08_Benchmarking.md

Benchmarking

Zum Schluss werden wir Benchmarks für die einzelnen Krypto-Algorithmen durchführen.

Makefile

```
USEPKG += relic

USEMODULE += shell
USEMODULE += shell_commands
USEMODULE += crypto_aes
USEMODULE += cipher_modes
USEMODULE += random
USEMODULE += xtimer             # zeitbezogene Funktionen wie Timer
USEMODULE += printf_float       # Float Konsolenausgaben
```

Neu hinzugekommen sind die Module `xtimer` für Zeitmessungen und `printf_float`, um Konsolenausgaben von Float-Werten auf IoT Hardware zu ermöglichen.

Headerdatei `include\bench.h`

```
#ifndef BENCH_H
#define BENCH_H

#include "relic.h"

#include <stdint.h>
#include <stdbool.h>

typedef bool (*bench_func)(void *args);
typedef void (*after_iter_func)(void *args);

struct benchmark_t
{
    int n_iterations;
    uint64_t min;
    uint64_t max;
    uint64_t total;
    float avg;
    bench_func iteration_func;
    after_iter_func after_iteration_func;
};

void print_bench_results(struct benchmark_t *bench);

bool run_benchmark(struct benchmark_t *result_ptr, int n_iterations, void *args);

#endif /* BENCH_H */
```

Das Header stellt die Struktur `benchmark_t` zur Verfügung, um Informationen wie die Anzahl an Iterationen und verschiedenen Zeitwerten in Mikrosekunden zu speichern.

Die beiden Pointer vom Typ `iteration_func` bzw. `after_iteration_func` zeigen auf Funktionen, die während einer Iteration bzw. zwischen mehereren Iterationen ausgeführt werden sollen.

Die konkreten Konsolenausgaben mit den Benchmark-Werten finden in der Funktion `print_bench_results` statt, während mit `run_benchmark` ein kompletter Benchmark für den gewünschten Krypto-Algorithmus mit gewählter Anzahl an Iterationen ausgeführt wird.

`bench_aes_ecb.c` : AES-ECB Benchmark

Im Folgenden wird am Beispiel von AES-ECB gezeigt, wie sich ein Benchmark für ein bestimmtes Verschlüsselungsverfahren implementieren lässt.

```
#include "include/bench_aes.h"

#include "board.h"

#include <stdbool.h>

static uint8_t key[AES_KEY_SIZE] = {
    0x64, 0x52, 0x67, 0x55,
    0x6B, 0x58, 0x70, 0x32,
    0x73, 0x35, 0x75, 0x38,
    0x78, 0x2F, 0x41, 0x3F};

struct aes_ecb_bench_params_t
{
    cipher_t cipher;
    uint8_t *input;
    uint8_t *output;
    int n_blocks;
    size_t total_buf_len;
};
```

Neben dem schon zuvor bekannten AES Key, werden in der Struktur `aes_ecb_bench_params_t` Verschlüsselungsparameter wie `cipher_t` Struktur, Input, Output, Anzahl Blöcke und Bufferlänge gespeichert. Diese Parameter werden von Iteration zu Iteration weitergegeben.

```
bool aes_ecb_bench_iteration(void *args)
{
    struct aes_ecb_bench_params_t *params = args;

    int err;
    int block = 0;

    for (size_t offset = 0; block < params->n_blocks; offset = AES_BLOCK_SIZE * ++block)
    {
        err = cipher_encrypt(&params->cipher, params->input + offset, params->output + offset);

        if (err != 1)
        {
            break;
        }
    }

    if (err != 1)
    {
        printf("Failed to Encrypt: %d\n", err);
        return false;
    }

    return true;
}
```

Eine einzelne Iteration entspricht hier einem Verschlüsselungsvorgang. Diese Iteration wird durch die Funktion `aes_ecb_bench_iteration` dargestellt.

```
void aes_ecb_bench_after_iteration(void *args)
{
    struct aes_ecb_bench_params_t *params = args;
    memcpy(params->input, params->output, params->total_buf_len);
}
```

Aus Performancegründen ist es nicht sinnvoll, für jede Iteration einen neuen Input mit `rand_bytes` zu erstellen. Deshalb werden Input und Output jeder Iteration miteinander verkettet: der Output-Buffer aus der letzten Iteration wird in den Input-Buffer für die nächste Iteration mit `memcpy` kopiert.

```
void bench_aes_cbc(int n_iterations, int n_blocks)
{

    /*
Board: iotlab-m3
MCU: stm32
AES-CBC Benchmark Summary:
 Using normal loops
 Calculating T-Tables on the fly
 # of blocks: 4 ( = 64 Bytes)
 # of Iterations: 10000
 Total usec: 1890092
 Average usec: 189.009201
 Min usec: 188
 Max usec: 196
     */

    struct benchmark_t bench;

    bench.iteration_func = aes_cbc_bench_iteration;
    bench.after_iteration_func = aes_cbc_after_iteration;

    struct aes_cbc_bench_params_t params = {
        .n_blocks = n_blocks,
        .total_buf_len = n_blocks * AES_BLOCK_SIZE};

    params.input = malloc(params.total_buf_len);
    params.output = malloc(params.total_buf_len);

    int err = cipher_init(&params.cipher, CIPHER_AES_128, key, AES_KEY_SIZE);

    if (err != CIPHER_INIT_SUCCESS)
    {
        printf("Failed to init cipher: %d\n", err);
        return;
    }

    rand_bytes(params.iv, 16);
    rand_bytes(params.input, params.total_buf_len);

    if (!run_benchmark(&bench, n_iterations, &params))
    {
        return;
    }

    printf("Board: %s\n", RIOT_BOARD);
    printf("MCU: %s\n", RIOT_MCU);
    printf("AES-CBC Benchmark Summary:\n");
#ifdef MODULE_CRYPTO_AES_UNROLL
    printf(" Using unrolled loops\n");
#else
    printf(" Using normal loops\n");
#endif

#ifdef MODULE_CRYPTO_AES_PRECALCULATED
```

```
    printf(" Using Precalculated T-Tables\n");
#else
    printf(" Calculating T-Tables on the fly\n");
#endif

    printf(" # of blocks: %d ( = %d Bytes)\n", n_blocks, n_blocks * AES_BLOCK_SIZE);
    print_bench_results(&bench);

    free(params.input);
    free(params.output);
}
```

Mit der Funktion `bench_aes_cbc` wird der Benchmark für AES-CBC mit den von Nutzern angegebenen Parametern in Gang gesetzt. Der initiale Input wird zunächst mit `rand_bytes` zufällig generiert, und dann der Benchmark mit `run_benchmark` tatsächlich gestartet. Im Anschluss darauf folgen sämtliche Konsolenausgaben mit den Benchmark-Ergebnissen, und zuletzt werden Input- und Output-Buffer wieder freigegeben.

main.c

```
#if CP_RSAPD == PKCS1
#define RSA_PAD_LEN             (11)
#elif CP_RSAPD == PKCS2
#define RSA_PAD_LEN             (2 * MD_LEN + 2)
#else
#define RSA_PAD_LEN             (2)
#endif
```

Hier wird die RSA Paddinggröße festgelegt, abhängig vom gewählten Paddingverfahren (PKCS1, PKCS2)

```
int bench_command(int argc, char **argv)
{
    if (argc < 3)
    {
        printf("Syntax: \n");
        printf(" - %s aes-ecb <iterations> [blocks]\n", argv[0]);
        printf(" - %s aes-cbc <iterations> [blocks]\n", argv[0]);
        printf(" - %s rsa <iterations> [block size]\n", argv[0]);

        return 1;
    }

    int n_iterations;

    if (sscanf(argv[2], "%d", &n_iterations) != 1)
    {
        printf("Could not parse iteration count\n");
        return 1;
    }

    int rsa_block_size = 128 - RSA_PAD_LEN;
    int n_blocks = 4;

    if (argc >= 4)
    {
        if (sscanf(argv[3], "%d", &n_blocks) != 1)
        {
            printf("Could not parse block count\n");
        } else {
            rsa_block_size = n_blocks;
        }
    }
```

```
    if (STR_EQ("aes-ecb", argv[1]))
    {
        bench_aes_ecb(n_iterations, n_blocks);
    }
    else if (STR_EQ("aes-cbc", argv[1]))
    {
        bench_aes_cbc(n_iterations, n_blocks);
    }
    else if (STR_EQ("rsa", argv[1])) {
        bench_rsa(n_iterations, rsa_block_size);
    }
    else
    {
        printf("Unknown command: %s\n", argv[1]);
        return 1;
    }

    return 0;
}

int main(void)
{
    core_init();

    shell_command_t commands[] = {
        {"bench", "Benchmark Crypto Algorithms", bench_command},
        {NULL, NULL, NULL}};

    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(commands, line_buf, SHELL_DEFAULT_BUFSIZE);

    core_clean();

    return 0;
}
```

Mit dem Custom Befehl `bench` können wir auf der RIOT Shell einen Benchmark starten. Wie schon aus dem Code hervorgeht, sind folgende Angaben als Parameter erforderlich:

1. Verschlüsselungsverfahren (`aes-ecb` , `aes-cbc` , `rsa`)
2. Anzahl Iterationen
3. Anzahl Blöcke/Blockgröße

Konsolenausgaben

Im Folgenden einige beispielhafte Benchmark-Ergebnisse:

AES-ECB:

```
> bench aes-ecb 1000 1000
bench aes-ecb 1000 1000
Board: native
MCU: native
AES-ECB Benchmark Summary:
 Using normal loops
 Calculating T-Tables on the fly
 # of blocks: 1000 ( = 16000 Bytes)
 # of Iterations: 1000
 Total usec: 263014
 Average usec: 263.014008
 Min usec: 255
 Max usec: 370
```

AES-CBC:

```
> bench aes-cbc 1000 1000
bench aes-cbc 1000 1000
Board: native
MCU: native
AES-CBC Benchmark Summary:
 Using normal loops
 Calculating T-Tables on the fly
 # of blocks: 1000 ( = 16000 Bytes)
 # of Iterations: 1000
 Total usec: 312433
 Average usec: 312.433014
 Min usec: 293
 Max usec: 612
```

RSA:

```
> bench rsa 1000 100
bench rsa 1000 100
Board: native
MCU: native
RSA Benchmark SummaryRSA Bits: 1024
RSA Padding: PKCS1
Input Size: 100 Bytes
 # of Iterations: 1000
 Total usec: 423579
 Average usec: 423.579010
 Min usec: 409
 Max usec: 569
```

Zurück zum Index

Zurück zu Teil 7: RSA Verschlüsselung mithilfe des RELIC-Tookits