
Aktuelle Themen der IT-Sicherheit: RIOT Challenges

WS 22/23: Prof. Dr. Jan Seedorf

Thomas Jakkel (1001594), Severin Nonenmann (1001599),

Lukas Reinke (1001213), Lars Weiß (1001596)

29. Januar 2023



Inhalt

1	Woche 1	2
1.1	Challenge 1: VM installation	2
1.2	Challenge 2: Erste schritte mit RIOT	2
1.2.1	First_test Application	2
1.2.2	Einfache Netzwerk Kommunikation	3
2	Woche 2	5
2.1	Challenge 1	5
2.1.1	Task 01	5
2.1.2	Task 02	6
2.1.3	Task 03	6
2.1.4	Task 04	6
2.2	Challenge 2	7
2.3	Challenge 3	7
2.3.1	Review AES-CBC	7
2.3.2	Additional Block Cipher Mode: CTR	8
2.4	Challenge 4	10
2.4.1	4.2. Understanding existing benchmarking code	10
2.4.2	4.3 Enhance existing benchmark code with AES-CTR	11
2.4.3	4.4 Benchmark Tests	11
3	Woche 4	13

Der in dieser Abgabe verwendete code kann auf GitHub eingesehen werden: <https://github.com/ITS2-FIDO2/Abgabe-Seedorf>

1 Woche 1

1.1 Challenge 1: VM installation

Das VM setup wird hier nicht genauer beschrieben.

1.2 Challenge 2: Erste schritte mit RIOT

Die Anleitung zum Setup von RIOT OS aus den RIOT Tutorials wurde durchlaufen und ein funktionierender Workspace erstellt.

Wir haben das Setup insofern verändert, dass unser Code in einem separaten Ordner neben dem von GitHub geklonten RIOT Dateien liegt.

1.2.1 First_test Application

Das Ziel ist ein erstes RIOT-OS selber zu kompilieren, mit einer eigen Funktion zu versehen und zu starten.

Im default Makefile müssen zwei Änderungen vorgenommen werden:

1. In der Variable `APPLICATION` der Name der Ausführbaren binary zu setzen
2. Die `RIOTBASE`, dem Pfad zu den Hauptdateien des RIOT-OS, zu setzen.

```
1 ...  
2 APPLICATION = First_test  
3  
4 BOARD ?= native  
5  
6 RIOTBASE ?= $(CURDIR)/../../RIOT/  
7 ...
```

Es soll eine Shell Kommando geschrieben werden das bei Aufruf einen String zurück gibt. Zugrunde liegt eine einfache C Funktion mit einem `printf()` statement:

```
1 static int whats_up(int argc, char **argv) {  
2     (void)argc;  
3     (void)argv;  
4 }
```

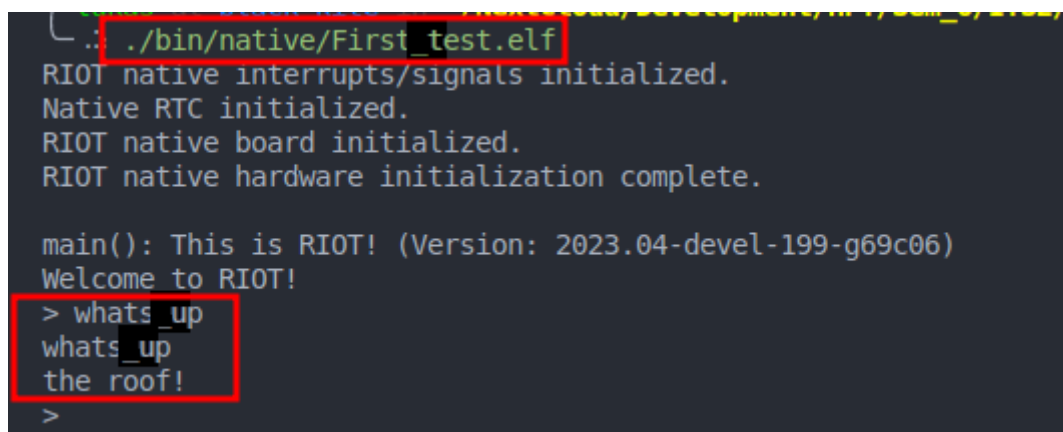
```
5     printf("The roof!\n");
6     return 0;
7 }
```

Des weiteren muss die Funktion in einem Array eingetragen und dieses Array als Quelle für Shell-befehle in der `main` Funktion registriert werden.

```
1 const shell_command_t shell_commands[] = {
2     {"whats_up", "prints the roof", whats_up},
3     { NULL, NULL, NULL}
4 };
```

```
1 shell_run(shell_commands, line_buf, SHELL_DEFAULT_BUFSIZE);
```

Nun kann mithilfe des `make`-Kommandos ein build gestartet werden und die resultierende Binary mit dem Namen **First_test.elf** ausgeführt werden. In der RIOT Shell kann nun der Befehl `whats_up` ausgeführt werden.



```
./bin/native/First_test.elf
RIOT native interrupts/signals initialized.
Native RTC initialized.
RIOT native board initialized.
RIOT native hardware initialization complete.

main(): This is RIOT! (Version: 2023.04-devel-199-g69c06)
Welcome to RIOT!
> whats_up
whats_up
the roof!
>
```

Abbildung 1: `whats_up` Befehl in RIOT shell

1.2.2 Einfache Netzwerk Kommunikation

Das Ziel dieser Challenge war, zwei RIOT-OS Instanzen über Netzwerk kommunizieren zu lassen.

Das im RIOT Repo mitgelieferte Script `tapsetup` kann genutzt werden um in der Linux Umgebung zwei interfaces (`tap0` und `tap1`) anzulegen.

Wird nun eine RIOT-OS Instanz mit dem Zusatz `PORT=tap0` ist das Interface `tap0` Verbunden und kann intern mit dem Befehl `ifconfig` gefunden werden. Neben dem Interface wird die Hardware-Adresse auf der das Interface später angesprochen werden kann angezeigt.

```
valid_lft forever preferred_lft forever
5: tapbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   link/ether 1a:0c:2f:70:77:b9 brd ff:ff:ff:ff:ff:ff
6: tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   link/ether f2:fe:f4:e3:88:d4 brd ff:ff:ff:ff:ff:ff
7: tap1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
   link/ether c2:b5:35:a8:66:fd brd ff:ff:ff:ff:ff:ff
```

Abbildung 2: Zwei virtuelle Interfaces mit verbindender “bridge”

```
./bin/native/First_test.elf PORT=tap1
RIOT native interrupts/signals initialized.
Native RTC initialized.
RIOT native board initialized.
RIOT native hardware initialization complete.

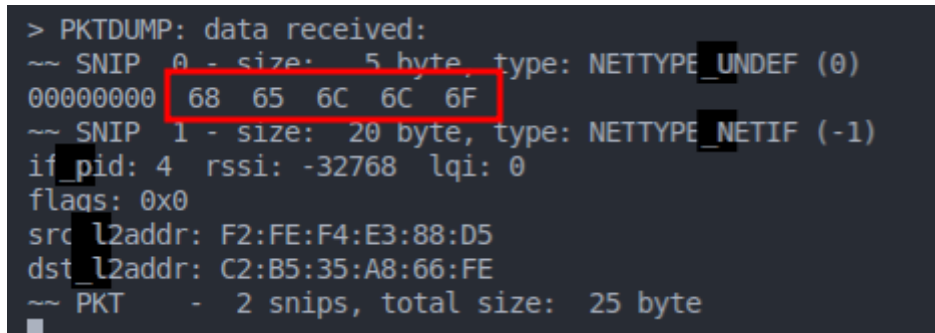
main(): This is RIOT! (Version: 2023.04-devel-199-g69c06)
Welcome to RIOT!
> ifconfig
ifconfig
Iface 4 HWaddr: B6:46:4F:29:49:0B
L2-PDU:1500 Source address length: 6
```

Abbildung 3: tap1 in RIOT-OS

Nun kann mit Hilfe des Befehls `txtsnd 4 C2:B5:35:A8:66:FE hello` eine Nachricht an ein anderes Interface gesendet werden. Der Befehl beinhaltet:

1. Die Interface Nummer auf der gesendet werden soll
2. Die Hardware Adresse des Ziels
3. Die Nachricht

Auf dem zweiten Instanz kann die gesendete Nachricht nun in Hexadezimaler Form empfangen werden.



```
> PKTDUMP: data received:
~ SNIP 0 - size: 5 byte, type: NETTYPE_UNDEF (0)
00000000 68 65 6C 6C 6F
~ SNIP 1 - size: 20 byte, type: NETTYPE_NETIF (-1)
if_pid: 4 rssi: -32768 lqi: 0
flags: 0x0
src_l2addr: F2:FE:F4:E3:88:D5
dst_l2addr: C2:B5:35:A8:66:FE
~ PKT - 2 snips, total size: 25 byte
```

Abbildung 4: Empfangene Nachricht

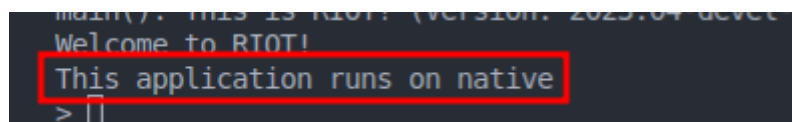
2 Woche 2

2.1 Challenge 1

Das Ziel ist die ersten vier Tutorials von RIOT durchzuarbeiten

2.1.1 Task 01

Einfügen der Code-Zeile `printf("This application runs on %s\n", RIOT_BOARD);` gibt den Hardware Typ aus für den RIOT Kompiliert wurde.



```
main(): THIS IS RIOT! (VERSION: 2023.04-devel)
Welcome to RIOT!
This application runs on native
> |
```

Abbildung 5: Ausgabe der neuen codezeile

2.1.2 Task 02

Die Funktion `echo()` aus dem Tutorial Code muss um die Zeile `printf("%s", argv[1]);` erweitert werden, um das erste Argument der Funktion zu auszugeben.

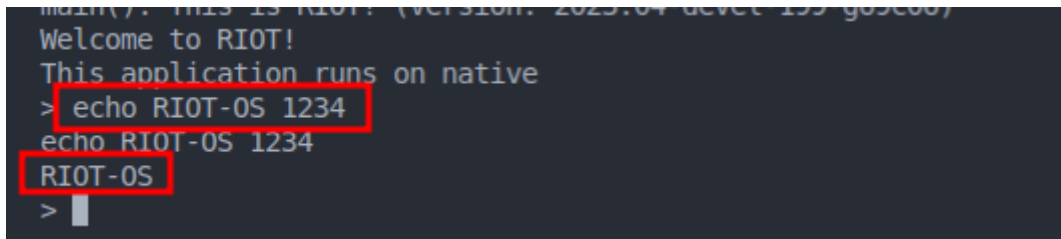
A terminal window showing the output of a program. The first line is "main(): This is RIOT! (Version: 2023.04-devel-199-g09c06)". The second line is "Welcome to RIOT!". The third line is "This application runs on native". The fourth line is "> echo RIOT-OS 1234". The fifth line is "echo RIOT-OS 1234". The sixth line is "RIOT-OS". The seventh line is ">". The text "RIOT-OS" is highlighted with a red box.

Abbildung 6: Ausgabe des Ersten Arguments

2.1.3 Task 03

Zum erstellen eines Threads muss die Multithreading library importiert und ein Array als Stack für den Thread erstellt werden.

```
1 ...  
2 #include "thread.h"  
3  
4 char rcv_thread_stack[THREAD_STACKSIZE_MAIN];  
5 ...
```

Jetzt kann mit `thread_create()` ein neuer Thread erstellt werden. Die Parameter enthalten eine Referenz auf den Stack und die Funktion die aufgerufen werden soll, diese wurde im Tutorial Code vorgegeben.

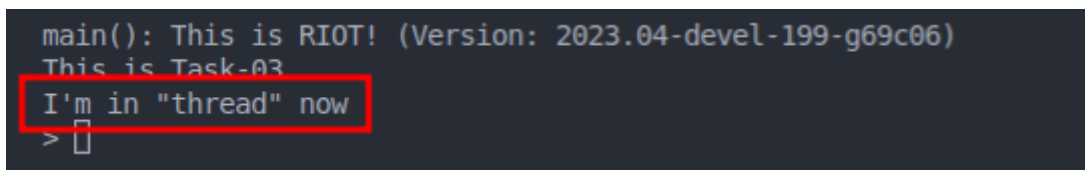
A terminal window showing the output of a program. The first line is "main(): This is RIOT! (Version: 2023.04-devel-199-g09c06)". The second line is "This is Task-03". The third line is "I'm in 'thread' now". The fourth line is ">". The text "I'm in 'thread' now" is highlighted with a red box.

Abbildung 7: Ausgabe des Threads

2.1.4 Task 04

Um Timer in RIOT-OS Nutzen zu Können muss im Makefile das Modul **xtimer** importiert werden:
`USEMODULE += xtimer`

In einem Thread wird nun mit `xtimer_now_usec()` die aktuelle Systemzeit in Millisekunden ausgegeben und dann mit `xtimer_sleep(2)` zwei Sekunden geschlafen.

```
1  ...
2  void *system_time(void *arg)
3  {
4      int time = 0;
5      while(true){
6          time = xtimer_now_usec();
7          printf("%d \n", time);
8          xtimer_sleep(2);
9      }
10
11     (void) arg;
12     return NULL;
13 }
14 ...
```

2.2 Challenge 2

In Chapter_2_Crypto wurden die beiden AES modes ECB und CBC erfolgreich kompiliert und eine Test message verschlüsselt und entschlüsselt. Dafür wurde der BASE_Pfad im Makefile angepasst und in der main() je der CIPHER_AES_128 zu CIPHER_AES geändert und den `include shell_commands` entfernt, da sie deprecated sind.

2.3 Challenge 3

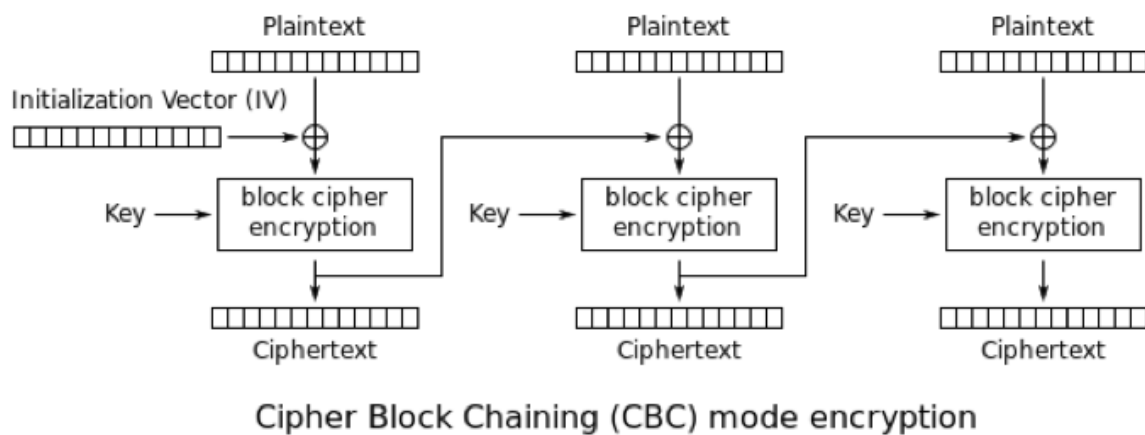
2.3.1 Review AES-CBC

Unter Chapter_2_Crypto befindet sich die Dokumentation zu 05_AES_CBC_en.md. Zu Beginn werden die Module cipher_modes (Für die verschiedenen AES cipher modes) und random (zur Generierung der IV) zum makefile hinzugefügt. Anschließend werden die zwei header Dateien für die Verschlüsselungs- und Entschlüsselungsmethoden vom CBC Mode hinzugefügt.

Für die Generierung eines cipher textes c1 Im CBC Mode wird der Plaintext mit der IV xor verknüpft und anschließend mit dem key verschlüsselt. Die IV ist dabei zufällig. Für den nächsten cipher text c2, dient c1 als IV.

Im main code wird nun zunächst ein Schlüssel, die message (plaintext) und cipher initialisiert. Anschließend folgen buffer Initialisierungen für input, output und decrypt.

Die IV ist eine zufällige, 16 byte große Zahl. Hier ist ein Verweis, dass man im productive mode einen

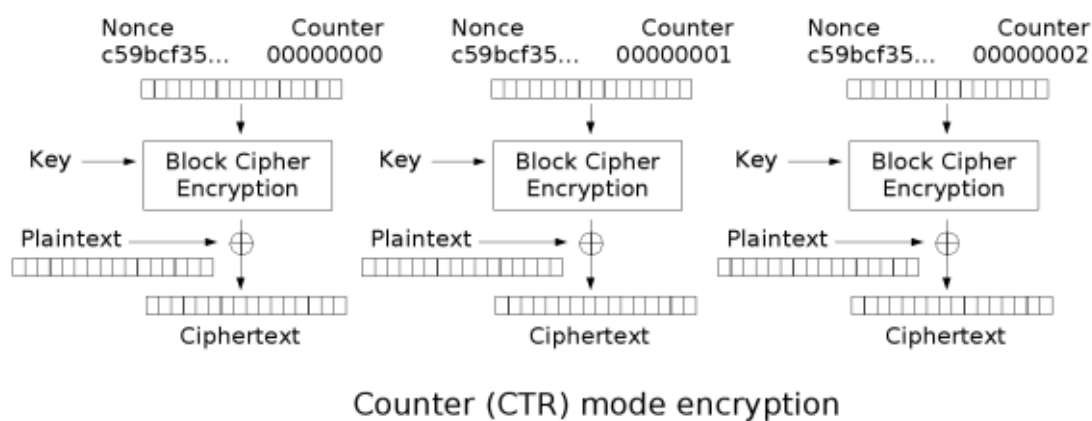
**Abbildung 8:** AES CBC Mode

kryptographischen Random Number Generator benutzen sollte, da die IV für jeden cipher text einzigartig sein muss.

Nun werden die `encrypt_cbc` und `decrypt_cbc` Methoden aufgerufen.

2.3.2 Additional Block Cipher Mode: CTR

In diesem Abschnitt wird der Code der obigen Review (AES CBC) zu AES_CTR abgeändert.

**Abbildung 9:** AES CTR Mode

Dafür definieren wir die folgenden Begriffe: - Nonce: Ist eine unique number. Diese wird oft Synonym mit der IV aus den anderen AES modes benutzt. - Counter: Eine weitere unique number, welche in

nachfolgenden Runden mit einer Inkrementier-Methode inkrementiert wird. - Counter Block: Ist die nonce konkatiert mit dem counter.

Der Counter Block ist wie die IV aus dem AES-CBC 16 byte groß, also 128 bit. Dieser muss in zwei Teile geteilt werden für die Nonce + Counter. Auf den initialen Counter wird eine Inkrementierungsmethode angewandt, diese ist standardmäßig: $(\text{Counter} + 1 \bmod 2^{\text{Counter_Block_length}})$. In NIST-SP800-38A Appendix B.2 ist eine der Herangehensweisen zur Initialisierung des Counter Blocks, dass die Nonce und der Counter genau die Hälfte des Counter Blocks einnehmen. Deswegen wählen wir eine 64 bit Nonce und einen 64 bit Counter. Diese werden wie in der Review des AES-CBC mit einer random number initialisiert, wobei in der Praxis ein kryptographisch sicherer Random Number Generator benutzt werden sollte.

Der Counterblock wird unter Verwendung des Keys mit einer AES encryption function verschlüsselt, um einen Keystream zu erzeugen, der dann mit dem Plaintext XOR-verknüpft wird, um den Ciphertext zu erzeugen. Also anders wie im CBC Mode, wo erst eine XOR-Verknüpfung des Plaintext mit der IV und anschließend eine Verschlüsselung stattfindet, wird bei CTR der Counter Block verschlüsselt und anschließend eine XOR-Verknüpfung mit dem Plaintext durchgeführt.

Für die Implementierung werden die Header Dateien mit den cipher_encrypt_ctr und cipher_decrypt_ctr hinzugefügt:

```
1 encrypt_ctr
2 int cipher_encrypt_ctr(const cipher_t *cipher, uint8_t counter_block,
    8, const uint8_t *input, size_t input_len, uint8_t *output);
3
4 //decrypt_ctr
5 int cipher_decrypt_ctr(const cipher_t *cipher, uint8_t counter_block,
    8, const uint8_t *input, size_t input_len, uint8_t *output);
```

Entsprechend werden folgende Zeilen des CBC-Codes für die main abgeändert:

- Anstatt von `Create IV`

```
1 /* ===== Create IV ===== */
2
3 uint8_t iv[16] = {0};
4 random_bytes(iv, 16); // IMPORTANT: In productive environment, use
    a cryptographically secure RNG!
```

wird ein Counter Block initialisiert: `uint8_t nonce[8] = {0}; random_bytes(nonce, 8); // IMPORTANT: In productive environment, use a cryptographically secure RNG!`

`uint8_t counter[8] = 1; // IMPORTANT: In productive environment, use a cryptographically secure RNG!`

`//Concat uint8_t* counter_block[16] = {0}; memcpy(counter_block, nonce, 8); memc-`

```
py(counter_block + 8, counter, 8);“
```

- Nun müssen die richtigen Methoden aufgerufen werden und im Output wird zusätzlich der Counter Block mit ausgegeben.

```
1  /* ===== Encryption and Decryption ===== */
2
3  if ((err = cipher_encrypt_ctr(&cipher, counter_block, 8, input,
4      total_len, output)) < 0) {
5      printf("Failed to encrypt data: %d\n", err);
6      return err;
7  }
8
9      if ((err = cipher_decrypt_ctr(&cipher, counter_block, 8,
10          output, total_len, decrypted)) < 0) {
11          printf("Failed to decrypt data: %d\n", err);
12          return err;
13      }
14
15  /* ===== Output ===== */
16
17  printf("Counter Block: ");
18  od_hex_dump(counter_block, 16, 0);
19  printf("\n\n");
20
21  printf("Plaintext:\n");
22  od_hex_dump(input, total_len, AES_BLOCK_SIZE);
23  printf("\n\n");
24
25  printf("Ciphertext:\n");
26  od_hex_dump(output, total_len, AES_BLOCK_SIZE);
27  printf("\n\n");
28
29  printf("Decrypted Ciphertext:\n");
30  od_hex_dump(input, total_len, AES_BLOCK_SIZE);
31  printf("\n\n");
```

2.4 Challenge 4

2.4.1 4.2. Understanding existing benchmarking code

Die Funktion “executeAesCbc()” in aes-cbc.c nimmt drei Parameter entgegen: numberOfRounds, key-Size und messageLength. Sie führt dann AES-CBC für die angegebene Anzahl von Runden wie folgt aus:

- Jede Runde generiert die Funktion eine zufällige Plaintext-Nachricht. Dann erzeugt sie einen zufälligen Initialisierungsvektor (IV). Danach wird das AES-Chiffrierobjekt mit einem zufällig

erzeugten Chiffrierschlüssel initialisiert.

- Der Verschlüsselungsprozess wird mit der Funktion `cipher_encrypt_cbc` durchgeführt, die den IV, den Plaintext und den `outputBuffer` als Eingabe erhält.
- Der Entschlüsselungsprozess wird mit `cipher_decrypt_cbc` durchgeführt und erhält wie eben IV, Output der `encrypt`-Funktion und den `decryptBuffer`
- Zum Ende werden jeweils die Rundenzeiten für `encrypt` und `decrypt` ausgegeben.

Die Funktion `“executeAesEcb()”` in `aes-ecb.c` ist gleich aufgebaut, wie die obere: Beide Funktionen erzeugen zufällige Eingabedaten und Verschlüsselungsschlüssel und führen die Ver- und Entschlüsselung durch, wobei der einzige Unterschied in der Art des verwendeten Algorithmus besteht (CBC vs. ECB).

2.4.2 4.3 Enhance existing benchmark code with AES-CTR

Genau wie in challenge 3 wurde der AES-CBC Algorithmus des Basisprojekts benutzt und an wenigen Zeilen zu CTR umgeändert. Dafür wurde in `/algorithm` die `aes-cbc.h` und `aes-cbc.c` kopiert und zu `aes-ctr` umbenannt. Im code wird dann die `ctr encrypt` und `decrypt` Methode aufgerufen und anstatt der zufällig generierten `IV[16]`, der zufällig generierte `block_counter[16]`, 8 übergeben, sodass wieder die ersten 8 bytes die Nuncce und die restlichen 8 bytes der Counter sind.

In der main werden dann alle drei Algorithmen nacheinander aufgerufen, um im nächsten Schritt Performance-Tests machen zu können.

2.4.3 4.4 Benchmark Tests

Mithilfe eines Python Scriptes wurde aus den Ergebnissen des Benchmarking codes für jede Messreihe der Durchschnitt und die Standardabweichung für Ver- und Entschlüsselung berechnet:

Aus der Tabelle ist zu erkennen, dass mit drei verschiedenen Größen an Nachrichten (2048-, 2^{16} - und 2^{17} -bytes) in Kombination mit drei verschiedenen Schlüsselgrößen (2048-, 2^{16} -, 2^{17} -bytes) getestet wurde. Unabhängig dieser Konfiguration, war die durchschnittliche Verschlüsselungslänge beim CBC- und CTR-Algorithmus etwa gleich lange. Interessanterweise unterscheiden sich die Entschlüsselungslängen. Beim CBC-Algorithmus dauerte die Entschlüsselung in allen Fällen fast doppelt so lange wie die Verschlüsselung. Die Entschlüsselung des CTR-Algorithmus dauerte dabei etwa gleich lange wie die Verschlüsselung.

Bei einer festen Schlüssellänge und der Verdoppelung der zu ver- und entschlüsselnden Nachrichtlänge von 2^{16} zu 2^{17} , stieg die Zeit in beiden Algorithmen (CBC, CTR) äquivalent an, also verdoppelte sich ebenso. Die Standardabweichung veränderte sich dabei schwankend gering.

Die Veränderung der Schlüssellänge von 16- zu 24-bytes verursachte etwa einen Offset von plus ~90ms auf alle Fälle.

Configuration		Algorithm	Encrypt		Dcrypt	
wordlength [byte]	key size [byte]		average [ms]	standard deviation [ms]	average [ms]	standard deviation [ms]
65536 (2 ¹⁶)	16	ECB	0.37	0.51	0.42	0.51
		CBC	571	118	1020	234
		CTR	539	74	540	73
2048 (2 ¹¹)	16	ECB	0.38	0.63	0.54	0.51
		CBC	19	4	34	8
		CTR	18	4	18	4
131072 (2 ¹⁷)	16	ECB	0.36	0.53	0.40	0.51
		CBC	1081	131	1917	215
		CTR	1072	165	1070	150
2048 (2 ¹¹)	24	ECB	0.56	0.51	0.94	0.64
		CBC	22	8	41	12
		CTR	18	4	18	4
65536 (2 ¹⁶)	24	ECB	0.40	0.53	0.47	0.53
		CBC	601	106	1132	202
		CTR	629	128	636	136
131072 (2 ¹⁷)	24	ECB	0.36	0.51	0.43	0.51
		CBC	1120	102	2110	157
		CTR	1122	126	1128	134

Abbildung 10: AES CTR Mode

Die Standardabweichung lag meistens bei ~100ms, außer bei der Entschlüsselung des CBC-Algorithmus, dort lag sie bei ~200ms, wobei sich wie oben beschrieben auch die Entschlüsselungsdauer generell verdoppelte.

3 Woche 4

Leider gab es zu Beginn des RIOT Projekts Startschwierigkeiten mit der VM. Die Challenges aus Woche 3 haben einen wesentlich größeren Aufwand verursacht, als erwartet. Demnach fehlt uns leider die Zeit das Kapitel 4 abzuschließen.