Secure Hardware and Embedded Devices

(ELEC6237)

Coursework Answer Sheet

Name: Wei Chien Teoh (Eugene)

Student ID: 28313704

Section A

Task 1

- Change the input vector to IN = [1] and execute the code again. Explain any differences between your results and the one shown above

```
% print keys that matches execution time
for col = 1:length(predTime)
    if predTime(:, col) == excTime
        % key=col-1 because it ranges from 0-15
        fprintf("Time matched key: %i\n", col-1)
    end
end
```

Figure 1

By adding Figure 1, we are able to show the keys that provide the exact same time as the actual execution time.

With IN = [1, 2, 3, 4], the output is provided below:

```
Time matched key: 13
```

The output shows that with KEY = 13, the execution time for all inputs is identical between actual and prediction, thus 14 is the only possible value of the key.

With IN = [1], the output is provided below:

```
Time matched key: 2
Time matched key: 4
Time matched key: 7
Time matched key: 8
Time matched key: 11
Time matched key: 13
```

The output shows that with KEY = [2, 4, 7, 8, 11, 13], the execution time for all inputs is identical between actual and prediction, thus they are all possible values of the key.

With less inputs, there are more possible choices of values for the key.

- Will you be able to guess the correct key if you use the input vector in step 2? Explain your answer.

  Yes, but with more uncertainty. [2, 4, 7, 8, 11, 13] are all possible values of the key. There is 1/6 chance of random guessing to obtain the actual key.

Task 2

- Vary the value of N in the code, then estimate the minimum value of N, needed to get an accurate estimation of the execution time.

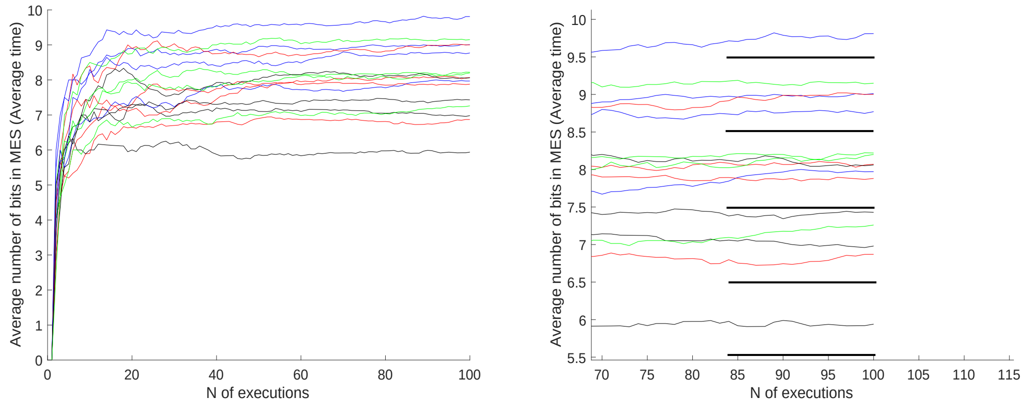

Figure 2: Average execution time for key='6CEF' for different inputs. Right image shows a zoomed in version of left image.

The estimated minimum value of N is 100. Figure 2 (right) shows that a minimum N value of 100 will allow the uncertainty for each input to not pass 0.5 ms, thus being able to retrieve the predicted time when rounded to the nearest integer. The black line shows the 0.5 ms uncertainties for each value.

- Explain the significance of the minimum value of N, in the context of a practical side channel attack.

  Suppose we have a more complicated algorithm (e.g., AES, Neural Networks), computation for one execution will take much longer. With a small minimum value of N, attackers will require less time for N executions of the algorithm to acquire side channel attack results. With a large minimum value of N, attackers will require much more time for N executions of the algorithm to acquire side channel attack results. Theoretically, a very large minimum value of N will possibly take up a lifetime (more than 100 years) to perform successful side channel attack. Thus, we can conclude that the system is practically secure from side channel attacks if the minimum value of N is large.

Task 3

- This example uses Pearson Correlation Coefficient, what other statistical metrics can be used in this case. Justify your answer with experimental evidence.

```
% Task 3.2
% Mutual Information function link
% https://github.com/Craigacp/MIToolbox
mi_vals = zeros(1, 17);
for iCtrIN = 1:17
```

```
    mi_vals(iCtrIN) = mi(timeModel(:, 1), timeModel(:, iCtrIN));
end
[mi_val, idx] = max(mi_vals(1, 2:17));

guessedKeyNibble = idx-1
```

Figure 3

Any statistical metrics that measures dependency between two variables/vectors can be used in this case. In my case, I used Mutual Information. Figure 3 shows the MATLAB code I used to calculate the mutual information for each column and return the index with the max value.

Other statistical metrics that can be used in this case are the 2-sample Kolmogorov-Smirnov (KS) test, permutation test.

Section B

Question B-2

- Draw a Data Flow Diagram, using the suggestions and guidance from the lecture notes, that shows the system on a level that is appropriate for architectural threat analysis

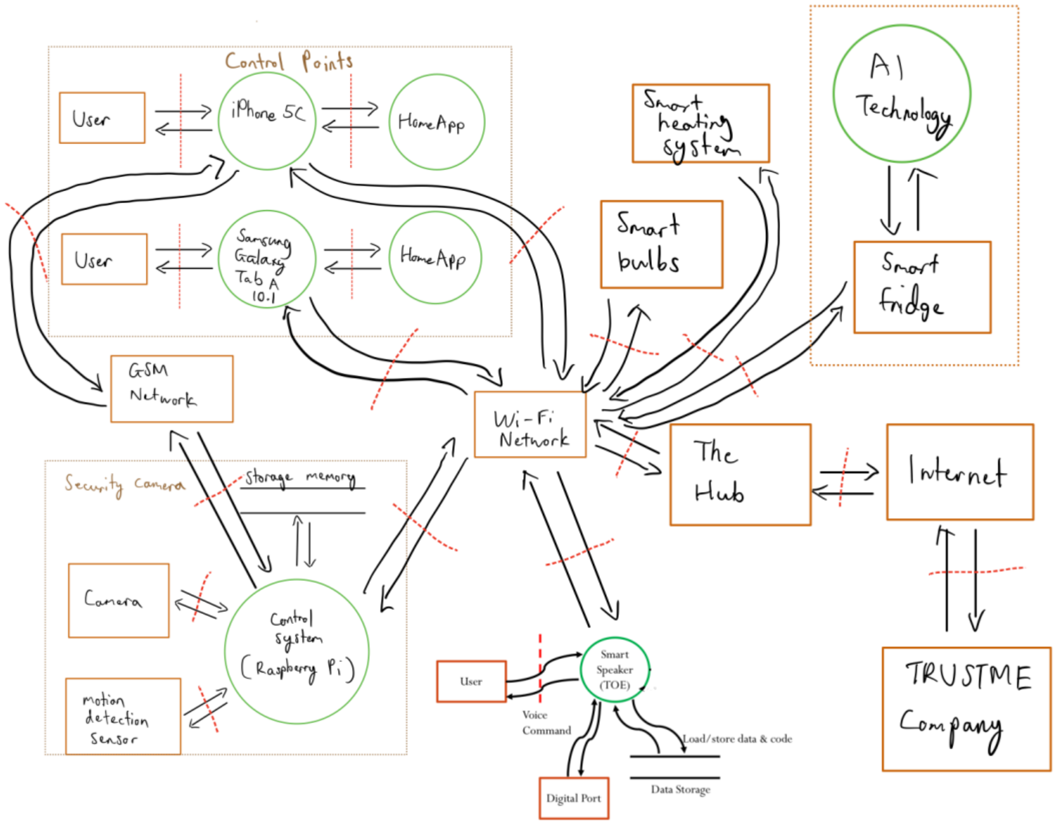

Figure 4 Data Flow Diagram

- Identify ten assets in the system, and justify your choice in each case

ID	Asset	Description

1	iPhone 5c, Samsung Galaxy Tab A 10.1	User interacts with the phone to access HomeApp
2	HomeApp	Smart home application that can control most of the smart devices
3	Camera	Able to stream video and capture images for the security camera
4	Motion detection sensor	Able to detect movements
5	Control system (Raspberry Pi)	Controls security camera mode, sends data to the mobile phone or the hub through WiFi or GSM.
6	AI technology (smart fridge)	Identifies all the types of food stored in the smart fridge and their corresponding quantities
7	Smart heating system	Can be controlled by HomeApp
8	Security camera sensor, video and image data	May contain private data of the environment or user.
9	Text message data (security camera to mobile phone)	May contain private data such as address or user location.
10	TRUSTME captured data	Captured data includes users' home addresses etc.

- For each asset you have chosen, identify one security threats based on the STRIDE model, explaining how such threat might materialize (i.e. give example of a relevant security attack technique)

ID	***Threat***	***Attack***
1	*Information Disclosure*	*Attacker getting hold of passwords to access the devices. Attacker might peak at the user when he/she is typing the password.*
2	*Tampering*	*If given access to the device, attacker can tamper with the source code to get private data, such as directing the post requests to attacker's server.*

3	*Information Disclosure*	*Attacker can track user's actions when security camera is on live stream.*
4	*Denial of Service*	*Disable motion detector so intrusion mode will not activate.*
5	*Elevation of Privilege*	*Launch DDoS attack.*
6	*Denial of Service*	*Cause the device to catch fire – thermal control.*
7	*Denial of Service*	*Attacker can switch of the heating system and cause the user to freeze.*
8	*Information Disclosure*	*Data can be illegally sold to third parties (FBI?) which can expose your private information.*
9	*Repudiation*	*Can be intercepted before the text reaches the user. Does not allow text to be sent to the user or send false texts.*
10	*Information Disclosure*	*Attacker can leak data to the public, exposing your address.*

- For each identified security threat, provide a countermeasure

ID	***Countermeasures***
1	*Be more socially aware, check your surroundings before typing your passwords.*
2	*Set up encryption and hashing of the data. Do not expose source files of HomeApp.*
3	*Encrypt data. Delete old and unwanted data.*
4	*Enforce access to motion detection sensor.*
5	*Enforce principle of least privilege.*
6	*Compile technology to binary format so that the technology can't me tempered.*
7	*Enforce the Hub authentication so that attacker cannot control the smart device.*
8	*Encrypt and hash data.*
9	*Use trusted and third-party mobile service provider and they will take care*

	of security to prevent attackers from intercepting the data.
10	*Prevent TRUSTME company from capturing data from your products.*

Section C

Question C-2

- Explain how you designed your circuit in order to meet the specifications, with appropriate supportive evidence (e.g., simulations)

  My Github code: https://github.com/eugenetwc/miniAES-Cpp
  I compiled with clang using c++17 standard. Should be able to work with gcc.

```
cd {miniAES-Cpp directory}
clang++ -std=c++17 -stdlib=libc++ -g ./*.cpp -o ./test
```

Multiplication in GF(2^4)

```
uint8_t gmul(uint8_t a, uint8_t b) {
    /*
    GF(2^4) multiplication
    using modified version of russian peasant algorithm
    it also uses branchless programming to prevent timing attacks

    reference: https://en.wikipedia.org/wiki/Finite_field_arithmetic
    */
    uint8_t p = 0;
    for (auto i = 0; i < 4; i++) {
        p ^= -(b & 1) & a;
        auto mask = -((a >> 3) & 1);
        // 0b10011 is x^4 + x + 1
        a = (a << 1) ^ (0b10011 & mask);
        b >>= 1;
    }

    return p;
}
```

For the GF multiplication, I used modified version of the Russian peasant algorithm. The Russian peasant algorithm performs multiplication in a bitwise manner, making it more optimised and efficient.

Pseudocode of algorithm:

Inputs: a, b = multiplicands containing binary notation of the polynomial(e.g. x^3 + x + 1 = 0b1011).

```
p = 0
for i in range(4 bits):
    1. If the rightmost bit of b == 1, p XOR a (polynomial addition)
    2. b >> 1 (divides the polynomial by x), discarding the x^0 term.
    3. carry = 1 if leftmost bit of a == 1, else 0
    4. a << 1 (multiplies the polynomial by x, but we still need to take account of
carry which represented the coefficient of x^3)
```

5. If carry == 1, XOR a with 0b0011. 0b0011 corresponds to the irreducible polynomial (x^4 + x + 1) with the high term eliminated.

Result: p

My implementation of the modified Russian peasant algorithm above uses branchless programming in order to prevent cache, timing and branch prediction side-channel attacks.

Addition in GF(2^4)

```
uint8_t gadd(uint8_t a, uint8_t b) {
    // GF(2^4) addition
    return (a ^ b);
}
```

GF(2^4) is just bitwise XOR.

Data Structures

```
std::vector< uint16_t > pt = {0b1001110001100011};
uint16_t key = 0b1100001111110000; // secret key

MiniAES test(key);
auto test_encrypt = test.encrypt(pt);
auto test_decrypt = test.decrypt(test_encrypt);
```

The code block above shows an example of how my implementation takes in data to encrypt and decrypt. The data structure used to be inputted into the encrypt function is std::vector <uint16_t>. This allows the user to input any number of 16 bit blocks for encryption. The data structure is the same for decryption.

Data types such as uint8_t and uint16_t allows the use of exact 1 or 2 bytes, thus it is more space optimised than normal data types such as int. All of my functions use bitwise operations.

```
// 2 x 2 uint8_t array
typedef std::array< std::array<uint8_t, 2>, 2 > matrix;

matrix MiniAES::uint16ToMatrix(uint16_t uint) {
    /*
    converts 16 bit uint16_t to 2x2 byte array using bit manipulation
    below is the index mapping the one on the paper
    [0][0]: 0, [1][0]: 1, [0][1]: 2, [1][1]: 3
    */
    matrix res;
    res[0][0] = uint >> 12;
    res[1][0] = (uint >> 8) & 15;
    res[0][1] = (uint >> 4) & 15;
    res[1][1] = uint & 15;

    return res;
}

uint16_t MiniAES::matrixToUInt16(matrix block) {
    uint16_t uint;
    for (auto j = 0; j < 2; j++) {
        for (auto i = 0; i < 2; i++) {
            uint <<= 4;
            uint |= block[i][j];
        }
    }
    return uint;
}
```

After std::vector <uint16_t> is inputted into the function, there will be a for loop to loop over each uint16_t in the vector (dynamic array). Each uint16 is then converted into a "matrix" to represent a 2x2 array. Once again, the conversion uses bitwise operators to extract bits 0-3, 4-7, 8-11, 12-15 for each index. After encryption is completed, the "matrix" will be converted back to uint16 using bitwise operators.

All of the functions below will use the "matrix".

NibbleSub

```
matrix MiniAES::nibbleSub (matrix block, bool inverse = false) {
    auto lookup = inverse ? inverse_s_box : s_box;
    for (auto i = 0; i < block.max_size(); i++) {
        for (auto j = 0; j < block[i].max_size(); j++)
            block[i][j] = lookup[block[i][j]];
    }

    return block;
}
```

NibbleSub uses S-box or inverse S-box (implemented using hash maps for constant time lookup) for lookup for each element in the matrix.

```
std::unordered_map<uint8_t, uint8_t> MiniAES::s_box = {
    {0b0000, 0b1110}, {0b1000, 0b0011},
    {0b0001, 0b0100}, {0b1001, 0b1010},
    {0b0010, 0b1101}, {0b1010, 0b0110},
    {0b0011, 0b0001}, {0b1011, 0b1100},
    {0b0100, 0b0010}, {0b1100, 0b0101},
    {0b0101, 0b1111}, {0b1101, 0b1001},
    {0b0110, 0b1011}, {0b1110, 0b0000},
    {0b0111, 0b1000}, {0b1111, 0b0111}
}

template<typename K, typename V>
std::unordered_map<V,K> inverse_map(std::unordered_map<K,V> &map)
{
    // https://www.techiedelight.com/reverse-lookup-stl-map-cpp/
    std::unordered_map<V,K> inverse;
    for (const auto &p: map) {
        inverse.insert(std::make_pair(p.second, p.first));
    }
    return inverse;
}

std::unordered_map<uint8_t, uint8_t> MiniAES::inverse_s_box = inverse_map(s_box);
```

The code above shows the implementation of s_box and inverse_s_box. inverse_s_box uses a function template (https://www.techiedelight.com/reverse-lookup-stl-map-cpp/) to inverse map the value to the key of s_box.

ShiftRow

```cpp
matrix MiniAES::shiftRow(matrix block) {
    uint8_t temp = block[1][1];
    block[1][1] = block[1][0];
    block[1][0] = temp;
    return block;
}
```

ShiftRow basically switches the positions of the lower two nibbles in the matrix.

MixColumn

```cpp
matrix MiniAES::mixColumn(matrix block) {
    matrix res;
    res[0][0] = gadd(gmul(3, block[0][0]), gmul(2, block[1][0])); // d0
    res[1][0] = gadd(gmul(2, block[0][0]), gmul(3, block[1][0])); // d1
    res[0][1] = gadd(gmul(3, block[0][1]), gmul(2, block[1][1])); // d2
    res[1][1] = gadd(gmul(2, block[0][1]), gmul(3, block[1][1])); // d3

    return res;
}
```

MixColumn performs matrix multiplication using previously defined gadd and gmul.

These are the mathematical operations the function performs:

$$d_0 = (0011 \otimes c_0) + (0010 \otimes c_1)$$

$$d_1 = (0010 \otimes c_0) + (0011 \otimes c_1)$$

$$d_2 = (0011 \otimes c_2) + (0010 \otimes c_3)$$

$$d_3 = (0010 \otimes c_2) + (0011 \otimes c_3)$$

KeyAddition

```cpp
matrix MiniAES::keyAddition(matrix block, matrix curr_rkey) {
    for (auto i = 0; i < 2; i++)
        for (auto j = 0; j < 2; j++)
            block[i][j] = gadd(block[i][j], curr_rkey[i][j]);

    return block;
}
```

KeyAddition performs gadd with the current round key for each of the element in the matrix.

The mini-AES Key-schedule

```cpp
MiniAES::MiniAES(uint16_t k) {
    setKey(k);
}

void MiniAES::setKey(uint16_t k) {
    key = k;

    // round key 0
    rkey[0][0][0] = key >> 12;
    rkey[0][1][0] = (key >> 8) & 15;
    rkey[0][0][1] = (key >> 4) & 15;
    rkey[0][1][1] = key & 15;

    // round key 1 and 2
    uint8_t rcon[2] = {0b0001, 0b0010};
    for (auto round = 1; round < 3; round++) {
        rkey[round][0][0] = gadd( gadd(rkey[round-1][0][0], s_box[rkey[round-1][1][1]]), rcon[round-1] );
        rkey[round][1][0] = gadd( rkey[round-1][1][0], rkey[round][0][0] );
        rkey[round][0][1] = gadd( rkey[round-1][0][1], rkey[round][1][0] );
        rkey[round][1][1] = gadd( rkey[round-1][1][1], rkey[round][0][1] );
    }
}
```

The round keys are initialised as a private instance variable of the MiniAES class (std::array< matrix, 3> rkey). It is then initialised in setKey or the constructor. The implementation is straightforward based on the paper.

Mini-AES Encryption

```cpp
std::vector< uint16_t > MiniAES::encrypt(std::vector< uint16_t > pt) {
    // encrypt from bitset vector to bitset vector
    std::vector < uint16_t > ct;

    for (auto i = 0; i < pt.size(); i++) {
        auto block = uint16ToMatrix(pt[i]);

        // round 0
        block = keyAddition(block, rkey[0]);
        // round 1
        block = nibbleSub(block);
        block = shiftRow(block);
        block = mixColumn(block);
        block = keyAddition(block, rkey[1]);
        // round 2
        block = nibbleSub(block);
        block = shiftRow(block);
        block = keyAddition(block, rkey[2]);

        // add to ct
        ct.push_back(matrixToUInt16(block));
    }

    return ct;
}
```

Again, the encryption is straightforward based on the paper. First, it loops over each of the vector elements, performs encryption, and appends to the ciphertext vector. The encryption implements the block cipher.

Mini-AES Decryption

```cpp
std::vector< uint16_t > MiniAES::decrypt(std::vector< uint16_t > ct) {
    // decrypt from bitset vector to bitset vector
    std::vector < uint16_t > pt;

    for (auto i = 0; i < ct.size(); i++) {
        auto block = uint16ToMatrix(ct[i]);

        block = keyAddition(block, rkey[2]);
        block = shiftRow(block);
        block = nibbleSub(block, true);

        block = keyAddition(block, rkey[1]);
        block = mixColumn(block);
        block = shiftRow(block);
        block = nibbleSub(block, true);

        block = keyAddition(block, rkey[0]);

        pt.push_back(matrixToUInt16(block));
    }

    return pt;
}
```

The decryption is the same as encryption but does everything backwards.

Testing

I inputted three inputs as a test vector into the instance to test the results.

```cpp
int main() {
    std::vector< uint16_t > pt = {0b1001110001100011, 0b1111, 0xffff};
    uint16_t key = 0b1100001111110000; // secret key

    MiniAES test(key);
    auto test_encrypt = test.encrypt(pt);
    auto test_decrypt = test.decrypt(test_encrypt);

    for (auto i = 0; i < test_encrypt.size(); i++) {
        std::cout << "pt" << i << ": " << pt[i] << std::endl;
    }
    std::cout << "key: " << key << std::endl;
    for (auto i = 0; i < test_encrypt.size(); i++) {
        std::cout << "test_encrypt" << i << ": " << test_encrypt[i] << std::endl;
        std::cout << "test_decrypt" << i << ": " << test_decrypt[i] << std::endl;
    }

}
```

Output:

```
pt0: 40035
pt1: 15
pt2: 65535
key: 50160
```

```
test_encrypt0: 29382
test_decrypt0: 40035
test_encrypt1: 48055
test_decrypt1: 15
test_encrypt2: 25018
test_decrypt2: 65535
```

The output is given as the decimal format of the binary/hex in the code. I compared it against my handwritten calculations and it gave me the same results.

We can see that after the decryption, we can get the plaintext back.

With the testing above, I can conclude that MiniAES is successfully implemented.

- Is your design vulnerable to timing analysis attacks? Explain your answer

  My design is vulnerable to timing/cache analysis attacks because the the CPU may cache the S-box hash map with the same address every single execution. This will result in timing leaks based on the time it takes to access the memory address.

  However, I try to mitigate the impact of timing attacks by implementing branchless programming in the gmul function.

  Modern CPUs often attempt to make conditional (if else etc.) jumps based on jump predictions beforehand. Hence it might perform the same conditional jumps for specific inputs, making it prone to timing attacks. By completely removing conditional branches (if else) from gmul, we will prevent conditional jumps from happening, thus preventing timing attacks.

  Reference: https://www.bearssl.org/constanttime.html