The Full Title of your Thesis

Your Name

Bachelor Thesis – 0, 2019.
Cyber Security
Department of Information Systems
University Münster, Germany

Principal Supervisor: Prof. Dr.-Ing. Thomas Hupperich

Abstract

The usage of smartphones has become quite popular in the last decade. Every new smartphone model contains new hardware features, thus making people more attracted to benefit from these features. But the more features a smartphone contains, the more potential privacy and security issues arise from these features. Many of these features utilize the various built-in smartphone sensors like the accelerometer. Accelerometer data lead to the ability for installed applications to track the user's actual condition and activity. Previous laboratory studies have proved that hardware imperfections during the accelerometer manufacturing process, provide the possibility to recognize smartphones by utilizing signal feature extraction. In our approach, we demonstrate whether this method is applicable to recognize device models just as well as unique devices.

Contents

1 Introduction

In this chapter, we give an overview on our motivation and the related work done so far on fingerprinting and recognizing accelerometers, followed by our contribution to this field of studies.

1.1 Motivation

The ability to recognize devices such as tablets and desktops has been discussed quite often in the last decade. Methods such as supercookies and tracking static IP addresses, and tools like panopticlick [?] have been utilized to recognize this kind of devices. With the growing popularity of mobile devices, advertising companies have also developed an interest in this category of devices. The challenge of recognizing mobile devices remains in the great similarity between the hardware and software specification of device models. Additionally, traditional tracking options have been restricted to prevent mobile devices being fingerprinted. For instance, in case of tracking cookies, the cookie law [?] which regulates storing and retrieving information on a computer, smartphone, or tablet of visitors to a website in EU countries was passed on May 26 2011. User-defined settings in case of privacy-concerning features, e.g., GPS sensors prevent applications from tracking the current device location. Another example is Apple's improved policy since the presentation of iOS 7. Apple's iOS application developers are prohibited from using unique device features such as the unique device identifier (UDID), international mobile station equipment identity (IMEI), and media access control (MAC) address of the Wi-Fi interface as recognition mechanism. These effective preventive measures have caused adversaries to look for fingerprinting methods that do not require any sort of permission. Getting deeper into the modern features of mobile devices, it turns out that the accelerometer does not need any of the mentioned permissions, proved by various gaming and fitness applications that have never required permission to access the accelerometer readings so far. Gaming applications need accelerometer readings to track device movements and perform predefined actions, according to recognized patterns from the accelerometer readings. Fitness tracking applications use accelerometer readings to distinguish between user's physical activities and extract the resulting heart rate [?]. Previous studies have illustrated that calibration imprecisions and hardware imperfections during the manufacturing process lead to unique hardware-tolerances. We refer to these studies in Section 1.2. By combining these unique imprecisions and lack of necessity of obtaining permission to read accelerometer readings, an adversary can calculate unique accelerometer fingerprints of a mobile device.

1.2 Related Work

The power of extracting time and frequency domain features in accelerometers has been shown by W. Dargie and M. Denko [?]. Their study includes random placement of accelerometers on moving humans and cars, and investigating the behavior of accelerometers during similar movements. They conclude that the extracted frequency domain features remain generally more robust than time-domain frequency features. In our study, we applied accelerometer readings that were gathered from both resting and moving devices. The features that were analyzed in this paper differ from the features that we used. Yet, several time domain features such as mean, standard deviation, and highest and lowest value, as well as frequency domain features such as spectral centroid are common in both works.
S. Dey et al. [?] created fingerprints for mobile devices by extracting time and frequency domain features from the accelerometer readings of vibrating sensors and mobile devices. Their study illustrates the existence of such unique fingerprints by conducting a series of training and test set scenarios on 107 different stand-alone chips, smartphones, and tablets under laboratory conditions. This paper is the main basis of this thesis. In this thesis, the duration of the time window of the accelerometer readings amounts 10 seconds while the AccelPrint paper used accelerometer readings with a duration of 2 seconds. A shorter time window leads to a more stationary signal which also provides more accurate feature extraction results and also more fingerprints for the training and test phase. Other than the applied bagging tree classifier in AccelPrint, we also tested random forests, extra-trees, and gradient boosting to have a better overall overview on device and model recognition. Additionally, we could compare the behavior of different ensemble methods.
Bojinov et al. presented accelerometer-based fingerprinting by applying JavaScript code to read out the accelerometer readings [?]. They attempted to recognize smartphones by calculating two bias parameters from the z axis while the device was facing up or down on a resting surface. Their best and worst recognition rates were 8.3 and 15.1%.
Application of the machine learning library *scikit-learn* on recognizing human activities has been shown by H. He [?]. This study used the accelerometer and gyroscope readings of 6 different human activities. The number of features that were extracted in this study were 561, which is much more than our 17 extracted features. In a PCA (principal component analysis) dimension reduction, He reduced the number of features to train the applied classifiers to 50 and 20. In the later number of features which is closer to the number of features used in our study, the achieved results in the random forests classifier are similar to our result (cf. Chapter **??**). The other

applied classification methods were different from our methods. We preferred to apply ensemble methods as classification methods and also test clustering methods for device and model recognition purposes.

1.3 Contribution

Our contribution can be summarized as follows:

1. We implemented the signal feature extraction process utilizing the *NumPy* and *SciPy* libraries and, where necessary, developed our own implementation of the formulas. This process was especially essential regarding the spectral feature extraction.
2. We show that besides recognition of unique mobile devices, it is also possible to recognize the device model. For this purpose, we conducted two similar tests where the same minimum number of fingerprints was applied to have a precise comparison between the results.
3. The comparison of 4 ensemble classification methods of the *scikit-learn* library to recognize unique mobile devices and device models is also part of this thesis. We compare the results of the classification methods and discuss their efficiency on data sets that are either free of noise or contain fingerprints created from faulty accelerometer readings.
4. To the best of our knowledge, this is the first work where clustering is attempted to recognize mobile devices or the model of a device through their accelerometer fingerprints. We conducted tests with two different clustering methods of *scikit-learn* to find out if clustering is also applicable to group accelerometer fingerprints correctly.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter **??** discusses the accelerometer briefly and introduces the time and frequency domain features and libraries that were used in this thesis. In Chapter 4, we provide the implementation details and explain the tested machine learning methods. Chapter **??** covers the data analysis of the data set and the results of applying machine learning on the accelerometer fingerprints. Chapter 6 presents the conclusion and possible future work.

2 Documentation - Encryption

2.1 Overview

The following chapter is based on (fips197) and the authors own knowledge/opinions, if not mentioned otherwise.

2.2 On the usage of optional C keywords

2.2.1 inline

2.2.2 const

2.2.3 restrict

2.2.4 static

2.3 Functions - constant generators

2.3.1 Galois field multiplication lookup table generator

Description

Implementation

Testing

2.3.2 Sbox generator

Description

Implementation

Testing

2.4 Functions - encryption

2.4.1 Key Addition

Description

$$\begin{array}{|c|c|c|c|}\hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ \hline k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ \hline k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ \hline k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array}$$

In this transformation a round key is combined with the state, thus modifying it. The combintation is accomplished using the bitwise XOR operation. The round key array is derived from the initial cipher key using the key schedule. Containing 16 bytes, each round key is equally long to a block. Since there are N+1 round keys generated (where N is the number of rounds), each round uses a different round key.

Implementation

```
/*
 * Adds the roundkey to a block.
 */
inline void AddRoundKey(uint8_t * restrict bytes,
                        const uint8_t * restrict keys)
{
        for(uint8_t i = 0; i < 16; i++) {
                bytes[i] ^= keys[i];
        }
}
```

This function is used to add the current round key to the current block. It takes a restricted pointer to the first byte of the block the cipher is currently operating on and a restricted pointer to a constant, which points to the first byte of the round key, that is supposed to be used currently. It proceeds to combine each of the sixteen bytes of the block with a corresponding byte of the round key designated to the current round using the bitwise XOR. The result is then stored in the block byte used in the XOR operation, meaning that first byte one of the block will be XORed with byte one of the round key and then the result will be stored in place of byte one and so forth for all sixteen bytes.

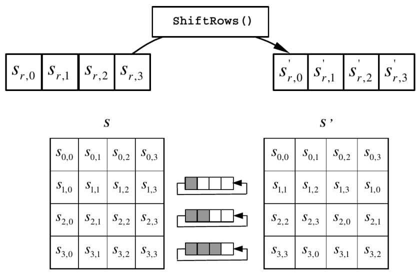
Since the function is relativly short the 'inline'-keyword is used to save the overhead of an additional function call in tradeoff with a bigger binary.

Testing

Since the function is fairly simple, it is not covered by the testing suite.

2.4.2 Shift rows

Description



(rijndael)(p.37) states, that this transformation of the state represents a byte transposition, using cyclical shifts with different offsets. The first row of the 4x4-Matrix of 16 bytes that constitutes the so called state is not shifted at all, the second row by one step to the left, the third row uses two steps and the fourth row three.

According to (rijndael) this transformation step is needed to ensure optimal diffusion of the state. The diffusion is supposed to protect against differential and linear crypt-analysis. The authors further elaborate the need for that in order to archive optimal diffusion all offsets of the cyclical shifts have to be different.

Since there are multiple possibilities for different offsets and not all of them provided equal protection studies of attacks against Rijndael were analyzed. From the offsets that proved to be the most resistant the simplest offset was chosen.

Implementation

```
/*
 * Achieves the AES-ShiftRows by representing it as a series of array-assignments.
 */
void ShiftRows(uint8_t * restrict block, uint8_t * restrict tempblock)
{
	memcpy(tempblock, block, 16 * sizeof(uint8_t));

	block[1] = tempblock[5];
	block[2] = tempblock[10];
	block[3] = tempblock[15];
	block[5] = tempblock[9];
	block[6] = tempblock[14];
	block[7] = tempblock[3];
	block[9] = tempblock[13];
	block[10] = tempblock[2];
	block[11] = tempblock[7];
	block[13] = tempblock[1];
	block[14] = tempblock[6];
	block[15] = tempblock[11];
}
```

The function ShiftRows performs the ShiftRows-transformation on the state. It takes a restricted pointer to the first byte of the block the cipher is currently operating on and a restricted pointer to a temporary block. First the content of the block is copied into the array the tempblock-pointer is marking. After that the row shifts are represented by assigning the bytes into their positions in the block after the shifts from the tempblock, for example:

> Since the eleventh byte of the original array (tempblock[10]) will (after the rotations) always be in the third spot in the resulting array (block[2]) we can simply assign the former to the latter. The first byte (tempblock[0]) is mapped to itself and therefore is not explicitly "mentioned" in the code, since an assignment to itself would be redundant.

The result, which is the ShiftRows-transformed state, can now be found in the array marked with the block-pointer and used for further transformations.

'temparray' is passed to the function as a pointer, because allocating the array every time the function is called creates a conciderable overhead.

Testing

```
@pytest.mark.parametrize(
    ("input_block", "expected"),
    (
        ("63cab7040953d051cd60e0e7ba70e18c", "6353e08c0960e104cd70b751bacad0e7"),
        ("a761ca9b97be8b45d8ad1a611fc97369", "a7be1a6997ad739bd8c9ca451f618b61"),
        ("3b59cb73fcd90ee05774222dc067fb68", "3bd92268fc74fb735767cbe0c0590e2d")
    )
)
def test_ShiftRows(input_block, expected):
    """Tests the C-implementation of the ShiftRows-function"""
    ba = bytearray.fromhex(input_block)
    reference = bytearray.fromhex(expected)
    byte_array = ctypes.c_ubyte * len(ba)
    temp_array = ctypes.c_ubyte * len(ba)
    aeslib.ShiftRows(byte_array.from_buffer(ba), temp_array.from_buffer_copy(ba))
    assert ba == reference
```

The testfunction for ShiftRows possesses a decorator, which tells pytest what parameters this function is supposed to be tested on. The function itself accepts two variables, one for the input of the ShiftRows-function labeled 'input_block' and a corresponding correct output labeled 'expected', which will be computed from the input, if the ShiftRows-function behaves correctly. Values for both input variables are expected to be strings of 16-byte hex numbers. Those numbers are transformed into two seperate byte arrays, 'ba' and 'reference'. The C-ShiftRows function accepts a pointer to an array of 16 unsigned bytes, which contain the current state and a pointer to an array of 16 unsigned bytes, which can be used as a temporary array. After the ShiftRows function returns, the result of the transformation is contained in the bytearray 'ba'. 'ba' is then compared to 'reference'. pytest will mark this test as passed, when all three comparisons of the transformed ‘ba’ and ’reference evaluated to true. The test vectors passed via the decorator are from (fips197) and represent the expected transformations.

2.4.3 Mix columns

Description

The MixColumns-transformation is called a bricklayer permutation by (rijndael), which operates on the state-matrix columnwise. The authors mention multiple design criteria, they deemed important for this particular transformation:

1. *The bricklayer transformation is supposed to operate on columns containing 4 bytes.* This aspect is supposed to aid with the optimal implementation of look-up tables on 32-bit architectures, thus ensuring a speedy computation of the transformation.
2. *The operation should be linear over GF(2)*, meaning the Galois Field of the two elements 0 and 1. This property aides the authors in their so-called ‘Wide Trail Strategy’ (p.126) which is supposed to protect against differential and linear cryptanalysis.
3. *The diffusion achieved by this transformation is supposed to have “relevant” power.* The third property is again supposed to support the ‘Wide Tail Strategy’.
4. *The authos put emphasis on the performance of this step on 8-bit CPUs.* They deemed this neccessary, as they feared that the MixColumns transformation would be “the only step that good performance on 8-bit processors is not trivial to obtain for.”(p.39)

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \mathsf{02\ 03\ 01\ 01} \\ \mathsf{01\ 02\ 03\ 01} \\ \mathsf{01\ 01\ 02\ 03} \\ \mathsf{03\ 01\ 01\ 02} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The transformation itself partitions the state-matrix into four columns. Each of those columns is transformed independently from the other three. A single column acts as a polynomial over GF(2), which is multiplied modulo x+1 with a fixed polynomial. In order to meet the aforementioned criteria regarding performance, diffusion and to fullfill the additional condition of creating an invertible transformation (in regards to encryption) the authors were forced to impose conditions on the fixed polynomials (rijndael). For example, performance was archived by using only simple values for the coefficients of the fixed polynomial. The fixed polynomial c(x) Daemen and Rijmen settled on is c(x) = 03 * x^3 + 01 * x^2 + 01 * x + 02 They state, that since c(x) and the aforementioned modulo x+1 are coprime the calculation is invertible. The matrix-multiplication, which is seen in Fig XXXX, is a representation of the modular multiplication with a fixed polynomial.

158 - 88 4.1.1. ### Implementation

```
/*
 * Achieves the AES-MixColumns by representing each result byte as a series of
 * table-lookups XORed with each other.
 */
void MixColumns(uint8_t * restrict block, uint8_t * restrict tempblock,
                const uint8_t (* restrict gal_mult_lookup)[256])
{
        memcpy(tempblock, block, 16 * sizeof(uint8_t));

        for(uint8_t i = 0; i < 16; i += 4) {

                block[i] = gal_mult_lookup[1][tempblock[i]] ^
                        gal_mult_lookup[2][tempblock[i+1]] ^
                        gal_mult_lookup[0][tempblock[i+2]] ^
                        gal_mult_lookup[0][tempblock[i+3]];

                block[i+1] = gal_mult_lookup[0][tempblock[i]] ^
                        gal_mult_lookup[1][tempblock[i+1]] ^
                        gal_mult_lookup[2][tempblock[i+2]] ^
                        gal_mult_lookup[0][tempblock[i+3]];

                block[i+2] = gal_mult_lookup[0][tempblock[i]] ^
                        gal_mult_lookup[0][tempblock[i+1]] ^
                        gal_mult_lookup[1][tempblock[i+2]] ^
                        gal_mult_lookup[2][tempblock[i+3]];

                block[i+3] = gal_mult_lookup[2][tempblock[i]] ^
                        gal_mult_lookup[0][tempblock[i+1]] ^
                        gal_mult_lookup[0][tempblock[i+2]] ^
                        gal_mult_lookup[1][tempblock[i+3]];
        }
}
```

The function MixColumns takes multiple arguments. First it takes a restricted pointer to the first byte of the block the cipher is currently operating on, followed by a restricted pointer to a temporary block and lastly the function gets a restricted pointer to the constant, multi-dimensional GFMLT. MixColumns starts by copying the current state from ‘block’ into ‘tempblock’. After that it iterates through every row by processing 4 consecutive bytes at a time, before it moves to the following for consecutive bytes of ‘block’. Each Byte in ‘block’ becomes the result of four Galois field multiplications combined through the bitwise XOR-operation, which represents the Galois field addition. The multiplications are implemented as lookups in the GFMLT and combine the coefficients of the fixed polynomial c(x) with the bytes of the current row. After processing all four rows and returning, MixColumns has written the results of this transformation in place of the current block being processed.

As (rijndael) recommends in chapter 4.1.1 we implemented the Galois field multiplications as lookup tables in GFMLT. Since the number of possible multiplication results is highly restricted in GF(2) the table storing all those results uses neglibe memory. This avoids computing the multiplications over and over. Furthermore in chapter 10.8.1 the authors suggest, that implementing the multiplications as table lookups makes the algorithm more resistant to timing attacks, since the finite field multiplication performed in MixColumns is the only operation that is not computed in constant time. In their 8-bit implementation in chapter 4.1.1 they suggest only generating a table where table[a] = 02 * a, since multiplication with one is the factor itself and multiplication with three can be efficiently archived by XORing the result of the multiplication with two with the original number: 03 * a = (02 * a) ˆ a To avoid leaking more timing data than absolutely neccessary the present implementation uses a lookup table for all values, since this ensures that for each multiplication the CPU (theoretically) does always the exact same amount of work, thus ensuring it always takes the exact same amount of time. This size increase by 2 * 256b = 512 bytes (compared to the suggested implementation with only one row) is negligible for our target of x86-CPU-system, which today typically feature memory sizes in the gigabyte range. (rijndael) did not make a mistake in their implementation suggestion though, since the small lookup table variant was only suggested for memory constrained 8-bit enviroments. For 32 bits or wider they make even more extensive use of lookup tables than the present implementation does (compare ch. 4.2). This variant has not been used, since it reduces the steps of the algorithm for each column in each round into four table lookups accessing large tables crafted for this purpose, and four XOR operations concatenating the lookup-results. This obscures the original layout heavily and we wished for it to be recognizable in our implementation.

In the end the suggestions (rijndael) makes regarding this topic have to be taken with a grain of salt, since the extensive use of lookup tables seems to enable new side channel attack vectors as previously discussed.

Testing

```
<# Vectors from [FIPS 197] Appendix C, Rounds 1, 2, 3
@pytest.mark.parametrize(
    ("input_block", "expected"),
    (
        ("6353e08c0960e104cd70b751bacad0e7", "5f72641557f5bc92f7be3b291db9f91a"),
        ("a7be1a6997ad739bd8c9ca451f618b61", "ff87968431d86a51645151fa773ad009"),
        ("3bd92268fc74fb735767cbe0c0590e2d", "4c9c1e66f771f0762c3f868e534df256")
    )
)
def test_MixColumns(input_block, expected):
    """Tests the C-implementation of the MixColumns-function"""
    ba = bytearray.fromhex(input_block)
    reference = bytearray.fromhex(expected)
    byte_array = ctypes.c_ubyte * len(ba)
    temp_array = ctypes.c_ubyte * len(ba)
    gal_mult_lookup_array = ctypes.c_ubyte * len(gal_mult_lookup)
    aeslib.MixColumns(byte_array.from_buffer(ba), temp_array.from_buffer_copy(ba),
                      gal_mult_lookup_array.from_buffer(gal_mult_lookup))
    assert ba == reference
```

Like other testing functions the test vectors for pytest to review are provided via decorator. They are copied from (fips197) to ensure correctness. The function takes one argument which represents the state before a MixColumns transformation and one argument, which represents the state after said transformation. Both are expected to be strings containing the bytes encoded in hexadecimal notation. ‘input_block’ gets converted to the bytearray ‘ba’ and is then passed along with another temporary sixteen-byte-array on towards the C-implementation of the MixColumns-function. After the latter function returns, ba contains the result of the desired transformation. pytest now compares each output state to the corresponding test vector value to determine if the MixColumns-implementation works correctly.

2.4.4 Substitute bytes

Description

Being the only non linear transformation, SubBytes is labeled a bricklayer permutation by (rijndael) that applies byte substitution to each byte of the state. This substitution is facilitated throught the AES Sbox, which was already discussed in more detail earlier. AES uses only this one Sbox, although the authors mention, that Rijndael “could as easily be defined with different S-boxes for every byte.” (p.35).

Implementation

```
/*
 * Substitutes all bytes in a block with bytes from a sbox.
 */
inline void SubBytes(uint8_t * restrict bytes, const uint8_t * restrict sbox)
{
	for(uint8_t i = 0; i < 16; i++) {
		bytes[i] = sbox[bytes[i]];
	}

}
```

Testing

2.4.5 Full block encryption

Description

Implementation

```
void encryptBlock(uint8_t * restrict block, uint8_t * restrict tempblock,
                const uint8_t * restrict keys, const uint8_t rounds,
                const uint8_t * restrict sbox,
                const uint8_t (* restrict gal_mult_lookup)[256])
{
       uint8_t ikeys = 0;
       AddRoundKey(block, keys);
       ikeys += 16;

       for(uint8_t i = 0; i < rounds - 1; i++) {
              SubBytes(block, sbox);
              ShiftRows(block, tempblock);
              MixColumns(block, tempblock, gal_mult_lookup);
              AddRoundKey(block, &keys[ikeys]);
              ikeys += 16;
       }

       SubBytes(block, sbox);
       ShiftRows(block, tempblock);
       AddRoundKey(block, &keys[ikeys]);
}
```

Testing

```
@pytest.mark.parametrize(
   ("plaintext", "key", "expected"),
   (
      (
         "3243f6a8885a308d313198a2e0370734",
         "2b7e151628aed2a6abf7158809cf4f3c",
         "3925841d02dc09fbdc118597196a0b32"
      ),
   )
)
def test_EncryptBlock(plaintext, key, expected):
   """Tests the C-implementation of the AES-Encryption of a single block"""
   ba = bytearray.fromhex(plaintext)
   byte_array_block = ctypes.c_ubyte * len(ba)

   reference = bytearray.fromhex(expected)
   baKey = expand_key(key)
   byte_array_key = ctypes.c_ubyte * len(baKey)

   temp_array = ctypes.c_ubyte * len(ba)
   sbox_array = ctypes.c_ubyte * len(sbox)
   gal_mult_lookup_array = ctypes.c_ubyte * len(gal_mult_lookup)


   aeslib.encryptBlock(byte_array_block.from_buffer(ba), temp_array.from_buffer_copy(ba),
                     byte_array_key.from_buffer(baKey), 10, sbox_array.from_buffer(sbox),
                     gal_mult_lookup_array.from_buffer(gal_mult_lookup))
   assert ba == reference
```

```
def test_EncryptBlockRandom():
   """
   Tests the C-implementation of the AES-Encryption of an arbitrary number of individual
       blocks.
   Plaintext and Key are randomized each time.
```

```
    """
    ba = bytearray(16)
    byte_array_block = ctypes.c_ubyte * len(ba)
    key = bytearray(16)
    baKey = bytearray(176)
    byte_array_key = ctypes.c_ubyte * len(baKey)
    temp_array = ctypes.c_ubyte * len(ba)
    sbox_array = ctypes.c_ubyte * len(sbox)
    gal_mult_lookup_array = ctypes.c_ubyte * len(gal_mult_lookup)
    for i in range(100): #arbitrary number of rounds
        key = urandom(16)
        baKey = expand_key(key.hex())
        b = urandom(16)
        ba = bytearray(b)
        aes_reference = pyaes.AESModeOfOperationECB(key)
        reference = aes_reference.encrypt(b)
        aeslib.encryptBlock(byte_array_block.from_buffer(ba), temp_array.from_buffer_copy(ba),
                        byte_array_key.from_buffer(baKey), 10, sbox_array.from_buffer(sbox),
                        gal_mult_lookup_array.from_buffer(gal_mult_lookup))
        assert ba == reference
```

2.4.6 Encryption of multiple consecutive blocks

Description

Implementation

```
/*
 * Initializes the constant tables sbox, galois-field-multiplication-lookup and keys.
 * Performs AES-encryption on multiple, consecutive blocks.
 */
void encryptAES(uint8_t * restrict bytes, uint8_t * restrict initval,
                 const size_t bytecount, const uint8_t rounds)
{
        uint8_t sbox[256];
        uint8_t gal_mult_lookup[3][256];
        for(size_t i = 0; i < 256; i++) {
                sbox[i] = *initval++;
        }
        for(size_t i = 0; i < 3; i++) {
                for(size_t j = 0; j < 256; j++) {
                gal_mult_lookup[i][j] = *initval++;
                }
        }

        const uint8_t *keys = initval;
        uint8_t tempblock[16];

        for(size_t i = 0; i < bytecount; i += 16) {
                encryptBlock(&bytes[i], tempblock, keys,
                             rounds, sbox, gal_mult_lookup);
        }
}
```

Testing

d

```
def test_encrypt_aes():
    """Tests the C-implementation of the AES-Encryption on multiple, consecutive, random
        Blocks.
    """
    no_bytes = 2**15 #arbitrary number of bytes (has to be divisible by 16)
    ba = bytearray(no_bytes)
    byte_array_block = ctypes.c_ubyte * len(ba)
    key = bytearray(16)
    baKey = bytearray(176)
    initvals = bytearray(len(sbox) + len(gal_mult_lookup) + len(baKey))
    byte_array_initvals = ctypes.c_ubyte * len(initvals)
    for i in range(20): #arbitrary number of rounds
        key = urandom(16)
        baKey = expand_key(key.hex())
        initvals = sbox + gal_mult_lookup + baKey
        b = urandom(no_bytes)
        ba = bytearray(b)
        aes_reference = pyaes.AESModeOfOperationECB(key)
        aeslib.encryptAES(byte_array_block.from_buffer(ba),
                    byte_array_initvals.from_buffer(initvals), len(ba), 10)

        i = 0
        while (i < no_bytes):
            current_reference = aes_reference.encrypt(b[i:i+16])
            current_ba = ba[i:i+16]
            assert current_ba == current_reference

            i += 16
```

- CPU central processing unit
- GFMLT Galois-Field multiplication lookup table

´

3 Approach

4 Implementation

5 Evaluation

6 Conclusion

A Acronyms

IMEI	international mobile station equipment identity
MAC	media access control
UDID	unique device identifier

List of Figures

List of Tables

List of Algorithms

List of Listings

Declaration

I hereby declare that, to the best of my knowledge and belief, this Bachelorthesis titled “The Full Title of your Thesis” is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

DATE	SIGNATURE

Consent Form

for the use of plagiarism detection software to check my thesis

Full Name: Your Name
Student Number: 000000
Course of Study: Information Systems
Title of Thesis: The Full Title of your Thesis

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Date	Signature