    else
        p1[length] = 'X';
}
printf("The final text is: ");
for (i = 0; i < length; ++i)
    printf("%c ", p1[i]);
char cipher_text[length];
int r1, r2, c1, c2;
int k1 = 0; // Initialize k1 here
for (k1 = 0; k1 < length; ++k1) { // Corrected k1 initialization and termination condition
    for (i = 0; i < 5; ++i) {
        for (j = 0; j < 5; ++j) {
            if (table[i][j] == p1[k1]) {
                r1 = i;
                c1 = j;
            } else if (table[i][j] == p1[k1 + 1]) {
                r2 = i;
                c2 = j;
            }
        }//end of for with j
    }//end of for with i
    if (r1 == r2) {
        cipher_text[k1] = table[r1][(c1 + 1) % 5];
        cipher_text[k1 + 1] = table[r1][(c2 + 1) % 5];
    } else if (c1 == c2) {
        cipher_text[k1] = table[(r1 + 1) % 5][c1];
        cipher_text[k1 + 1] = table[(r2 + 1) % 5][c1];
    } else {
```

```
            cipher_text[k1] = table[r1][c2];
            cipher_text[k1 + 1] = table[r2][c1];
        }
        // Increment k1 by 2, not 1
        k1 = k1 + 1;
    }//end of for with k1
    printf("\n\nThe Cipher text is: ");
    for (i = 0; i < length; ++i)
        printf("%c ", cipher_text[i]);
    return 0;
}
```

Output:

```
**********Playfair Cipher************

Enter the length of the Key: 12
Enter the Key: V
A
N
D
E
M
A
T
A
R
A
M

The table is as follows:
V A N D E
M T R B C
F G H I K
L O P Q S
U W X Y Z

Enter the length of the plain text (without spaces): 10

Enter the Plain text: I
L
O
V
E
I
N
D
I
A

The replaced text (J with I): I L O V E I N D I A
The cipher has to enter 0 bogus char. It is either 'X' or 'Z'

Value of length is 10.
The final text is: I L O V E I N D I A

The Cipher text is: F Q L A D K D E G D
Process returned 0 (0x0)   execution time : 39.851 s
Press any key to continue.
```

Practical-4

Aim: To implement Polyalphabetic cipher encryption-decryption.

Solution:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
// Function to generate and print the matrix of alphabets used in the cipher
void print_alphabet_matrix() {
    printf("Alphabet Matrix:\n");
    for (int i = 0; i < 26; i++) {
        printf("%c | ", 'A' + i);
        for (int j = 0; j < 26; j++) {
            printf("%c ", (i + j) % 26 + 'A');
        }
        printf("\n");
    }
    printf("\n");
}
// Function to perform Polyalphabetic encryption and return the keyword shift values
char* polyalphabetic_encrypt(const char* plaintext, const char* keyword, int** shift_values) {
    int plain_len = strlen(plaintext);
    int key_len = strlen(keyword);
    char* ciphertext = (char*)malloc(plain_len + 1);
    *shift_values = (int*)malloc(plain_len * sizeof(int));
    for (int i = 0; i < plain_len; i++) {
        if (!isalpha(plaintext[i])) {
            ciphertext[i] = plaintext[i];
            (*shift_values)[i] = 0;
```

```c
        } else {
            char base = isupper(plaintext[i]) ? 'A' : 'a';
            int shift = toupper(keyword[i % key_len]) - 'A';
            ciphertext[i] = (plaintext[i] - base + shift) % 26 + base;
            (*shift_values)[i] = shift;
        }
    }
    ciphertext[plain_len] = '\0';
    return ciphertext;
}
// Function to perform Polyalphabetic decryption and return the keyword shift values
char* polyalphabetic_decrypt(const char* ciphertext, const char* keyword, int** shift_values) {
    int cipher_len = strlen(ciphertext);
    int key_len = strlen(keyword);
    char* plaintext = (char*)malloc(cipher_len + 1);
    *shift_values = (int*)malloc(cipher_len * sizeof(int));

    for (int i = 0; i < cipher_len; i++) {
        if (!isalpha(ciphertext[i])) {
            plaintext[i] = ciphertext[i];
            (*shift_values)[i] = 0;
        } else {
            char base = isupper(ciphertext[i]) ? 'A' : 'a';
            int shift = toupper(keyword[i % key_len]) - 'A';
            plaintext[i] = (ciphertext[i] - base - shift + 26) % 26 + base;
            (*shift_values)[i] = shift;
        }
    }
    plaintext[cipher_len] = '\0';
    return plaintext;
}
```

```c
int main() {
    print_alphabet_matrix();
    char plaintext[1000];
    char keyword[100];
    printf("Enter the keyword: ");
    scanf("%s", keyword);
    printf("Enter the plaintext: ");
    getchar(); // Clear the newline character from the buffer
    fgets(plaintext, sizeof(plaintext), stdin);
    int* encrypt_shift_values;
    int* decrypt_shift_values;
    char* encrypted = polyalphabetic_encrypt(plaintext, keyword, &encrypt_shift_values);
    char* decrypted = polyalphabetic_decrypt(encrypted, keyword, &decrypt_shift_values);
    printf("Encrypted text: %s\n", encrypted);
    printf("Decrypted text: %s\n", decrypted);
    free(encrypted);
    free(decrypted);
    free(encrypt_shift_values);
    free(decrypt_shift_values);
    return 0;
}
```

Output:

```
Alphabet Matrix:
A | A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
B | B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
C | C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D | D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
E | E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F | F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G | G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H | H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I | I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J | J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K | K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L | L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M | M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N | N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O | O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
P | P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q | Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R | R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S | S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T | T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U | U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V | V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W | W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X | X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y | Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z | Z A B C D E F G H I J K L M N O P Q R S T U V W X Y

Enter the keyword: DECEPTIVE
Enter the plaintext: WEAREDISCOVEREDSAVEYOURSELF
Encrypted text: ZICVTWQNGRZGVTWAVZHCQYGLMGJ

Decrypted text: WEAREDISCOVEREDSAVEYOURSELF


Process returned 0 (0x0)   execution time : 16.736 s
Press any key to continue.
```

Practical-5

Aim: To implement Hill cipher encryption-decryption.

Solution:

Code:

```
#include <stdio.h>
#include <stdlib.h>
// Function to calculate the modulo multiplicative inverse of a number
int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) {
            return x;
        }
    }
    return 0;
}
// Function to encrypt the message using Hill cipher
void hillEncrypt(int **key, int *message, int *encrypted, int matrixSize, int messageSize) {
    for (int i = 0; i < messageSize; i += matrixSize) {
        for (int j = 0; j < matrixSize; j++) {
            encrypted[i + j] = 0;
            for (int k = 0; k < matrixSize; k++) {
                encrypted[i + j] += key[j][k] * message[i + k];
            }
            encrypted[i + j] %= 26; // Modulo 26 to handle alphabetic characters (assuming a-z = 0-25)
        }
    }
}
// Function to decrypt the message using Hill cipher
void hillDecrypt(int **key, int *encrypted, int *decrypted, int matrixSize, int messageSize) {
```

```
int determinant = 0;
// Calculate the determinant of the key matrix
if (matrixSize == 2) {
    determinant = (key[0][0] * key[1][1] - key[0][1] * key[1][0]);
} else if (matrixSize == 3) {
    determinant = (key[0][0] * (key[1][1] * key[2][2] - key[1][2] * key[2][1]) -
                   key[0][1] * (key[1][0] * key[2][2] - key[1][2] * key[2][0]) +
                   key[0][2] * (key[1][0] * key[2][1] - key[1][1] * key[2][0]));
} else {
    printf("Matrix size %d is not supported for decryption.\n", matrixSize);
    return;
}
// Ensure the determinant is non-zero (i.e., matrix is invertible)
while (determinant < 0) {
    determinant += 26;
}
int inv_det = modInverse(determinant, 26);
// Calculate the adjugate of the key matrix
int **adj = (int **)malloc(matrixSize * sizeof(int *));
for (int i = 0; i < matrixSize; i++) {
    adj[i] = (int *)malloc(matrixSize * sizeof(int));
}
if (matrixSize == 2) {
    adj[0][0] = key[1][1];
    adj[0][1] = -key[0][1];
    adj[1][0] = -key[1][0];
    adj[1][1] = key[0][0];
} else if (matrixSize == 3) {
    adj[0][0] = key[1][1] * key[2][2] - key[1][2] * key[2][1];
    adj[0][1] = key[0][2] * key[2][1] - key[0][1] * key[2][2];
    adj[0][2] = key[0][1] * key[1][2] - key[0][2] * key[1][1];
```

```
    adj[1][0] = key[1][2] * key[2][0] - key[1][0] * key[2][2];
    adj[1][1] = key[0][0] * key[2][2] - key[0][2] * key[2][0];
    adj[1][2] = key[0][2] * key[1][0] - key[0][0] * key[1][2];
    adj[2][0] = key[1][0] * key[2][1] - key[1][1] * key[2][0];
    adj[2][1] = key[0][1] * key[2][0] - key[0][0] * key[2][1];
    adj[2][2] = key[0][0] * key[1][1] - key[0][1] * key[1][0];
}
// Calculate the inverse of the key matrix
for (int i = 0; i < matrixSize; i++) {
    for (int j = 0; j < matrixSize; j++) {
        while (adj[i][j] < 0) {
            adj[i][j] += 26;
        }
        adj[i][j] = adj[i][j] * inv_det % 26;
    }
}
// Decrypt the message using the inverse of the key matrix
for (int i = 0; i < messageSize; i += matrixSize) {
    for (int j = 0; j < matrixSize; j++) {
        decrypted[i + j] = 0;
        for (int k = 0; k < matrixSize; k++) {
            decrypted[i + j] += adj[j][k] * encrypted[i + k];
        }
        decrypted[i + j] %= 26; // Modulo 26 to handle alphabetic characters (assuming a-z = 0-25)
    }
}
// Free the dynamically allocated memory for the adjugate matrix
for (int i = 0; i < matrixSize; i++) {
    free(adj[i]);
}
free(adj);
```

```
}

int main() {
    int matrixSize;
    // Get the matrix size from the user
    printf("Enter the size of the key matrix (e.g., 2, 3, 4, etc.): ");
    scanf("%d", &matrixSize);
    // Allocate memory for the key matrix dynamically
    int **key = (int **)malloc(matrixSize * sizeof(int *));
    for (int i = 0; i < matrixSize; i++) {
        key[i] = (int *)malloc(matrixSize * sizeof(int));
    }
    // Get the key matrix elements from the user
    printf("Enter the key matrix elements (row by row):\n");
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            scanf("%d", &key[i][j]);
        }
    }
    // Input message (replace this with your actual message)
    int messageSize;
    printf("Enter the size of the message (must be a multiple of %d): ", matrixSize);
    scanf("%d", &messageSize);
    int *message = (int *)malloc(messageSize * sizeof(int));
    int *encrypted = (int *)malloc(messageSize * sizeof(int));
    int *decrypted = (int *)malloc(messageSize * sizeof(int));
    printf("Enter the message elements (numeric representation, e.g., 0-25):\n");
    for (int i = 0; i < messageSize; i++) {
        scanf("%d", &message[i]);
    }
```

```
    // Encrypt the message
    hillEncrypt(key, message, encrypted, matrixSize, messageSize);
    // Print the encrypted message
    printf("Encrypted message: ");
    for (int i = 0; i < messageSize; i++) {
        printf("%c", 'A' + encrypted[i]); // Convert numeric representation back to alphabetic characters
    }
    printf("\n");
    // Decrypt the message
    hillDecrypt(key, encrypted, decrypted, matrixSize, messageSize);
    // Print the decrypted message
    printf("Decrypted message: ");
    for (int i = 0; i < messageSize; i++) {
        printf("%c", 'A' + decrypted[i]); // Convert numeric representation back to alphabetic characters
    }
    printf("\n");
    // Free dynamically allocated memory
    for (int i = 0; i < matrixSize; i++) {
        free(key[i]);
    }
    free(key);
    free(message);
    free(encrypted);
    free(decrypted);
    return 0;
}
```

Output:

```
Enter the key matrix elements (row by row):
17
17
5
21
18
21
2
2
19
Enter the size of the message (must be a multiple of 3): 12
Enter the message elements (numeric representation, e.g., 0-25):
15
0
24
12
14
17
4
12
14
13
4
24
Encrypted message: LNSHDLEWMTRW
Decrypted message: PAYMOREMONEY

Process returned 0 (0x0)   execution time : 43.588 s
Press any key to continue.
```

Practical-6

Aim: To implement Rail Fence and Columnar transposition cipher encryption-decryption.

Solution:

- **Rail Fence Cipher**

Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
// Function to perform Rail Fence cipher encryption
void railFenceEncrypt(char plaintext[], int key) {
    int len = strlen(plaintext);
    // Create the matrix for the rail fence pattern
    char matrix[key][len];
    for (int i = 0; i < key; i++) {
        for (int j = 0; j < len; j++) {
            matrix[i][j] = '\0';
        }
    }
    // Fill the matrix with the plaintext characters
    int row = 0, col = 0;
    int direction = 1; // 1 for down, -1 for up
    for (int i = 0; i < len; i++) {
        matrix[row][col] = plaintext[i];
        // Change direction if we reach the top or bottom rail
        if (row == key - 1) {
            direction = -1;
        } else if (row == 0) {
            direction = 1;
```

```
    }

    // Move to the next row in the zigzag pattern
    row += direction;
    col++;
}
// Print the encrypted ciphertext
printf("Encrypted Ciphertext: ");
for (int i = 0; i < key; i++) {
    for (int j = 0; j < len; j++) {
        if (matrix[i][j] != '\0') {
            printf("%c", matrix[i][j]);
        }
    }
}
printf("\n");
// Now, perform decryption using the same ciphertext
printf("Decrypted Plaintext: ");
row = 0;
col = 0;
for (int i = 0; i < len; i++) {
    if (matrix[row][col] != '\0') {
        printf("%c", matrix[row][col]);
    }
    // Change direction if we reach the top or bottom rail
    if (row == key - 1) {
        direction = -1;
    } else if (row == 0) {
        direction = 1;
    }
    // Move to the next row in the zigzag pattern
```

```
        row += direction;
        col++;
    }
    printf("\n");
}
int main() {
    char plaintext[100];
    int key;
    printf("Enter plaintext: ");
    fgets(plaintext, sizeof(plaintext), stdin);
    plaintext[strcspn(plaintext, "\n")] = '\0'; // Remove trailing newline
    printf("Enter key (number of rails): ");
    scanf("%d", &key);
    // Convert plaintext to uppercase (optional but simplifies the encryption)
    for (int i = 0; i < strlen(plaintext); i++) {
        plaintext[i] = toupper(plaintext[i]);
    }
    railFenceEncrypt(plaintext, key);
    return 0;
}
```

Output:

```
Enter plaintext: ATTACK AT ONCE
Enter key (number of rails): 2
Rail Fence Matrix:
A   T   C       T   O   C
  T   A   K   A       N   E

Encrypted Ciphertext: ATC TOCTAKA NE
Decrypted Plaintext: ATTACK AT ONCE

Process returned 0 (0x0)   execution time : 9.368 s
Press any key to continue.
```

➢ Columnar Transposition Cipher

Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX_LEN 1000
void setPermutationOrder(char key[], int keyMap[], int key_len)
{
    // Add the permutation order into array based on alphabetical order
    for (int i = 0; i < key_len; i++)
    {
        keyMap[i] = i;
    }
    // Perform the Bubble Sort to rearrange the order based on the keyword
    for (int i = 0; i < key_len - 1; i++)
    {
        for (int j = 0; j < key_len - i - 1; j++)
        {
            if (key[keyMap[j]] > key[keyMap[j + 1]])
            {
                int temp = keyMap[j];
                keyMap[j] = keyMap[j + 1];
                keyMap[j + 1] = temp;
            }
        }
    }
}
void printMatrix(char msg[], char key[])
{
    int row, col, j;
```

```
int keyMap[256] = {0};
int key_len = strlen(key);
setPermutationOrder(key, keyMap, key_len);
// Calculate column of the matrix
col = key_len;
// Calculate Maximum row of the matrix
row = strlen(msg) / col;
if (strlen(msg) % col)
    row += 1;
char matrix[row][col];
for (int i = 0, k = 0; i < row; i++)
{
    for (int j = 0; j < col;)
    {
        if (msg[k] == '\0')
        {
            // Adding the padding character '_'
            matrix[i][j] = '_';
            j++;
        }
        if (isalpha(msg[k]) || msg[k] == ' ')
        {
            // Adding only space and alphabet into matrix
            matrix[i][j] = msg[k];
            j++;
        }
        k++;
    }
}
printf("Rail Fence Matrix:\n");
for (int i = 0; i < row; i++)
```

```
    {
        for (int j = 0; j < col; j++)
        {
            printf("%c ", matrix[i][j]);
        }
        printf("\n");
    }
}
// Encryption
void encryptMessage(char msg[], char key[], char cipher[])
{
    int row, col, j;
    int keyMap[256] = {0};
    int key_len = strlen(key);
    setPermutationOrder(key, keyMap, key_len);
    // Calculate column of the matrix
    col = key_len;
    // Calculate Maximum row of the matrix
    row = strlen(msg) / col;
    if (strlen(msg) % col)
        row += 1;
    char matrix[row][col];
    for (int i = 0, k = 0; i < row; i++)
    {
        for (int j = 0; j < col;)
        {
            if (msg[k] == '\0')
            {
                // Adding the padding character '_'
                matrix[i][j] = '_';
                j++;
```

```
        }

        if (isalpha(msg[k]) || msg[k] == ' ')
        {
            // Adding only space and alphabet into matrix
            matrix[i][j] = msg[k];
            j++;
        }
        k++;
      }
    }
    int index = 0;
    for (int l = 0, j; l < key_len;)
    {
      j = keyMap[l++];
      for (int i = 0; i < row; i++)
      {
        if (isalpha(matrix[i][j]) || matrix[i][j] == ' ' || matrix[i][j] == '_')
            cipher[index++] = matrix[i][j];
      }
    }
    cipher[index] = '\0';
}
// Driver Program
int main()
{
    // Message
    char msg[MAX_LEN];
    printf("Enter plaintext: ");
    fgets(msg, sizeof(msg), stdin);
    msg[strcspn(msg, "\n")] = '\0'; // Remove trailing newline
```

```
    // Keyword
    char key[MAX_LEN];
    printf("Enter keyword: ");
    fgets(key, sizeof(key), stdin);
    key[strcspn(key, "\n")] = '\0'; // Remove trailing newline
    // Print the Rail Fence Matrix
    printMatrix(msg, key);
    // Print the key in number format
    int keyMap[256] = {0};
    int key_len = strlen(key);
    setPermutationOrder(key, keyMap, key_len);
    printf("Key in number format: ");
    for (int i = 0; i < key_len; i++)
    {
        printf("%d ", keyMap[i]);
    }
    printf("\n");
    // Encryption
    char cipher[MAX_LEN];
    encryptMessage(msg, key, cipher);
    printf("Encrypted Message: %s\n", cipher);
    return 0;
}
```

Output:

```
Enter plaintext: GEEKS ON WORK
Enter keyword: HACK
Rail Fence Matrix:
G E E K
S   O N
  W O R
K _ _ _
Key in number format: 1 2 0 3
Encrypted Message: E W_EOO_GS KKNR_

Process returned 0 (0x0)   execution time : 30.378 s
Press any key to continue.
```

Practical-7

Aim: To implement Simplified Data Encryption Standard.

Solution:

Code:

```
#include <stdio.h>
int main() {
    int key[10];
    printf("Enter a 10-bit key (10 integers): ");
    for (int i = 1; i <= 10; i++) {
        scanf("%d", &key[i]);
    }
    int arranged_key[10];
    arranged_key[1] = key[3]; // k3
    arranged_key[2] = key[5]; // k5
    arranged_key[3] = key[2]; // k2
    arranged_key[4] = key[7]; // k7
    arranged_key[5] = key[4]; // k4
    arranged_key[6] = key[10];// k10
    arranged_key[7] = key[1]; // k1
    arranged_key[8] = key[9]; // k9
    arranged_key[9] = key[8]; // k8
    arranged_key[10] = key[6];// k6
    printf("\nOriginal 10-bit key: ");
    for (int i = 1; i <= 10; i++) {
        printf("%d ", key[i]);
    }
    printf("\nArranged 10-bit key: ");
    for (int i = 1; i <= 10; i++) {
        printf("k%d:%d ", i, arranged_key[i]);
    }
```

```
printf("\nAfter P10, we get 10-bit key: ");
for (int i = 1; i <= 10; i++) {
    printf("%d ", arranged_key[i]);
}
printf("\n**************************************************\n");
// Divide the key into 2 halves and apply one bit left-shift
int left_half[5], right_half[5];
for (int i = 1; i <= 5; i++) {
    left_half[i] = arranged_key[i];
    right_half[i] = arranged_key[i + 5];
}
printf("\nLeft-half key: ");
for (int i = 1; i <= 5; i++) {
    printf("%d ", left_half[i]);
}
printf("\nRight-half key: ");
for (int i = 1; i <= 5; i++) {
    printf("%d ", right_half[i]);
}
int LS1_key[10];
for (int i = 1; i <= 5; i++) {
    LS1_key[i] = left_half[(i % 5) + 1];
    LS1_key[i + 5] = right_half[(i % 5) + 1];
}
printf("\nAfter applying one bit left-shift: ");
for (int i = 1; i <= 10; i++) {
    printf("%d ", LS1_key[i]);
}
printf("\n**************************************************\n");
// P8 Permutation to obtain an 8-bit key
int eight_bit_key[8];
```

```
    eight_bit_key[1] = LS1_key[6];
    eight_bit_key[2] = LS1_key[3];
    eight_bit_key[3] = LS1_key[7];
    eight_bit_key[4] = LS1_key[4];
    eight_bit_key[5] = LS1_key[8];
    eight_bit_key[6] = LS1_key[5];
    eight_bit_key[7] = LS1_key[10];
    eight_bit_key[8] = LS1_key[9];
    printf("\nAfter P8, we get 8-bit key: ");
    for (int i = 1; i <= 8; i++) {
        printf("%d ", eight_bit_key[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
Enter a 10-bit key (10 integers): 1
0
1
0
0
0
0
0
1
0

Original 10-bit key: 1 0 1 0 0 0 0 0 1 0
Arranged 10-bit key: k1:1 k2:0 k3:0 k4:0 k5:0 k6:0 k7:1 k8:1 k9:0 k10:0
After P10, we get 10-bit key: 1 0 0 0 0 0 1 1 0 0
*****************************************************

Left-half key: 1 0 0 0 0
Right-half key: 0 1 1 0 0
After applying one bit left-shift: 0 0 0 0 1 1 1 0 0 0
*****************************************************

After P8, we get 8-bit key: 1 0 1 0 0 1 0 0

Process returned 0 (0x0)   execution time : 53.088 s
Press any key to continue.
```

Practical-8

Aim: To implement Diffi-Hellman Key Exchange method.

Solution:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main() {
    long long int prime, primitive_root, private_key_A, private_key_B;
    printf("Enter a prime number (shared): ");
    scanf("%lld", &prime);
    printf("Enter a primitive root modulo %lld (shared): ", prime);
    scanf("%lld", &primitive_root);
    printf("Enter private key for User A: ");
    scanf("%lld", &private_key_A);
    printf("Enter private key for User B: ");
    scanf("%lld", &private_key_B);
    // Calculate public keys using pow() function
    long long int public_key_A = fmod(pow(primitive_root, private_key_A), prime);
    long long int public_key_B = fmod(pow(primitive_root, private_key_B), prime);
    // Shared secret key calculation
    long long int secret_key_A = fmod(pow(public_key_B, private_key_A), prime);
    long long int secret_key_B = fmod(pow(public_key_A, private_key_B), prime);
    printf("\nPublic Key for User A: %lld\n", public_key_A);
    printf("Public Key for User B: %lld\n", public_key_B);
    printf("Shared Secret Key for User A: %lld\n", secret_key_A);
    printf("Shared Secret Key for User B: %lld\n", secret_key_B);
    if (secret_key_A == secret_key_B) {
        printf("\nShared secret keys match. Secure communication established!\n");
    } else {
```

```
        printf("\nShared secret keys do not match. Secure communication failed.\n");
    }
    return 0;
}
```

Output:

```
Enter a prime number (shared): 71
Enter a primitive root modulo 71 (shared): 7
Enter private key for User A: 5
Enter private key for User B: 5

Public Key for User A: 51
Public Key for User B: 51
Shared Secret Key for User A: 41
Shared Secret Key for User B: 41

Shared secret keys match. Secure communication established!

Process returned 0 (0x0)   execution time : 16.065 s
Press any key to continue.
```

Practical-9

Aim: To implement RSA encryption-decryption algorithm.

Solution:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// Function to calculate the greatest common divisor (GCD) of two numbers
long long int gcd(long long int a, long long int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}
// Function to calculate the modular exponentiation (base^exponent % modulus)
long long int mod_pow(long long int base, long long int exponent, long long int modulus) {
    long long int result = 1;
    base = base % modulus;
    while (exponent > 0) {
        if (exponent % 2 == 1) {
            result = (result * base) % modulus;
        }
        base = (base * base) % modulus;
        exponent = exponent / 2;
    }
    return result;
}
int main() {
    long long int p, q, n, phi_n, e, d;
    long long int message, encrypted, decrypted; // Declare message, encrypted, and decrypted as long
```

```
long int
    // Input prime numbers p and q
    printf("Enter first prime number p: ");
    scanf("%lld", &p);
    printf("Enter second prime numbers q: ");
    scanf("%lld", &q);
    // Calculate n and phi(n)
    n = p * q;
    printf("The value of n: %lld\n", n);
    phi_n = (p - 1) * (q - 1);
    printf("The value of phi_n: %lld\n", phi_n);
    // Choose a public key 'e' such that 1 < e < phi(n) and gcd(e, phi(n)) = 1
    printf("Enter a public key (e): ");
    scanf("%lld", &e);
    if (e < 2 || e >= phi_n || gcd(e, phi_n) != 1) {
        printf("Invalid public key 'e'.\n");
        return 1;
    }
    // Calculate the private key 'd' using the formula d = (1 + k * phi(n)) / e
    long long int k = 1;
    d = (1 + k * phi_n) / e;
    while ((1 + k * phi_n) % e != 0) {
        k++;
        d = (1 + k * phi_n) / e;
    }
    printf("The value of d: %lld\n", d);
    printf("Enter a message to encrypt (numeric form): ");
    scanf("%lld", &message); // Read a long long integer
    // Encrypt the message
    encrypted = mod_pow(message, e, n);
```

```
    printf("Encrypted message: %lld\n", encrypted);
    // Decrypt the message
    decrypted = mod_pow(encrypted, d, n);
    printf("Decrypted message: %lld\n", decrypted);
    return 0;
}
```

Output:

```
Enter first prime number p: 3
Enter second prime numbers q: 11
The value of n: 33
The value of phi_n: 20
Enter a public key (e): 3
The value of d: 7
Enter a message to encrypt (numeric form): 5
Encrypted message: 26
Decrypted message: 5

Process returned 0 (0x0)   execution time : 20.404 s
Press any key to continue.
```

Practical-10

Aim: Demonstrate and perform various encryption-decryption techniques with cryptool.

Solution:

Cryptool is an open-source and freeware program that can be used in various aspects of cryptographic and cryptanalytic concepts. There are no other programs like it available over the internet where you can analyze the encryption and decryption of various algorithms. This tool provides graphical interface, better documentation to achieve the encryption and decryption, bundles of analytic tools, and several algorithms.

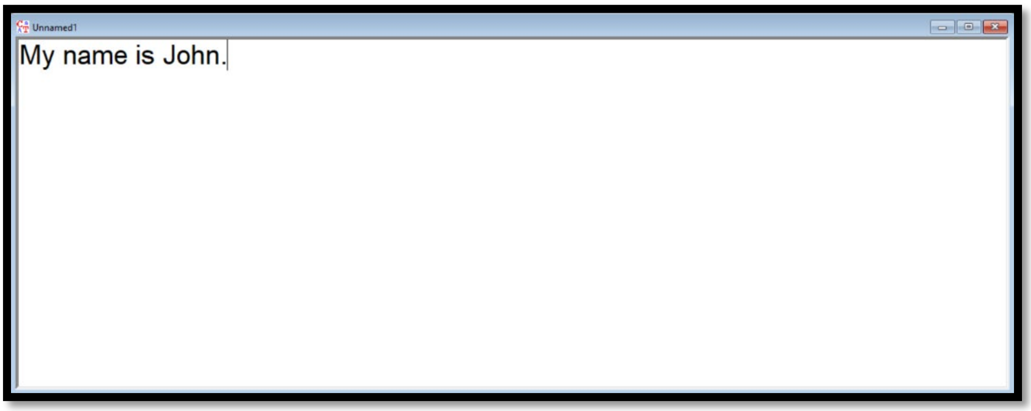
What is Cryptool?

- A freeware program with graphical user interface (GUI).
- A tool for applying and analyzing cryptographic algorithms.
- With extensive online help, it's understandable without deep crypto knowledge.
- Contains nearly all state-of-the-art crypto algorithms.
- “Playful” introduction to modern and classical cryptography.
- Not a “hacker" tool.

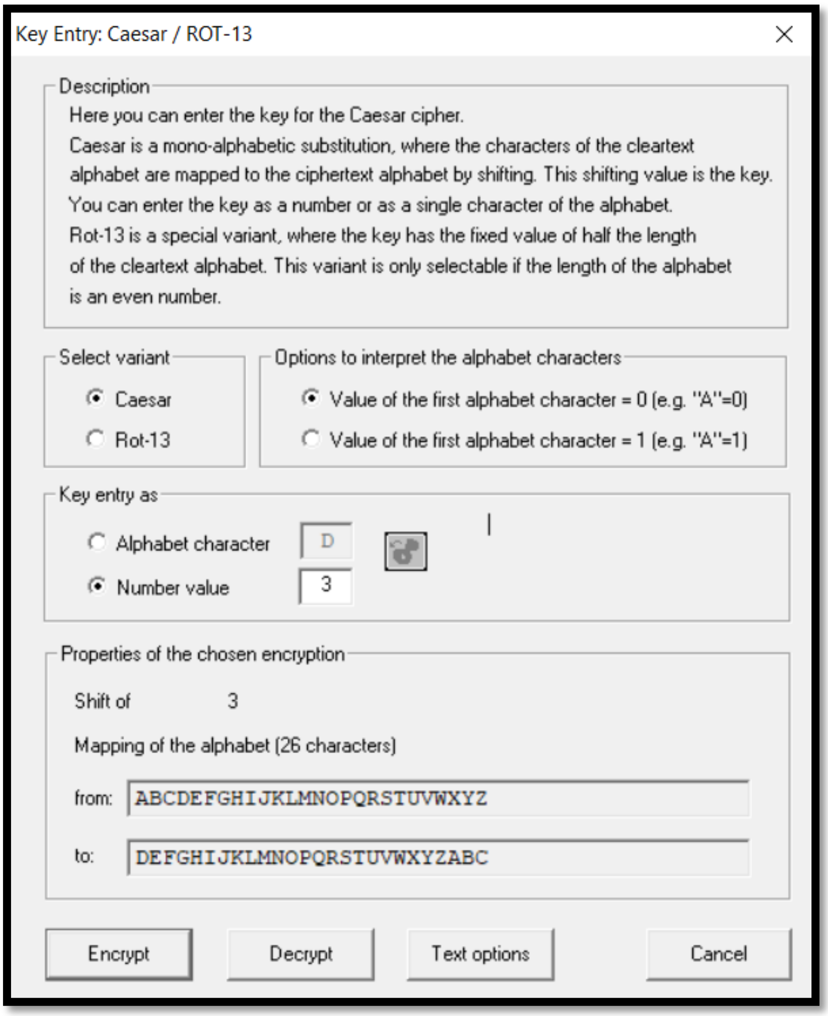
1. Encryption and Decryption of Caesar Cipher:

Here, we will implement an encryption and decryption of Caesar Cipher, which is a substitution method of cryptography. The Caesar Cipher involves replacing each letter of the alphabet with a letter – placed down or up according to the key given. To start with the process, you must move to the **Encrypt/Decrypt tab** of the program. There, you will find the **Symmetric (Classic) tab - Choose Caesar Cipher**. For further information, you can get guided by the image below. In encryption, we are replacing the plaintext letter with the 3rd letter of the alphabet that is if "A" is our plaintext character, then the Ciphertext will be "D".

Output:



Input Plaintext



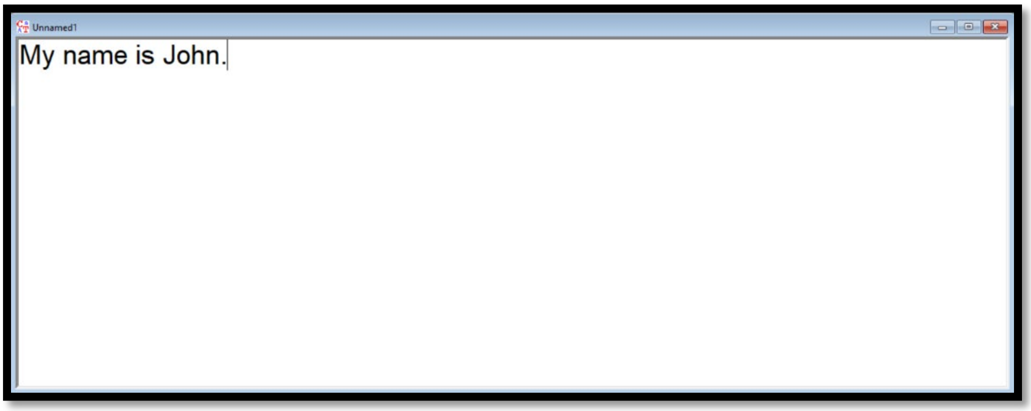
Encryption & Decryption Process

Output Ciphertext

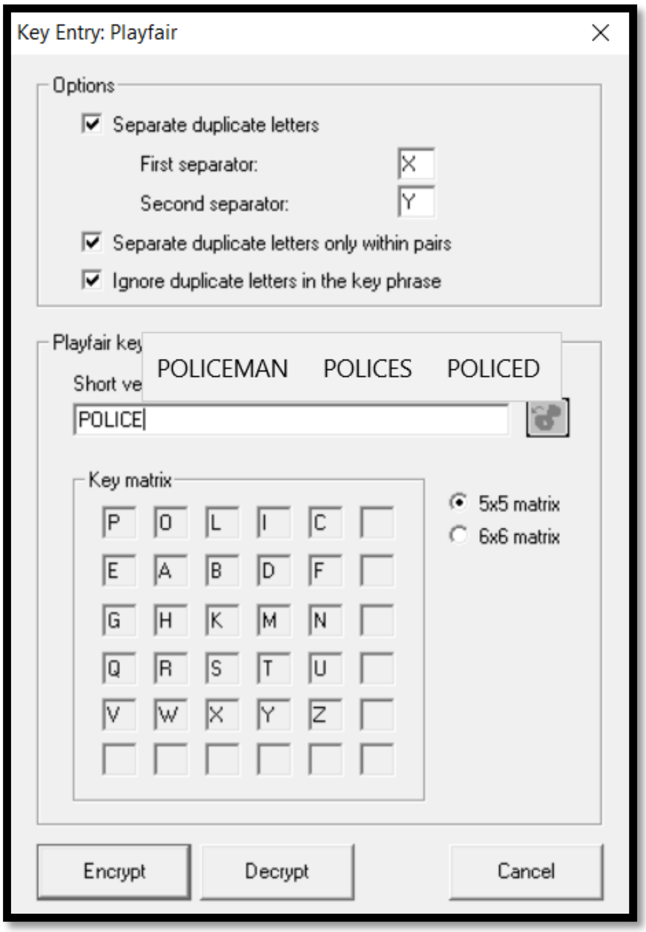
2. Encryption and Decryption of Playfair Cipher:

We must move to **Encrypt/Decrypt - Symmetric - Playfair Cipher** and perform the encryption part. We are putting the same plaintext. So, when we press the encrypt button, we will get Ciphertext.

Output:

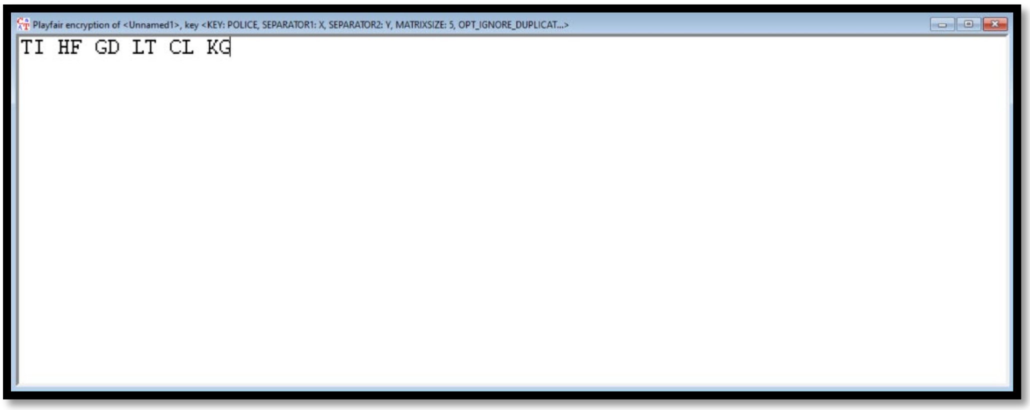


Input Plaintext



Encryption & Decryption Process

Pair of letters made from Plaintext

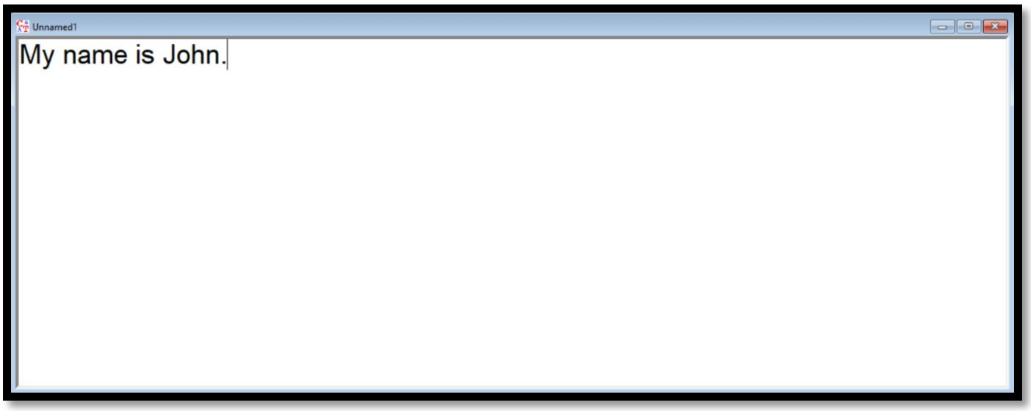


Output Ciphertext

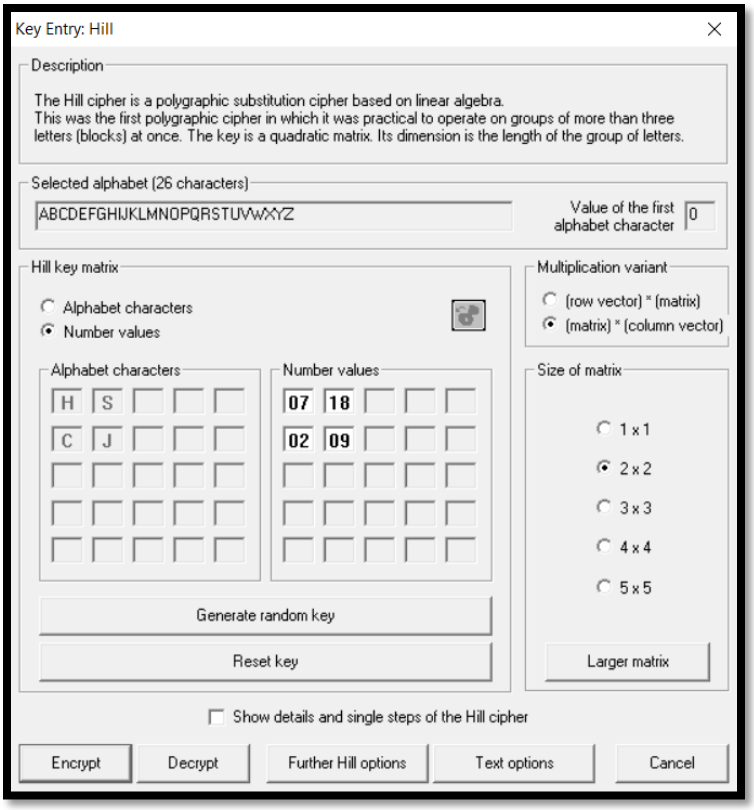
3. Encryption and Decryption of Hill Cipher:

We must move to **Encrypt/Decrypt - Symmetric - Hill Cipher** and perform the encryption part. We are putting the plaintext and assuming that the program gives us the Ciphertext. So, when we press the encrypt button, we will get Ciphertext.

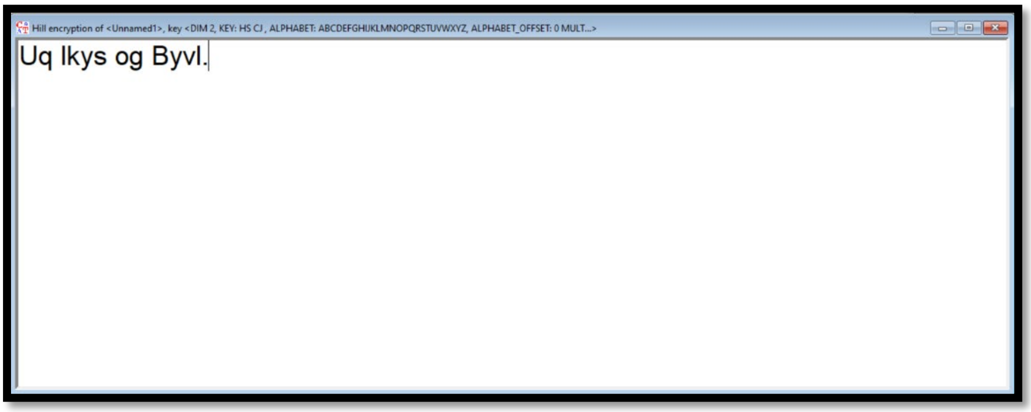
Output:



Input Plaintext



Encryption & Decryption Process

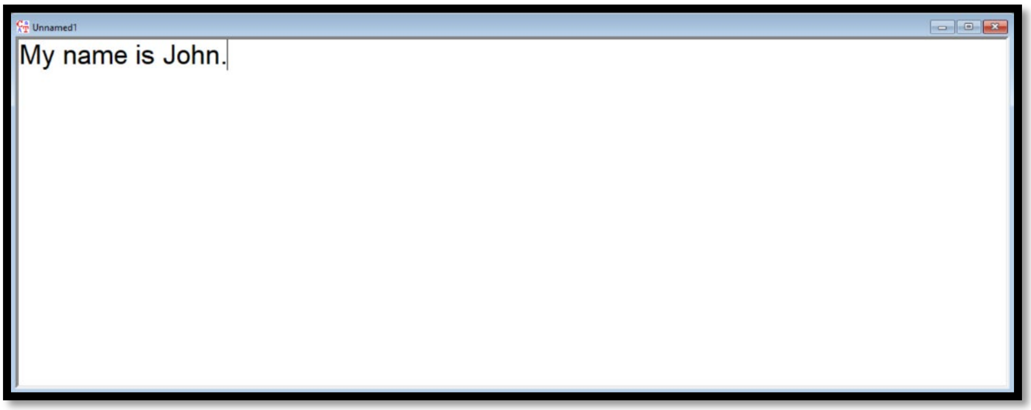


Output Ciphertext

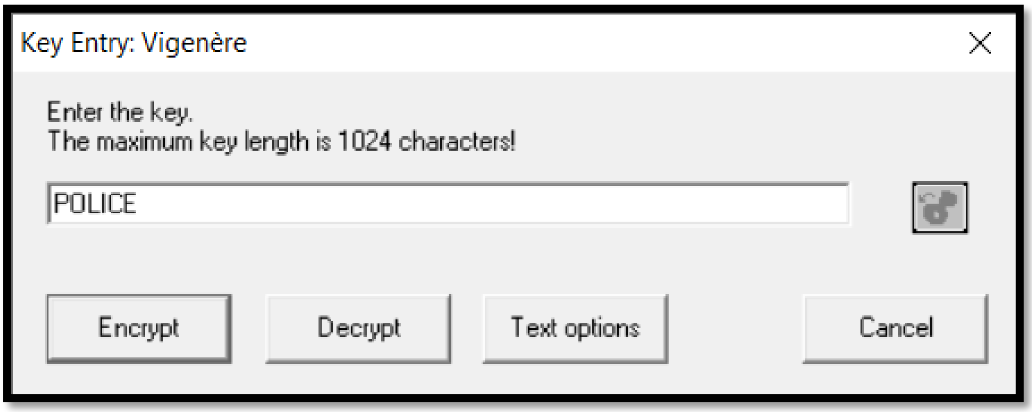
4. Encryption and Decryption of Vigenère Cipher:

We must move to **Encrypt/Decrypt - Symmetric - Vigenère Cipher** and perform the encryption part. We are putting the plaintext and assuming that the program gives us the Ciphertext with the help of key as – POLICE. So, when we press the encrypt button, we will get Ciphertext.

Output:



Input Plaintext



Encryption & Decryption Process

Output Ciphertext

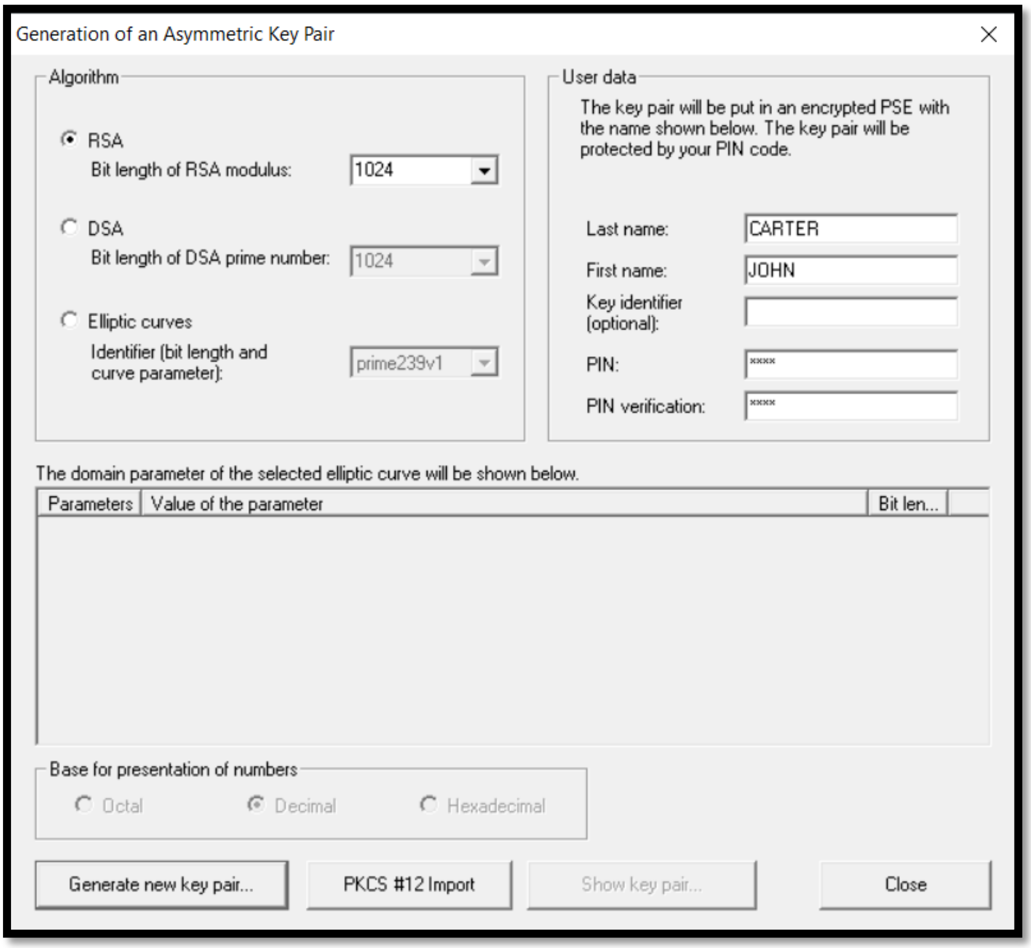
➢ Digital Signatures/PKI

This selection of tools is designed specifically for supporting a user implementing various aspects of public key encryption. More specifically PKI (standing for public key infrastructure) combines a public key with a user identity and so is more applicable to authentication. So, for example, there is a tool in this section that generates a key pair for RSA. This tool incorporates aspects of input user details so that the generated keys are affiliated with the user, and so is more appropriate for authentication.

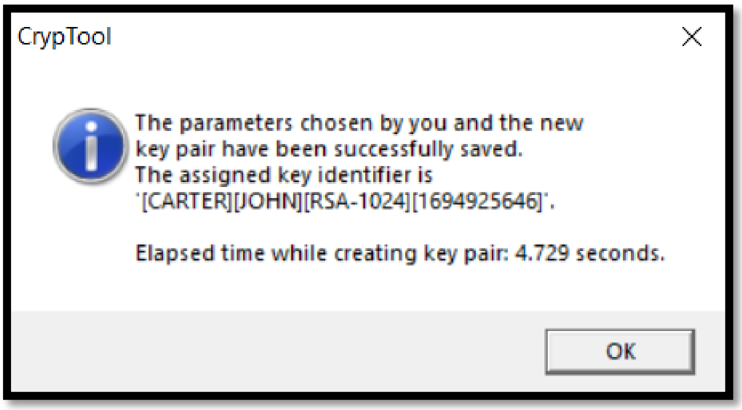
Under this heading the two most useful options are:

- **Digital Signatures/PKI > PKI > Generate/Import Keys** – A tool for creating new asymmetric key pairs.
- **Digital Signatures/PKI > PKI > Display/Export Keys** – This tool is for managing available keys.

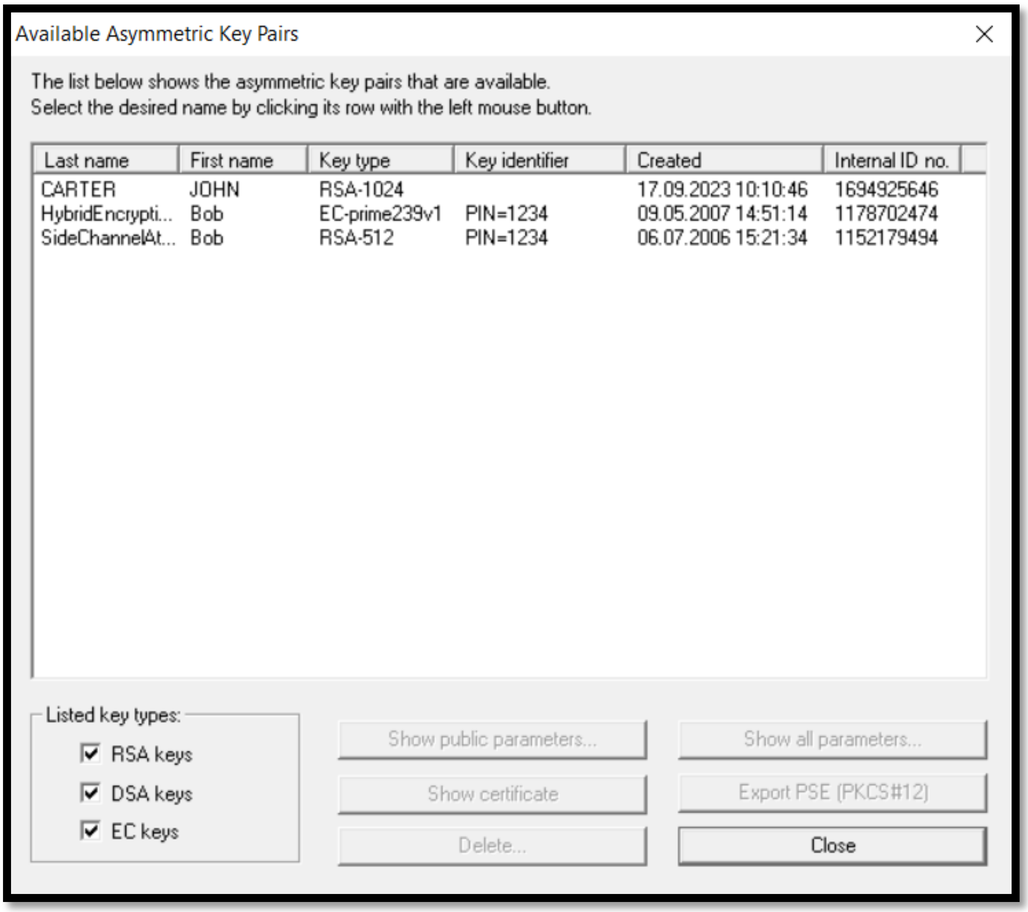
Output:



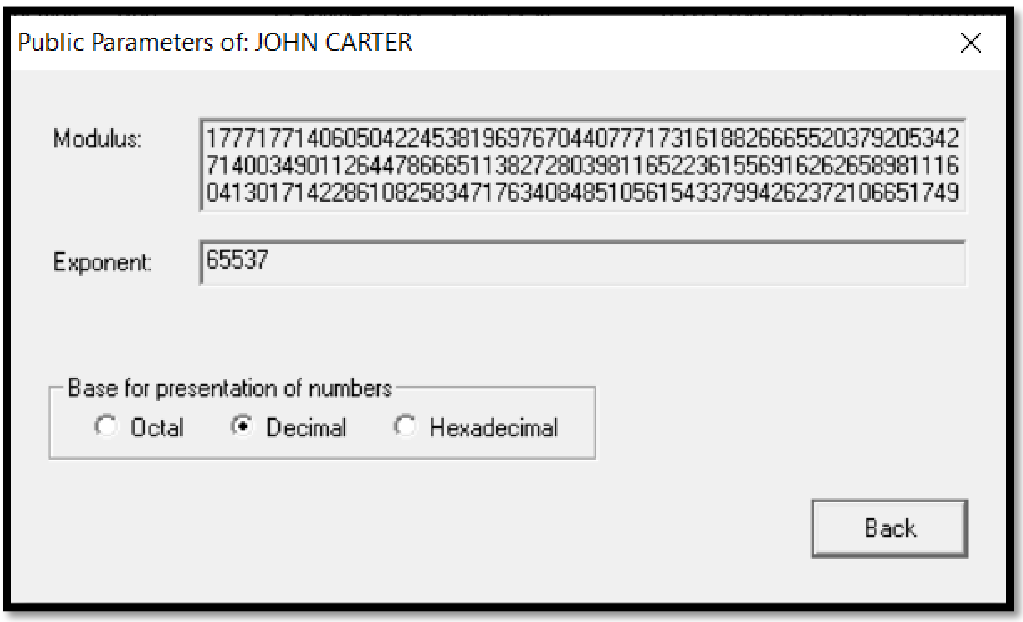
Generation of an Asymmetric Key Pair



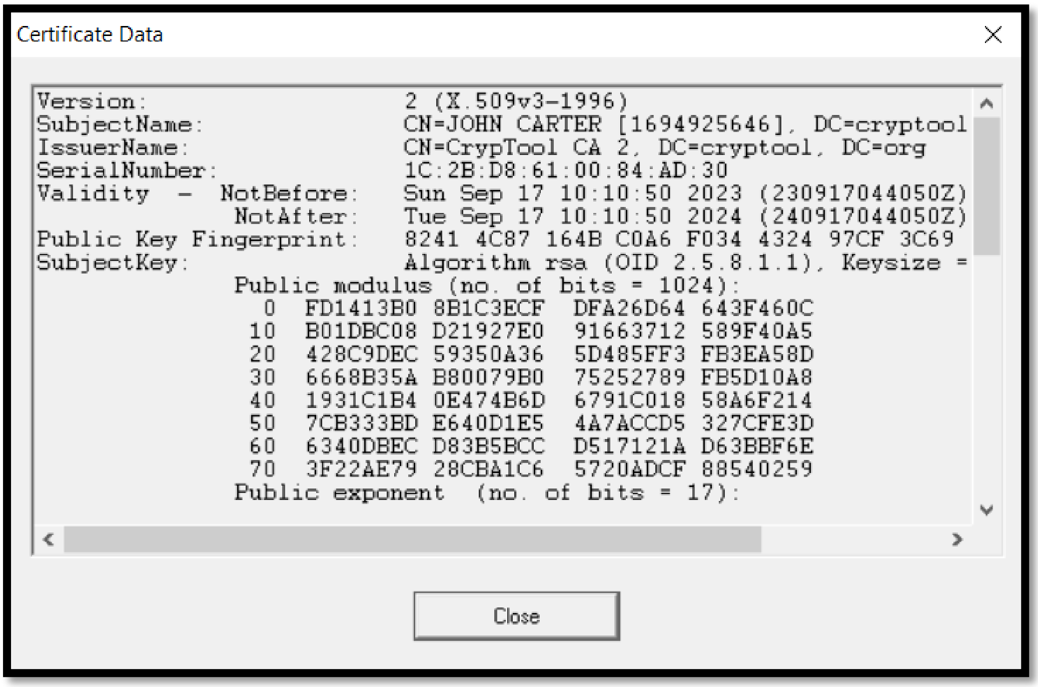
Successfully Generation of Key



Total Available Asymmetric Key Pairs



Public Parameters of Key



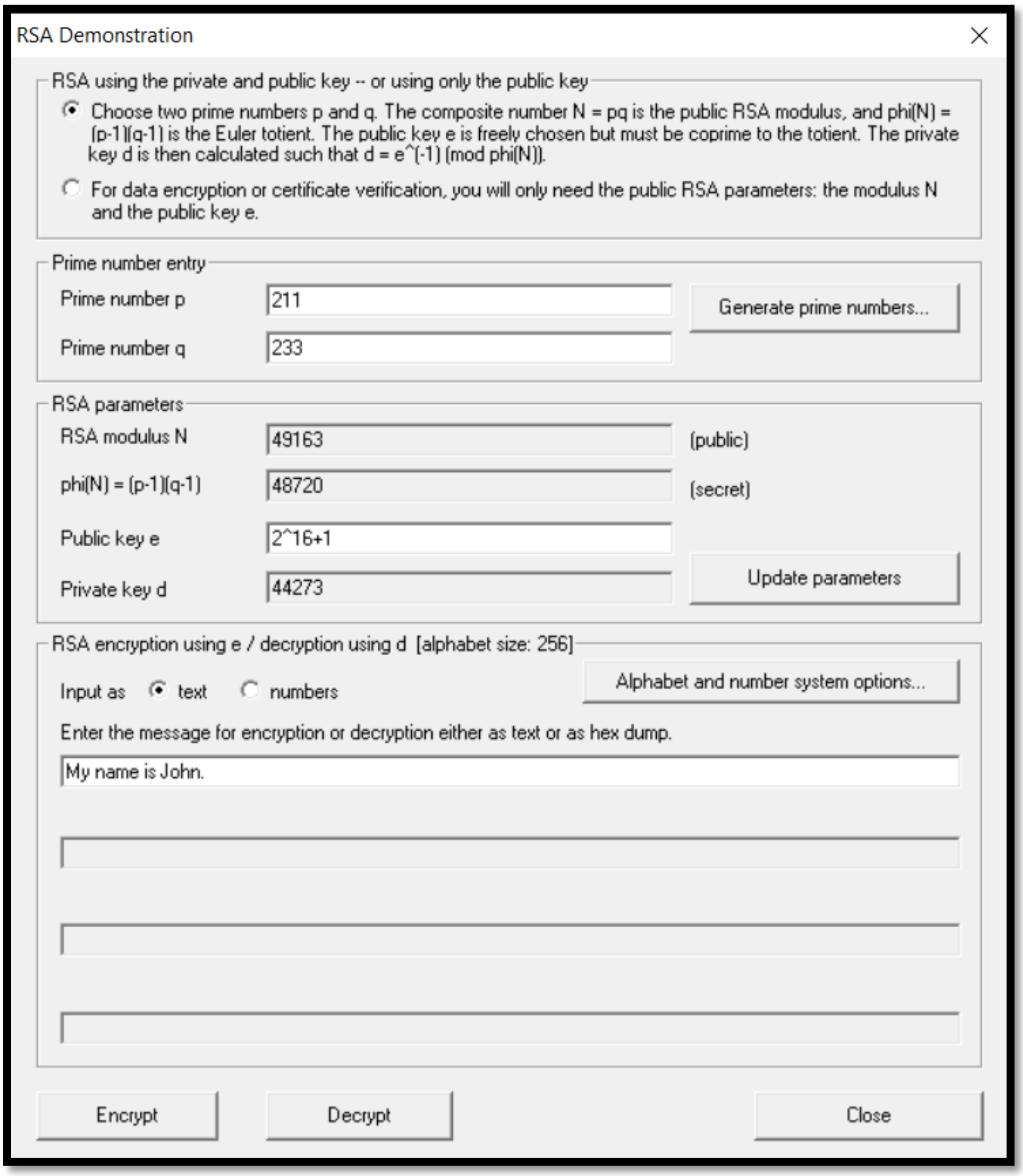
Certificate Data of Key

➢ RSA Demonstration

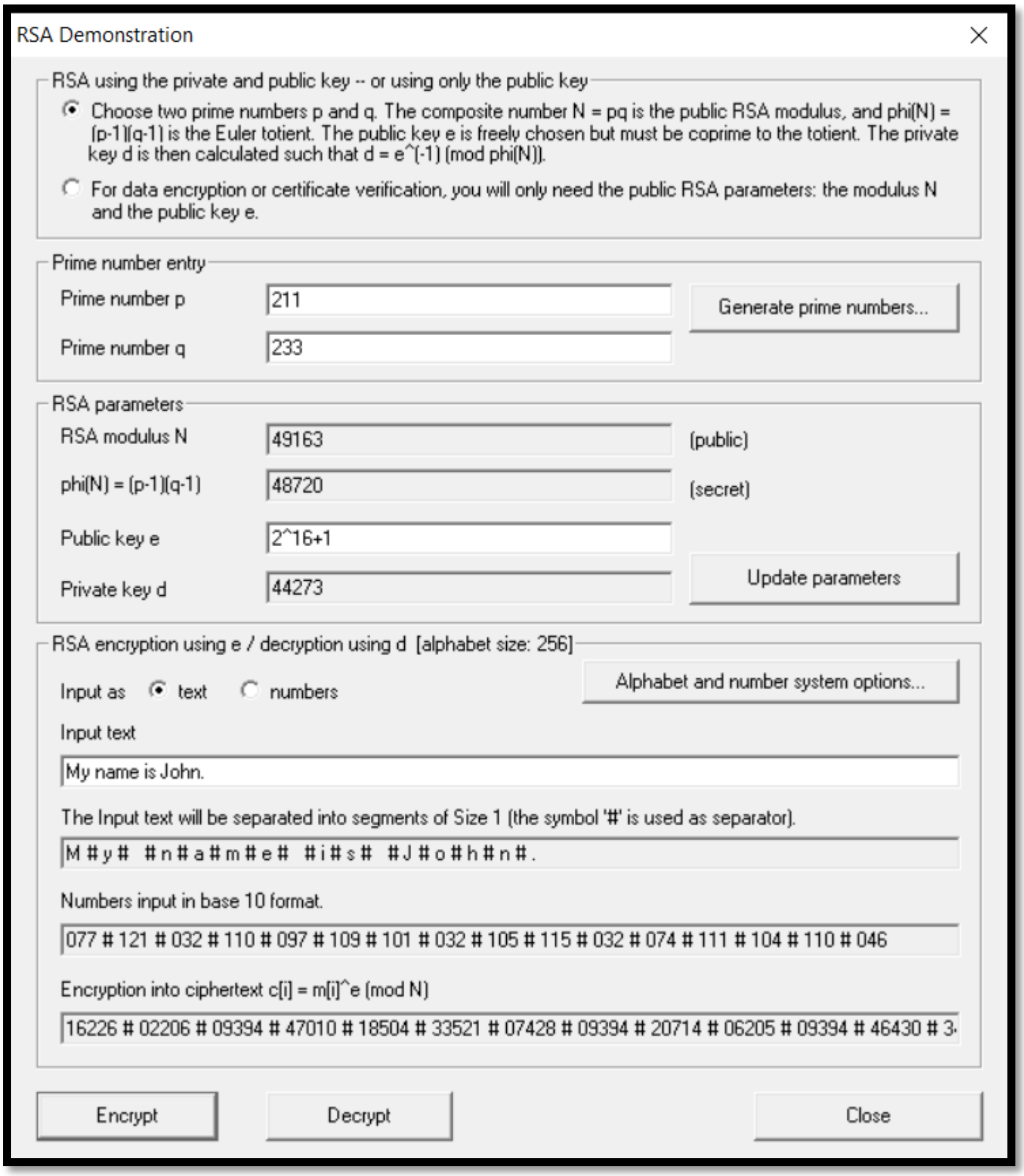
We shall first use a tool to demonstrate its workings. The demonstration material may be accessed through **Indiv. Procedures > RSA Cryptosystem > RSA Demonstration… or Crypt/Decrypt > Asymmetric > RSA Demonstration…**

This presents a window with a series of options.

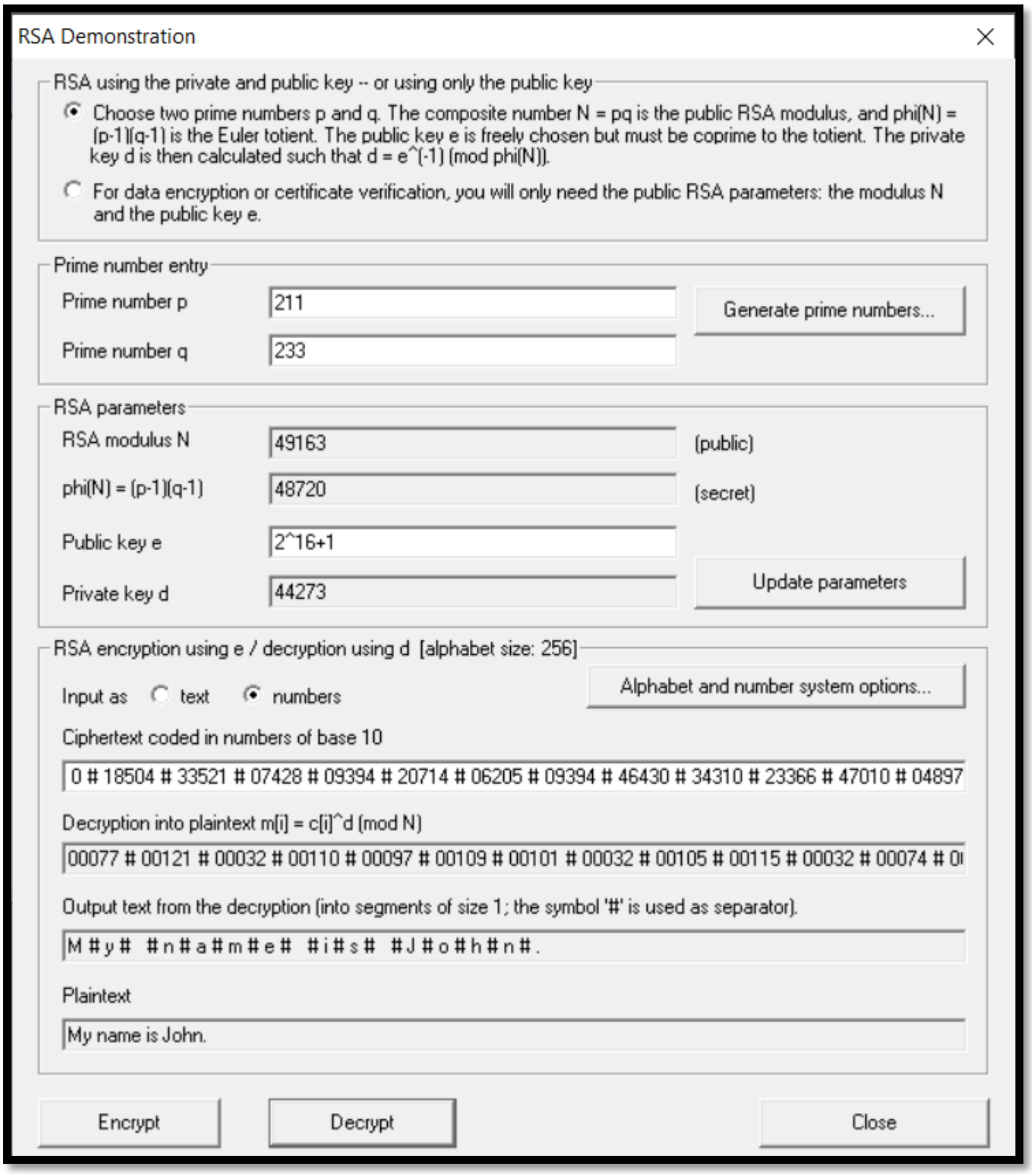
Output:



RSA Demonstration - Encryption & Decryption Process



Obtaining Ciphertext from Plaintext



Obtaining Plaintext from Ciphertext

Practical-11

Aim: Study and use open-source packet analyzer-Wireshark to understand security mechanism of various network protocols.

Solution:

What is Wireshark?

Wireshark is an open-source packet analyzer, which is used for education, analysis, software development, communication protocol development, and network troubleshooting.

It is used to track the packets so that each one is filtered to meet our specific needs. It is commonly called a sniffer, network protocol analyzer, and network analyzer. It is also used by network security engineers to examine security problems.

Wireshark is a free to use application which is used to apprehend the data back and forth. It is often called a free packet sniffer computer application. It puts the network card into an unselective mode, i.e., to accept all the packets which it receives.

Uses of Wireshark:

Wireshark can be used in the following ways:

1. It is used by network security engineers to examine security problems.
2. It allows the users to watch all the traffic being passed over the network.
3. It is used by network engineers to troubleshoot network issues.
4. It also helps to troubleshoot latency issues and malicious activities on your network.
5. It can also analyze dropped packets.
6. It helps us to know how all the devices like laptops, mobile phones, desktop, switch, routers, etc., communicate in a local network or the rest of the world.

What is a packet?

A packet is a unit of data which is transmitted over a network between the origin and the destination. Network packets are small, i.e., maximum 1.5 Kilobytes for Ethernet packets and 64 Kilobytes for IP packets. The data packets in the Wireshark can be viewed online and can be analyzed offline.

History of Wireshark:

In the late 1990's Gerald Combs, a computer science graduate of the University of Missouri-Kansas City was working for the small ISP (Internet Service Provider). The protocol at that time did not complete the primary requirements. So, he started writing ethereal and released the first version around 1998. The Network integration services owned the Ethernet trademark.

Combos still held the copyright on most of the ethereal source code, and the rest of the source code was re-distributed under the GNU GPL. He did not own the Ethereal trademark, so he changed the name to Wireshark. He used the contents of the ethereal as the basis.

Wireshark has won several industry rewards over the years including eWeek, InfoWorld, PC Magazine and as a top-rated packet sniffer. Combos continued the work and released the new version of the software. There are around 600 contributed authors for the Wireshark product website.

Functionality of Wireshark:

Wireshark is like tcpdump in networking. Tcpdump is a common packet analyzer which allows the user to display other packets and TCP/IP packets, being transmitted and received over a network attached to the computer. It has a graphic end and some sorting and filtering functions. Wireshark users can see all the traffic passing through the network.

Wireshark can also monitor the unicast traffic which is not sent to the network's MAC address interface. But the switch does not pass all the traffic to the port. Hence, the promiscuous mode is not sufficient to see all the traffic. The various network taps or port mirroring is used to extend capture at any point.

Port mirroring is a method to monitor network traffic. When it is enabled, the switch sends the copies of all the network packets present at one port to another port.

Features of Wireshark:

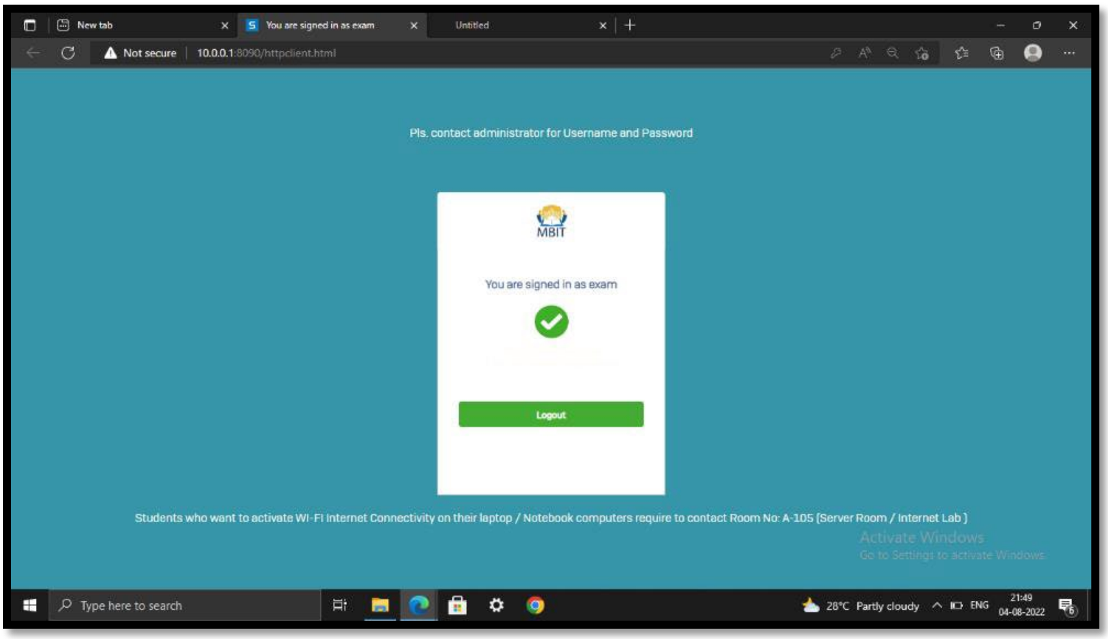
1. It is multi-platform software, i.e., it can run on Linux, Windows, OS X, FreeBSD, NetBSD, etc.
2. It is a standard three-pane packet browser.
3. It performs deep inspection of the hundreds of protocols.
4. It often involves live analysis, i.e., from the different types of the network like the Ethernet, loopback, etc., we can read live data.
5. It has sort and filter options which makes ease to the user to view the data.
6. It is also useful in VoIP analysis.
7. It can also capture raw USB traffic.
8. Various settings, like timers and filters, can be used to filter the output.
9. It can only capture packets on the PCAP (an application programming interface used to capture the network) supported networks.
10. Wireshark supports a variety of well-documented capture file formats such as the PcapNg and Libpcap. These formats are used for storing the captured data.
11. It is the no.1 piece of software for its purpose. It has countless applications ranging from tracing down, unauthorized traffic, firewall settings, etc.

Example of Wireshark:

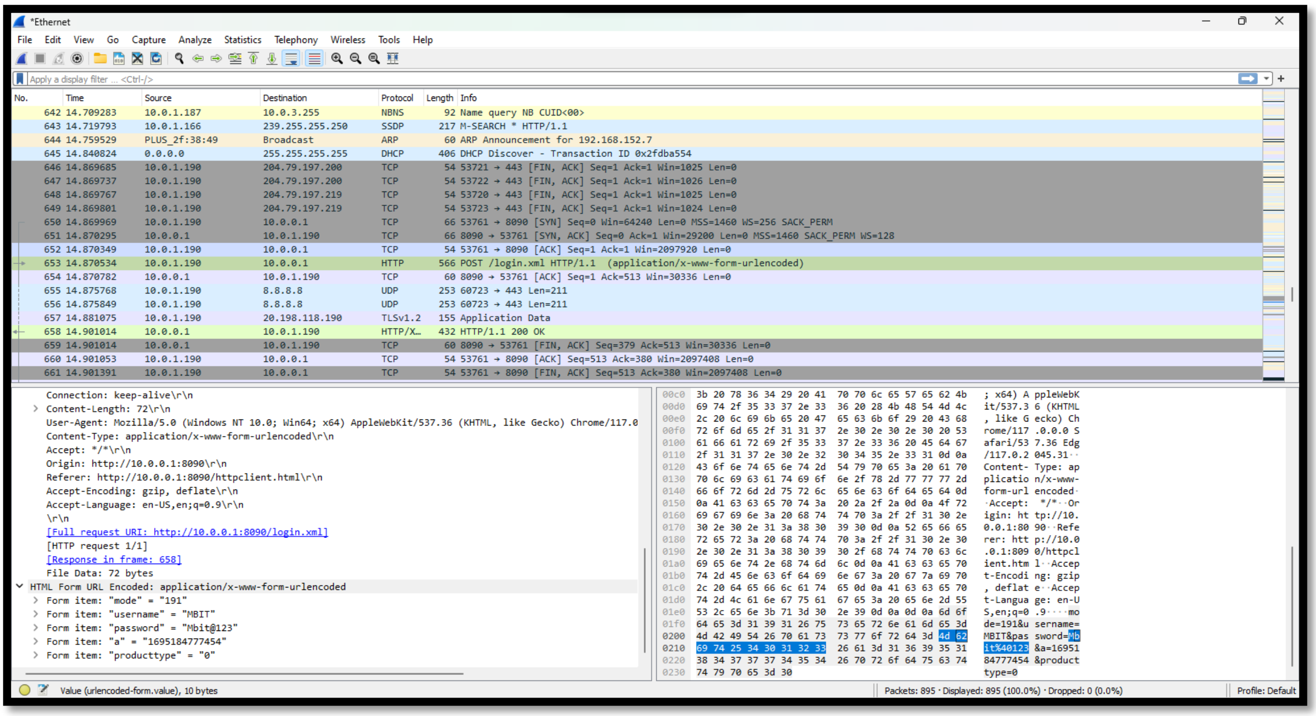
For example, if we open the MBIT localhost website of IP address 10.0.0.1 of port number 8090 and if we do the login with username and password than it is captured in Wireshark as it captures the packet so when we open the Wireshark and filter it with http.

We can see a login http session so we right click on follow the stream than it can show the username and password which I entered in MBIT website. Now click the Html Form URL Encoded which is for user credentials are stored in URL encoded tag, then http web page login details are displayed.

Output:



MBIT Login Page



Practical-12

Aim: Detail Case study: Real world implementation of Network Security Algorithm/Concept.

Solution:

A Case Study in Testing a Network Security Algorithm

1. INTRODUCTION

Testing software properly to demonstrate that it performs as expected is a very time-consuming and difficult process. However, in general the inputs, outputs, and expected behavior are known before the software is deployed. In the case of software intended to perform security functions, this may not be the case. This is especially true of security algorithms that are expected to detect security events on a host or network, such as intrusion or anomaly detection systems, worm detection algorithms, and behavioral analysis systems.

Testing security detection algorithms, particularly ones that use net-work data as input, is complicated because the input can be extremely variable. Network traffic varies considerably by time of day, time of year, type of network (e.g., university, government, corporation), size of network, and enforced security policies. Further, network traffic is continually evolving as new applications are developed. Even assuming no malicious data is present in the network and testing the algorithm purely for false positives, it is not possible to test against every possible combination of factors given the variability in possible deployment environments and of network traffic itself.

The two approaches that are most used for testing net-work security algorithms are simulation (particularly the use of the Lincoln Labs [8] dataset) and network traces [2]. Simulation has the advantage that, if the variables required can be identified, variability can be included in the dataset and the algorithm tested against it. However, a simulation may not necessarily be representative of actual data and may contain unintended biases. While network traces do not have this issue, it cannot be guaranteed that they contain the events of interest, nor is the location of such events known (thus making false negatives difficult or impossible to determine). While this can potentially be resolved through the use of red team to perform security experiments against the monitored network, this is generally not a viable option due to both resource requirements and legal limitations.

2. EVALUATION METHODOLOGIES

Traditional intrusion detection research has not focused on developing appropriate evaluation methodologies. The result is that the capability of detection algorithms has tended to be presented in terms of either the Lincoln Labs dataset [8] or the use of proprietary network traces accessible only to the authors of a particular system. As noted by Athanasies et al. [2], the result is that the "ad-hoc methodology that is prevalent in today's testing and evaluation of network intrusion detection algorithms and systems makes it difficult to compare different algorithms and approaches." The two approaches can be summed up as using either real data or simulation, and both approaches are discussed in more detail below, along with a third option — emulation — that is less widely used.

3. EMULATION APPROACH

Simulation has a definite advantage in that it provides labeled data, and that the characteristics of the attack or network can be controlled. I aim to maintain these advantages while using real network traces. In order to achieve this, I perform the attacks on a network testbed, capturing the resulting traffic for analysis. By doing this I can control exactly the characteristics of the attack. This step is the same as that performed by Lincoln Labs in generating their attack data [8], however they generated their attacks considering only a single network. Given the flexibility of network testbeds, multiple network configurations can be examined (based on the availability of network traces, which is discussed further below). Further, any characteristics that can be controlled in an attack (such as the number of attacking hosts or the attacking algorithm) can be varied so that a detector can be tested in a systematic manner.

4. CASE STUDY

A case study is presented in this section that indicates how the emulation approach can be used to test a security detector. The problem domain for which the detector was developed — that of detecting-ordinated scans — is presented first. This is followed by a brief description of the detection algorithm, presented to demonstrate the variables that might impact on its detection capability. The emulation data generated, based on a combination of attacks generated on a testbed and actual network traces, is described with focus on how the variables that might affect the detector were determined and varied. The last subsection provides a brief summary of the results obtained, demonstrating that the capabilities of a security detector can be modeled when examined in a systematic fashion, and that this model implies how the detector will perform in other circumstances.

4.1 Problem Domain

The hypothesis that was tested was that it was possible to detect the presence of coordinated scanning activity. Scanning consists of a series of probes against a target system or network, where a probe is a reconnaissance activity aimed at a single target. A target here could be a single host, or a particular service on a particular host. Reconnaissance activity consists of the attempt to determine if the target exists. For example, someone could send an ICMP ping to a particular host. In this case, the ping is the reconnaissance activity, and the host is the target. Alternatively, someone could send a single SYN packet to port 80 on a particular host. Here the reconnaissance activity is the attempt to determine if a web server is present on that host. When an adversary probes multiple targets, this is a scan.

4.2 Detection Algorithm

The algorithm developed to detect the presence of coordinated scans focused on scans performed against a network — either horizontal or strobe scans. It assumed that a scan detection system was already in place to monitor the target network, and that the scans performed as part of the coordinated activity are detected by that system. Thus, the input to the algorithm is the set of detected over some period. In particular, the algorithm required for each scan the source of the scan and each scan target (IP/port pair).

Given this input, the algorithm was inspired by the set covering problem, where the goal is to determine if there is some subset of the scans that, when combined, cover the same port or ports across a contiguous portion of the network. The set covering problem consists of determining the minimum number of sets required to cover an entire space. In contrast, this algorithm focused finding set of sets (scans) such that the coverage of the space (network) was maximized while the overlap between sets (scans) was minimized. This algorithm is described in detail by Gates [4].

There are three variables that are controlled by the administrator who is using this algorithm: scan window, coverage, and overlap. The scan window refers to the number of scans that are input for analysis to determine if any coordinated scans are present. Coverage refers to the percentage of the target network that the adversary has scanned. The administrator can specify if he only wants to detect those adversaries who have scanned the entire network, or some minimum portion of it. Overlap refers to the amount of overlap between the scans. This variable was added to ensure that the adversary could not avoid detection by having sources that can overlapping targets. The administrator can set the maximum amount of overlap he expects an adversary to use.

4.3 Generating Test Data Sets

The DETER network1, which is based on Emu lab software2[16],is a testbed that is focused on supporting network security research. Figure 1 demonstrates a typical network setup for testing the security algorithm described in Subsection 4.2. In this case there are five agents (scanning clients), one handler (who controls the five agents), two hosts emulating a /16 network, and a monitoring host positioned between the scanning hosts and the scanned network. The monitoring host was running tcp dump to capture all the network traffic generated by the scanning hosts.

The algorithm being tested could be reasonably varied in seven different dimensions. The first three are coverage, overlap and scan window, which are controllable by the security administrator and discussed in Subsection 4.2. The adversary can also choose scan characteristics that might affect the detection capability of the algorithm, such as the number of sources involved in the coordinated scan, the number of ports being scanned, and the scanning algorithm being used. The scanning algorithm refers to the algorithm used to distribute the scan destinations amongst the scanning hosts (two algorithms were available for these tests). The last variable is the size of the network being scanned, as the detection capability might be different between, for example, a /24 network and a /16 network. In this case, the network size was treated as a discrete value, with only /24 and /16 networks being considered. This limitation was placed on the data due to the characteristics of the available network traces.

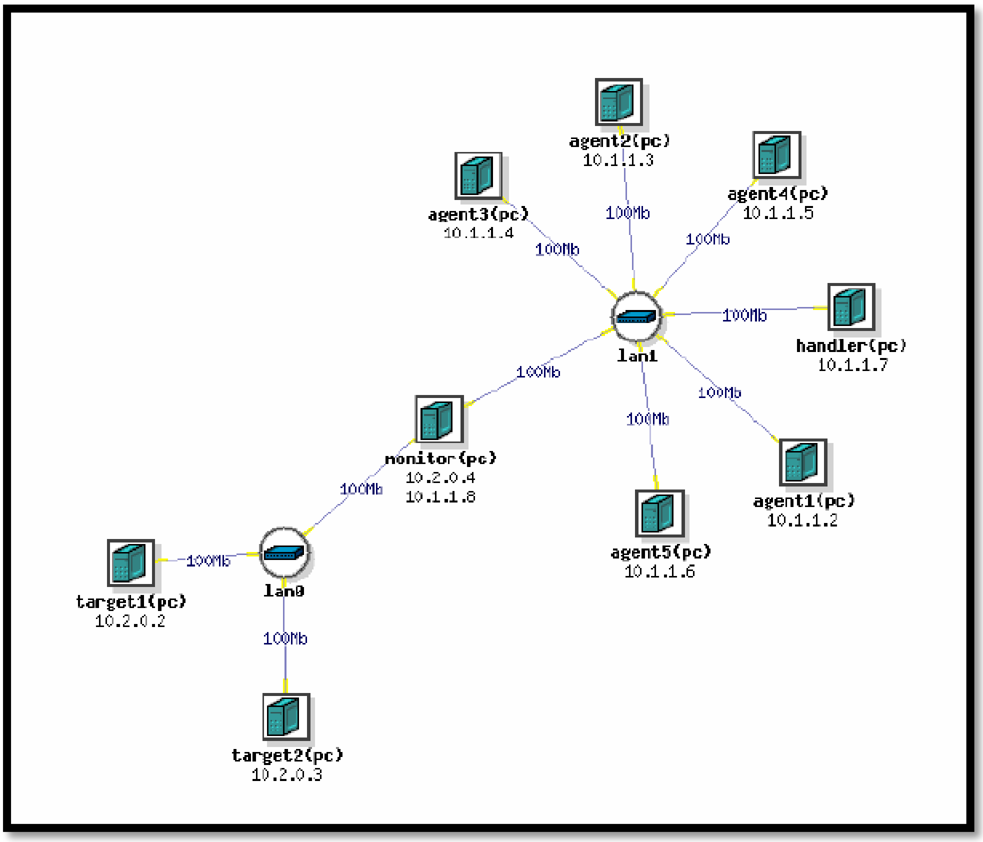


Figure 1: Co-ordinated port scan DETER set up with 5 agents,1 handler and a /16 subnet.

4.4 Testing Results

I analyzed the results from our experiments using detection rate and false positive rate as described by Axelsson [3]. Here the detection rate is the probability of generating an alert, A, given that there was an event, I,P(A|I). The false positive rate is the probability that there was an alert given that there was no event, P(A|¬I).The detection algorithm was run on each of the

116 data sets de-scribed in Section 4.3 with the detection rates and false positive rates calculated for each data set. The detection rate was defined as the number of sources correctly identified as being part of a coordinated scan, while the false positive rate was defined as the number of sources that were incorrectly identified as being part of a coordinated scan. The number of coordinated scans detected was not taken into account. Thus, the detection rate might be 100%with a 0% false positive rate for a particular data set, yet the single coordinated scan was actually detected as multiple coordinated scans. Cases such as this are not analyzed separately in this paper.

$$\hat{P}(\text{co-ordinated scan is detected}) = \frac{e^{\hat{y}}}{1 + e^{\hat{y}}}$$

where ˆyis a weighted summation of the seven variables idented Subsection 4.3. The scanning algorithms were mapped to either zero or one, while the network size was represented by the number of hosts in the subnet (256 or 65536). The number of variables was reduced using the Akaike Information Criterion [1], indicating those variables that contributed most to the detection rate of the algorithm. The sign on the weight for each variable indicates if an increase in the value of the variable causes an increase or decrease in the detection rate.

The false positive rate was similarly modeled, however it used linear regression rather than a logistic regression (as the false positives did not demonstrate a bimodal distribution). The actual results and modeling approach are described in more detail by Gates [4].

This subsection demonstrates that testing an algorithm in a systematic manner can result in a model of how that detector performs. Gates [4] demonstrates in detail how well the model performs at predicting the detector's performance in previously unseen circum-stances through further testing, and further demonstrates how this approach can be used to compare detectors. A complete description of these results is outside the scope of this paper but is briefly provided here to demonstrate the power of using an emulation approach to testing network security detectors.

5. CONCLUSIONS

In this paper I described the two approaches currently used in testing network security detectors: simulation and real network traces. Simulation provides the most control over the test environment, however it is difficult to simulate network traffic [13] and previous attempts [8] have had several flaws [11, 9]. Network traces have the advantage of containing actual data (and so eliminate the problems associated with simulation), however the data is not labeled, and so it is not possible to determine the detection rate as the number of events in the data are not known. Additionally, events of interest may not even be present in the captured traffic. Further, testing in a single environment, be it simulated or real, does not indicate how well an algorithm will perform given a different environment.

I presented an emulation approach to testing network security detectors that is based on a combination of actual network traces and attacks captured from a network testbed. This combined approach provides both realistic background traffic and realistic attacks. It addresses the disadvantage of simulations not having realistic data, as well as the disadvantage of network traces not containing labelled attacks. However, this approach is particularly suited to testing detectors designed for a single event, rather than, for example, generic intrusion detectors.

This approach was demonstrated by testing a co-ordinated scan detector using actual network traces and co-ordinated scans captured from the DETER network security testbed. The results were analysed, demonstrating that they could be modelled using regression equations. The use of the emulation approach allows the researcher to test their algorithm in multiple environments in a controlled and systematic fashion. Further, the use of regression equations provides the potential to extrapolate from the emulation to previously unexplored environments, indicating how the detector might per-form given a new network or attack characteristic.

6. REFERENCES

1. H. Akaike. Information theory as an extension of the maximum likelihood principle. In B. N. Petrov and F. Cassiterites, Proceedings of the 2nd International Symposium on Information Theory, pages 267 – 281, Budapest, Hungary,1973.
2. N. Athanasiades, R. Abler, J. Levine, H. Own, and G. Riley. Intrusion detection testing and benchmarking methodologies. In Proceedings of First IEEE International Workshop on Information Assurance, pages 63 – 72, Darmstadt, Germany, March 2003.
3. S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. ACM Transactions on Information and System Security, 3(3):186 – 205, 2000.
4. C. Gates. Co-ordinated Port Scans: A Model, A Detector, and an Evaluation Methodology. PhD thesis, Dalhousie University, Feb 2006.
5. C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas. More NetFlow tools: For performance and security. In Proceedings of the 18th Large Installation System Administration Conference (LISA 2004), pages121–132, Atlanta, Georgia, USA, November 2004.
6. C. Gates, J. J. McNutt, J. B. Kadane, and M. Kellner. Scan detection on very large networks using logistic regression modeling. In Proceedings of the IEEE Symposium on Computers and Communications, pages 402 – 407,Pula-Cagliari, Sardinia, Italy, June 2006.
7. Intrusec. Intrusec alert: 55808 trojan analysis. http://www.intrusec.com/55808.html, 2003.Last visited: 1 July 2003.
8. R. P. Lippmann et al. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In DARPA Information Survivability Conference and Exposition, volume 2, pages 12 – 26, 2000.
9. M. V. Mahoney and P. K. Chan. An analysis of the 1999DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection, pages 220 – 237, Pittsburgh, PA, USA, September 2003.
10. R. A. Maxion and K. M. Tan. Benchmarking anomaly-based detection systems. In Proceedings of 2000 International Conference on Dependable Systems and Networks, pages623 – 630, June 2000.
11. J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA

intrusion detection system evaluations as performed by Lincoln Laboratory. ACM Transactions on Information and System Security, 3(4):262 –294, 2000.

12. R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pages 27 – 40, Taormina, Sicily, Italy, October 2004.
13. V. Paxson and S. Floyd. Why we don’t know how to simulate the internet. In Proceedings of the 29th conference on Winter simulation, pages 1037–1044, Wisconsin, 1997. ACM Press.
14. S. Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy port scans. Journal of Computer Security, 10(1):105 – 136, 2002.
15. S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical automated detection of stealthy port scans. In Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 2000.
16. B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad,M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distribution systems and networks. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, pages 255 –270, Boston, MA, USA, December 2002. USENIX Association.