Escola Politécnica da USP

Departamento de Engenharia de Computação e Sistemas Digitais

PCS3722 – Arquitetura e Organização de Computadores

Profª Cíntia Borges Margi

Projeto - Fase 1

Profiling de AES

Bruno Susko Marcellini

Eric Rodrigues Pires

Mateus Nakajo de Mendonça

Tiago Koji Castro Shibata

Data: 18/06/2017

1. Introdução

Neste relatório serão apresentados os passos tomados para a execução do trabalho proposto, desde a escolha do algoritmo utilizado até os testes realizados para a 1ª fase do projeto.

2. Atividades

O trabalho foi dividido em 7 etapas distintas:

- Escolha de algoritmo
- História e funcionamento do AES
- Escolha do modo de operação
- Escolha de implementação “Golden”
- Testes em desktop
- Benchmarks
- Escolha de funções para implementação em hardware

2.1 Escolha de algoritmo

Foram consideradas diferentes opções para o algoritmo do projeto, mas rapidamente foi definido que realizaríamos algo em torno de criptografia devido ao fato de ser uma aplicação justificável para um hardware dedicado (operação repetitiva e com custo computacional relevante).

Foram considerados todas as opções disponíveis no MiBench, como Blowfish, SHA, PGP e Rijndael. No final decidimos escolher o AES (Advanced Encryption Standard), um tipo específico do Rijndael, que é atualmente utilizado em diversas aplicações, mesmo sendo um padrão relativamente recente, devido à facilidade de execução e ao pouco uso de memória. O algoritmo é considerado bastante seguro e confiável.

2.2 História e Funcionamento do AES

O Advanced Encryption Standard é um algoritmo de criptografia baseado em blocos, adotado pelo governo dos EUA como padrão em 2001. Ele foi selecionado em um concurso do National Institute of Standards and Technology (NIST), após o padrão anterior, DES, ter sido considerado inseguro. O algoritmo original vencedor do processo seletivo, Rijndael – fusão do nome dos criadores belgas, Vincent Rijmen e Joan Daemen –, permite chaves de diversos tamanhos para criptografar um bloco de 128 bits; embora comumente se utilize apenas chaves de 128 bits na atualidade.

O AES é uma cifra de blocos, isto é, realiza operações determinísticas em grupos de bits como se fossem uma unidade, utilizando uma chave simétrica para transformação direta e inversa (ou seja, usa uma única chave compartilhada para criptografar e descriptografar a mensagem). Cada bloco é um arranjo bidimensional de bytes com 4x4 posições, constituindo 128 bits; portanto, pode ser distribuído em 4 inteiros de 32 bits em uma implementação de hardware, o que nos será útil posteriormente.

São realizadas quatro estágios de operação sobre os bytes da matriz de estado da mensagem durante a criptografia em cada passo:

1. AddRoundKey - Cada byte é combinado via OU-exclusivo com a subchave própria do passo (RoundKey). Esta subchave é derivada da chave principal usando o algoritmo de escalonamento de chaves, que pode utilizar rotações de byte, Rcon (constante arredondada – obtida pela exponenciação de polinômio 2) e S-box (mais detalhes no próximo item). Como Rcon é uma matriz determínistca, pode ser implementada como tabela em cache para hardware.
2. SubBytes - Cada byte será atualizado de acordo com uma tabela de substituição (S-box), calculada a partir de uma matriz inversa calculada para a solução de Rijndael por trás da cifra. Na implementação em hardware, pode-se utilizar uma tabela pré-calculada como cache para acelerar a etapa.
3. ShiftRows - Cada fileira do arranjo 4x4 de bytes, exceto a primeira, é deslocada de determinado número de posições (a fileira n é rotacionada n-1 bytes para a esquerda).

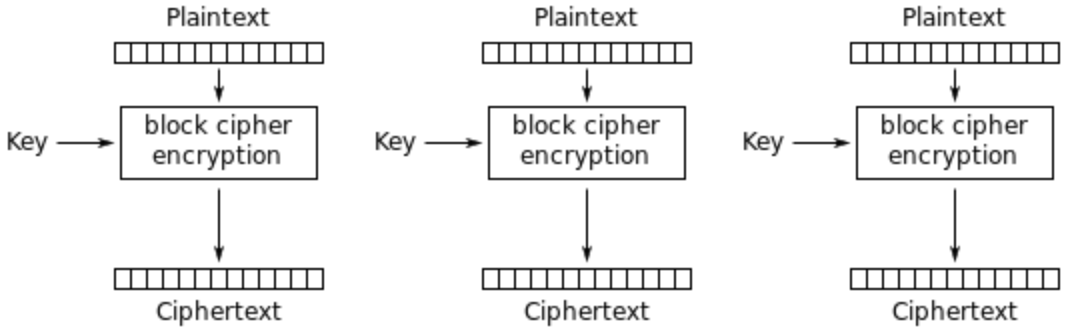
4. MixColumns - Cada coluna do arranjo 4x4 de bytes é transformada, multiplicando-se por uma matriz fixa e invertível.

O turno final substitui o estágio de MixColumns por um novo estágio de AddRoundKey. Para descriptografar a mensagem, apenas realizam-se as operações inversas com a mesma chave, com uma S-box apropriadamente modificada (AddRoundKey é a própria inversa, portanto não é alterada).

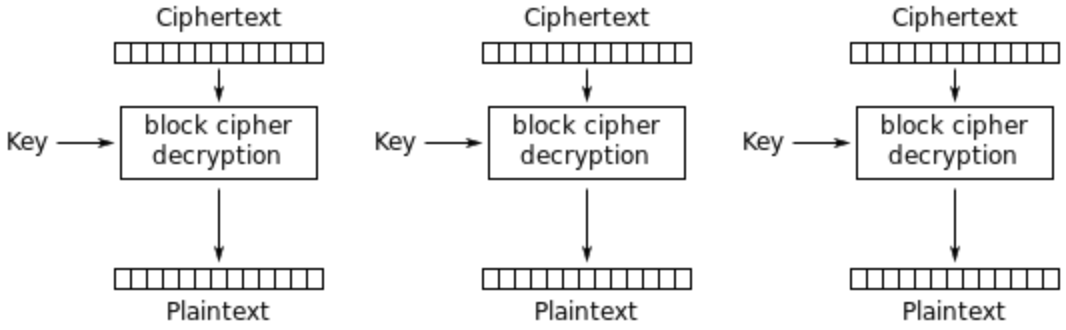
2.3 Escolha de modo de operação

Dada uma cifra de bloco, podemos escolher entre alguns modos de operação padrões para cifrar vários blocos de dados. Explicaremos rapidamente os modos de operação oficiais do AES recomendados pelo NIST (CBC, CFB, OFB e CTR) e o modo ECB. Não entraremos em detalhes em cada um deles, apenas citando as vantagens que nos fizeram optar pelo CTR.

O modo **ECB** (electronic codebook) consiste em apenas passar os blocos pelo algoritmo de cifragem. Para recuperar os dados, os blocos passam pelo de decifragem.



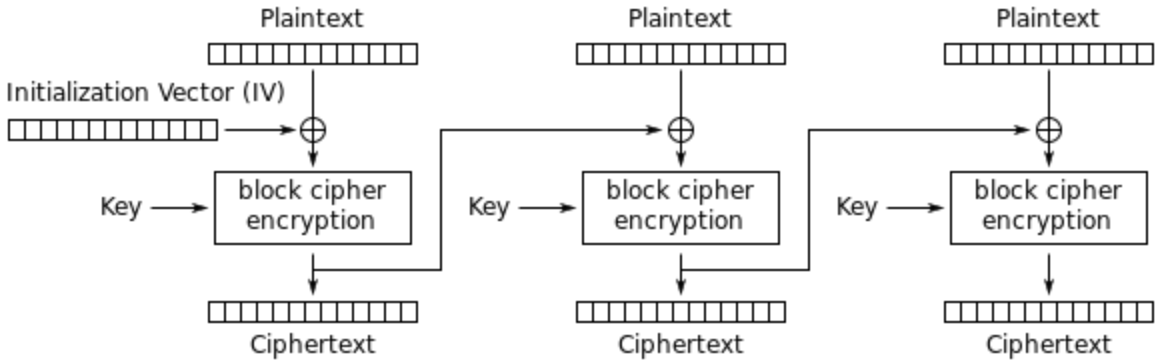
Electronic Codebook (ECB) mode encryption



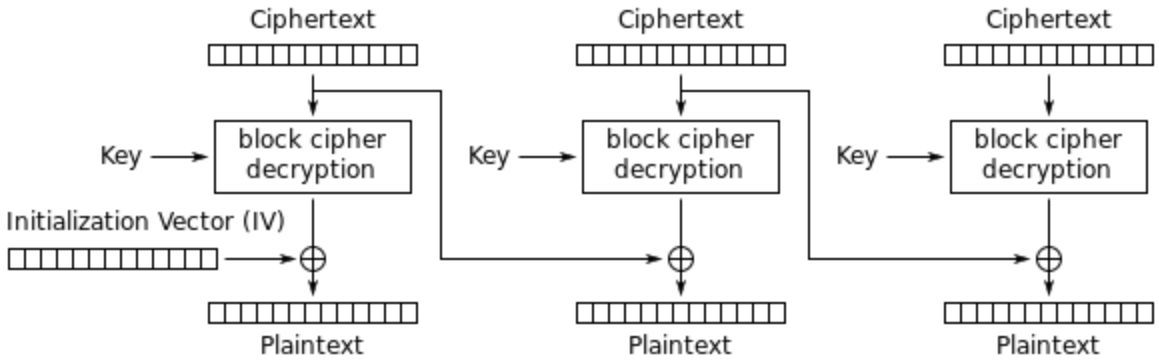
Electronic Codebook (ECB) mode decryption

O modo ECB não é comumente usado e é apenas útil para cifrar blocos que nunca se repitam. Se dois blocos em texto plano forem iguais, um atacante saberá que eles são iguais ao observar dois blocos cifrados iguais.

O modo **CBC** (Chipher Block Chaining) requer um vetor de inicialização (IV) aleatório. O primeiro bloco passará por ou exclusivo com o IV antes de passar pela cifra, e blocos subsequentes passarão por ou exclusivo com a saída cifrada do bloco anterior.



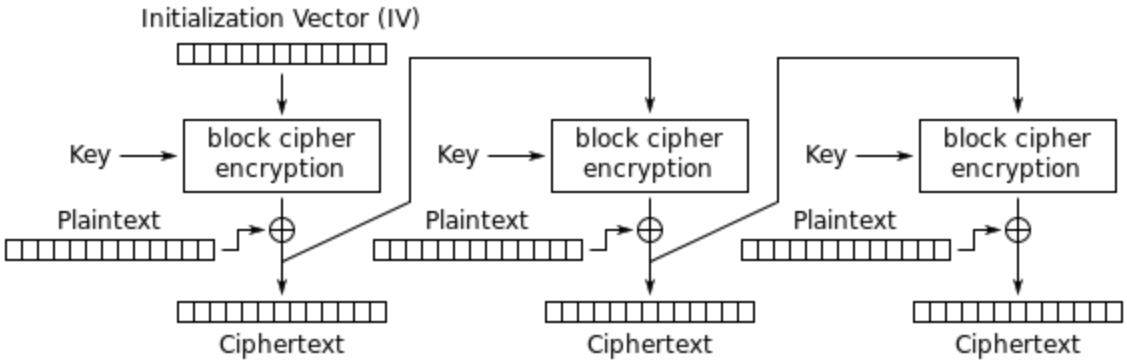
Cipher Block Chaining (CBC) mode encryption



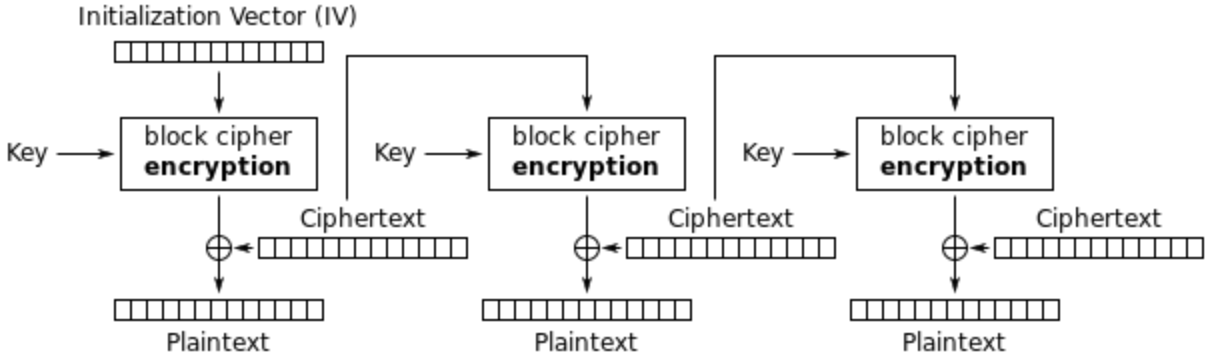
Cipher Block Chaining (CBC) mode decryption

Se um atacante não tiver controle sobre o IV e a cifra for segura, não é possível extrair informações da entrada observando as saídas. O fato do IV ser aleatório e das saídas das cifras (que possuem distribuição aleatória) passarem por ou exclusivo com as próximas entradas torna as saídas dos blocos indiferenciáveis a partir das entradas, diferentemente do modo ECB. Em CBC, a cifragem é obrigatoriamente serial (pois depende da saída do bloco anterior), mas a decifragem pode ser paralelizada (depende da cifra do bloco anterior e do atual, e não de toda a série de blocos antes do atual).

O modo CFB (Chipher Feedback) usa a cifra de bloco do AES para atuar de maneira similar a uma cifra de fluxo. Uma série de blocos é gerada e passa por ou exclusivo com o texto plano para gerar os dados cifrados. A primeira iteração é feita com um IV aleatório, e as próximas com feedback da cifra do último bloco.



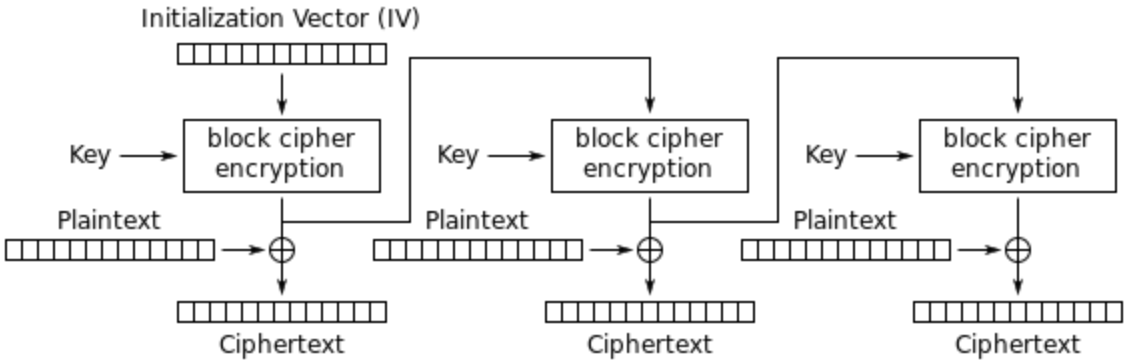
Cipher Feedback (CFB) mode encryption



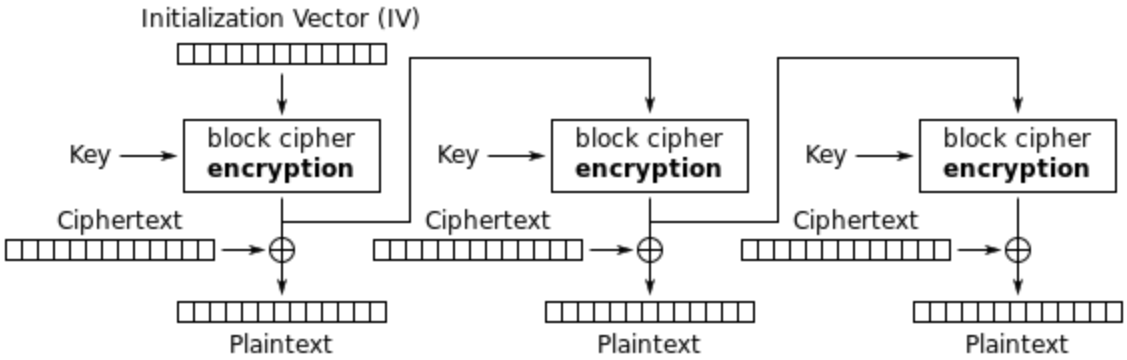
Cipher Feedback (CFB) mode decryption

As possibilidades de paralelização da decifragem são as mesmas do modo CBC. Uma vantagem do modo CFB é que apenas uma função de cifra de bloco é necessária, e não seu inverso. Isso ocorre pois o inverso do ou-exclusivo é o próprio ou-exclusivo, portanto o mesmo fluxo é usado na decifragem.

O modo OFB (Output Feedback) também aplica uma cifra de bloco como se fosse uma cifra de fluxo. A saída do bloco é usada para alimentar o próximo bloco.



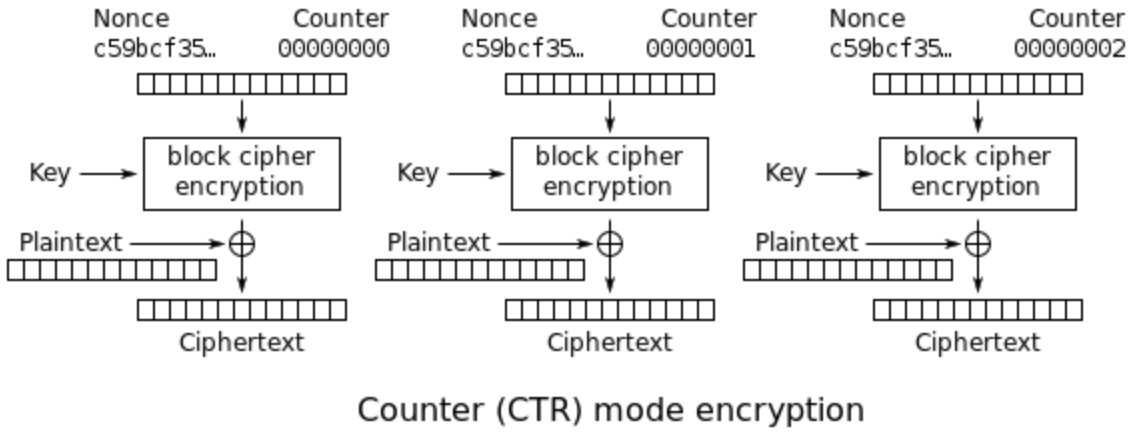
Output Feedback (OFB) mode encryption



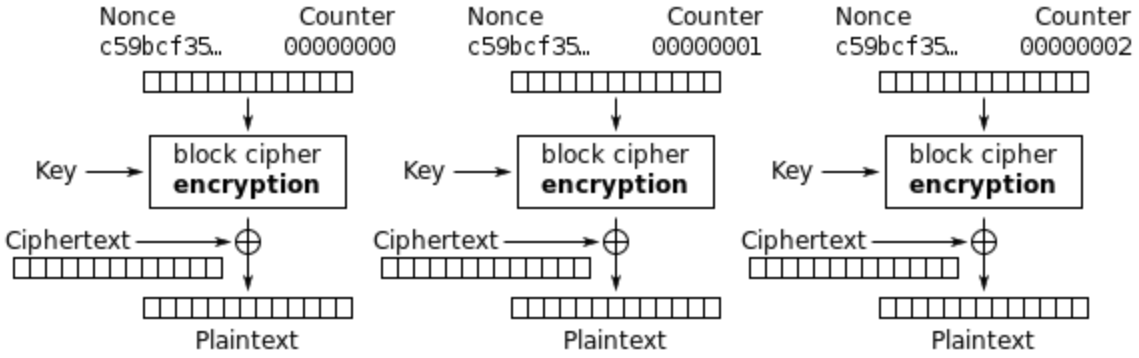
Output Feedback (OFB) mode decryption

Esse modo tem operação parecida com o CFB, mas não permite serialização de nenhuma das duas etapas.

O modo CTR (Counter) também aplica a cifra de bloco como se fosse uma cifra de fluxo, gerando um fluxo de bits e aplicando ou exclusivo sobre os dados planos. No entanto, não há realimentação entre os blocos; é usado um contador com um valor diferente em cada bloco (o contador pode ser qualquer valor que não se repita entre blocos, inclusive um contador incremental).



Counter (CTR) mode encryption



Counter (CTR) mode decryption

O modo CTR foi escolhido pelo grupo, por ser simples e paralelizável entre blocos. Seria possível implementar dois blocos em paralelo em hardware, apenas duplicando a mesma implementação. Além disso, o fato de apenas uma operação da cifra de bloco ser usada (apenas cifragem) significa que precisamos de menos código e menos tabelas de lookup.

2.4 Escolha de implementação "Golden"

Estudamos algumas implementações disponíveis. Tomamos alguns critérios de escolha:

- **Confiabilidade:** A implementação deve vir de fonte conhecida e confiável e ter sido testada e usada em outros projetos.
- **Completude:** A implementação deve suportar AES com modo de operação CTR, escolhido para esse projeto.
- **Facilidade de uso em sistemas embarcados:** A implementação deve ser capaz de rodar e ter sido testada em sistemas embarcados e não depender de nenhuma instrução específica de hardware (por exemplo, o conjunto de instruções AES-NI em x86-64).

- **Facilidade de portar para nosso projeto:** Se a implementação de AES é parte de uma biblioteca maior, consideramos a facilidade de extrair apenas a implementação do AES da biblioteca.
- **Casos de teste:** Se a implementação possuir casos de teste, será mais fácil para testarmos nosso projeto. Se não, podemos pegar os casos de teste oficiais fornecidos pelo NIST.

Considerando esses requisitos, escolhemos como "golden" a biblioteca *mbed TLS* (https://tls.mbed.org/). A *mbed* é uma plataforma com um conjunto de bibliotecas para dispositivos ARM Cortex-M, criada pela ARM e empresas parceiras, com foco em IoT. A *mbed TLS* é uma implementação de vários algoritmos de criptografia para suportar TLS em dispositivos de baixo processamento e memória. A implementação de AES suporta quase todos os modos de operação oficiais (CBC, CFB, CTR, sendo apenas OFB não implementado), incluindo o CTR, que é de nosso interesse. Há vetores de testes do NIST na biblioteca, que pode ser compilada com execução de testes.

2.5 Testes em desktop

Para os testes, modificamos algumas configurações fornecidas pelo *mbed TLS* para AES para torná-lo mais próximo do que desejamos usar no projeto. As configurações foram editadas no arquivo config.h:

- Desativamos todos os modos, fora CTR.
- Desativamos o suporte a instruções de hardware para AES (extensão VIA Padlock em x86 e AES-NI em x86-64), que invalidariam o benchmark.
- Ativamos o uso de tabelas pré-computadas.
- Desativamos a etapa de decifragem do AES.

Nos últimos dois pontos, vale lembrar que para o modo CTR não precisamos da decifragem do AES, o que apresenta também a vantagem de excluir praticamente metade das tabelas pré-computadas usadas. Usamos tabelas pré-computadas pois não requerem uma quantidade muito grande de ROM e reduzem o tamanho do código.

Criamos um pequeno programa para chamar a função de testes da biblioteca:

```
#include <stdio.h>
#include <string.h>

#include "mbedtls/aes.h"
#include "mbedtls/config.h"

#ifndef MBEDTLS_SELF_TEST
#error MBEDTLS_SELF_TEST must be defined to compile tests
#endif

int main(int argc, char **argv)
{
    int verbose = 0;

    switch (argc) {
    case 1:
```

```
        break;

    case 2:
        if (!strcmp(argv[1], "-v")) {
            verbose = 1;
            break;
        }
        // fall-through

    default:
        fprintf(stderr, "Usage: %s [-v]\n"
                "\t-v\tVerbose mode.\n", argv[0]);
        return -1;
    }

    if (verbose)
        printf("=== Running AES tests ===\n");
    return mbedtls_aes_self_test(verbose);
}
```

Os testes passaram. Aproveitamos o programa para também analisar o uso estático de memória. Esse programa inicial, incluindo as funções de teste, foi bastante bom; o segmento text, que mantém código e dados constantes, possui 13566 bytes, com uso de tabelas pré-computadas:

```
$ size run_test
   text    data     bss     dec     hex filename
  13566     584      16   14166    3756 run_test
```

Em seguida, desativamos o suporte a chaves maiores que 128 bits. O tamanho final do programa de teste e apenas do objeto da biblioteca foram:

```
$ size run_test
   text    data     bss     dec     hex filename
  11974     584      16   12574    311e run_test
$ size CMakeFiles/run_test.dir/src/aes.c.o
   text    data     bss     dec     hex filename
   9379       0       0    9379    24a3 CMakeFiles/run_test.dir/src/aes.c.o
```

E ao compilar sem testes, o tamanho apenas da biblioteca foi:

```
$ size CMakeFiles/run_test.dir/src/aes.c.o
   text    data     bss     dec     hex filename
   7356       0       0    7356    1cbc CMakeFiles/run_test.dir/src/aes.c.o
```

Não sabemos ainda as restrições de memória que teremos no projeto, mas consideramos 7356 bytes um tamanho seguro a ser usado. Após essa análise acreditamos que a biblioteca caberá na ROM do softcore e na FPGA.

2.6 Benchmarks

Realizamos benchmarks com os programas gprof, perf e callgrind. Os benchmarks foram realizados com otimizações ativadas, para melhor representar o uso que faremos da biblioteca, símbolos de depuração ativados e uso de frame pointer em todas as chamadas. As opções `-g -fno-omit-frame-pointer` foram usadas em todos os benchmarks e `-pg` nos benchmarks com gprof.

A implementação do *mbed TLS* é bastante sucinta, com uma única função chamada para cifragem. Essa função utiliza algumas macros para realizar os rounds do algoritmo:

```
/*
 * AES-ECB block encryption
 */
#if !defined(MBEDTLS_AES_ENCRYPT_ALT)
void mbedtls_aes_encrypt( mbedtls_aes_context *ctx,
                          const unsigned char input[16],
                          unsigned char output[16] )
{
    int i;
    uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;

    RK = ctx->buf; /* ctx->rk; */

    GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
    GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
    GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
    GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;

    for( i = ( 10 /* ctx->nr */ >> 1 ) - 1; i > 0; i-- )
    {
        AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
        AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
    }

    AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );

    X0 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );

    X1 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y1       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );
```

```
    X2 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y2       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );

    X3 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );

    PUT_UINT32_LE( X0, output,  0 );
    PUT_UINT32_LE( X1, output,  4 );
    PUT_UINT32_LE( X2, output,  8 );
    PUT_UINT32_LE( X3, output, 12 );
}
#endif /* !MBEDTLS_AES_ENCRYPT_ALT */
```

gprof e perf não foram muito efetivos, indicando apenas duas folhas com as chamadas de preparação da chave e cifragem (mbedtls_aes_setkey_enc e mbedtls_aes_encrypt):

```
./run_test ; gprof ./run_test > report.txt ; gprof2dot.py report.txt | dot -Tpng
-o profile.png
```

Grafo de chamadas via gprof

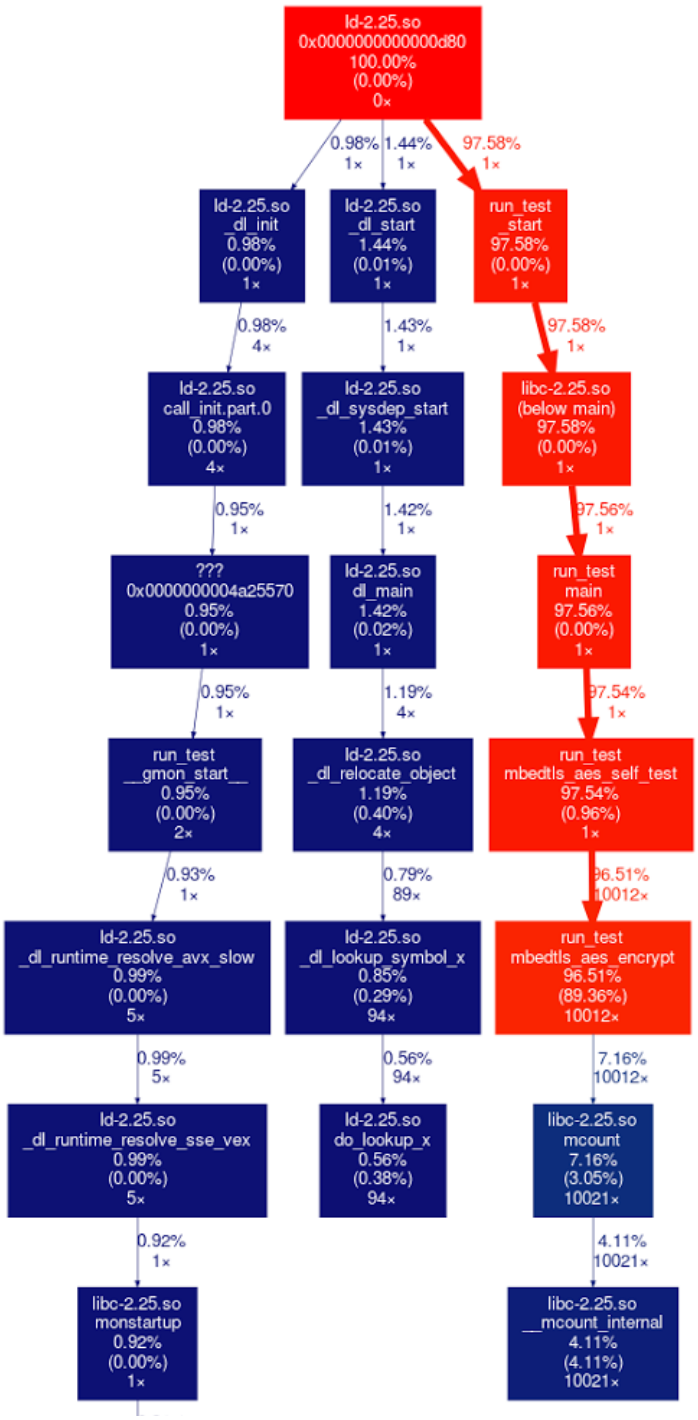
```
perf record ./run_test
perf script
```

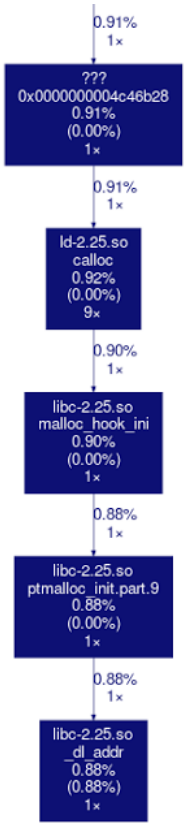
```
     run_test 23732 18070.619214:          1 cycles:u:      7f415846ad80 _start (/usr/lib/ld-2.25.so)
       run_test 23732 18070.619217:          1 cycles:u:      7f415846ad80 _start (/usr/lib/ld-2.25.so)
       run_test 23732 18070.619218:         11 cycles:u:      7f415846ad80 _start (/usr/lib/ld-2.25.so)
       run_test 23732 18070.619218:        350 cycles:u:      7f415846ad80 _start (/usr/lib/ld-2.25.so)
       run_test 23732 18070.619230:     253364 cycles:u:      7f415846bda3 dl_main (/usr/lib/ld-2.25.so)
       run_test 23732 18070.619372:     999304 cycles:u:            401063 mbedtls_aes_encrypt
(/home/tiago/code/AES-FPGA/build/run_test)
       run_test 23732 18070.619693:     966184 cycles:u:            401013 mbedtls_aes_encrypt
(/home/tiago/code/AES-FPGA/build/run_test)
```

Contagem de ciclos gastos em cada função via perf

O uso de callgrind gerou um gráfico um pouco melhor, apesar de ainda pouco informativo. No entanto, pudemos gerar também o código fonte anotado com número de instruções executadas por linha. Com o código fonte anotado pudemos observar os trechos de código com maior consumo de tempo.

```
valgrind --tool=callgrind ./run_test
gprof2dot.py -f callgrind callgrind.out.<PID> | dot -Tpng -o output.png
```


ld-2.25.so
0x0000000000000d80
100.00%
(0.00%)
0×
0.98%
1×
1.44%
1×
97.58%
1×
ld-2.25.so
_dl_init
0.98%
(0.00%)
1×
ld-2.25.so
_dl_start
1.44%
(0.01%)
1×
run_test
_start
97.58%
(0.00%)
1×
0.98%
4×
1.43%
1×
97.58%
1×
ld-2.25.so
call_init.part.0
0.98%
(0.00%)
4×
ld-2.25.so
_dl_sysdep_start
1.43%
(0.01%)
1×
libc-2.25.so
(below main)
97.58%
(0.00%)
1×
0.95%
1×
1.42%
1×
97.56%
1×
???
0x0000000004a25570
0.95%
(0.00%)
1×
ld-2.25.so
dl_main
1.42%
(0.02%)
1×
run_test
main
97.56%
(0.00%)
1×
0.95%
1×
1.19%
4×
97.54%
1×
run_test
__gmon_start__
0.95%
(0.00%)
2×
ld-2.25.so
_dl_relocate_object
1.19%
(0.40%)
4×
run_test
mbedtls_aes_self_test
97.54%
(0.96%)
1×
0.93%
1×
0.79%
89×
96.51%
10012×
ld-2.25.so
_dl_runtime_resolve_avx_slow
0.99%
(0.00%)
5×
ld-2.25.so
_dl_lookup_symbol_x
0.85%
(0.29%)
94×
run_test
mbedtls_aes_encrypt
96.51%
(89.36%)
10012×
0.99%
5×
0.56%
94×
7.16%
10012×
ld-2.25.so
_dl_runtime_resolve_sse_vex
0.99%
(0.00%)
5×
ld-2.25.so
do_lookup_x
0.56%
(0.38%)
94×
libc-2.25.so
mcount
7.16%
(3.05%)
10021×
0.92%
1×
4.11%
10021×
libc-2.25.so
monstartup
0.92%
(0.00%)
1×
libc-2.25.so
__mcount_internal
4.11%
(4.11%)
10021×



Grafo de chamadas gerado com callgrind

```
callgrind_annotate --auto=yes callgrind.out.<PID>
```

Geração de report com número de instruções por linha de código fonte

Analizamos o report. Muitas funções secundárias possuem algumas centenas de instruções executadas durante os testes, mas a função principal consumia milhares ou milhões de instruções:

```
-- line 714 ----------------------------------------
         .
         .  /*
         .   * AES-ECB block encryption
         .   */
         .  #if !defined(MBEDTLS_AES_ENCRYPT_ALT)
         .  void mbedtls_aes_encrypt( mbedtls_aes_context *ctx,
         .                            const unsigned char input[16],
         .                            unsigned char output[16] )
    70,084  {
         .      int i;
         .      uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;
         .
         .      RK = ctx->buf; /* ctx->rk; */
```

```
        .
   20,024       GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
   30,036       GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
   20,024       GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
   40,048       GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;
        .
   80,096       for( i = ( 10 /* ctx->nr */ >> 1 ) - 1; i > 0; i-- )
        .       {
1,922,304           AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
2,042,448           AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
        .       }
        .
  490,588       AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
        .
   40,048       X0 = *RK++ ^ \
   20,024               ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
   40,048               ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
   50,060               ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
   40,048               ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
        .
   50,060       X1 = *RK++ ^ \
   20,024               ( (uint32_t) FSb[ ( Y1       ) & 0xFF ]       ) ^
   40,048               ( (uint32_t) FSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
   50,060               ( (uint32_t) FSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
   40,048               ( (uint32_t) FSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );
        .
   40,048       X2 = *RK++ ^ \
   20,024               ( (uint32_t) FSb[ ( Y2       ) & 0xFF ]       ) ^
   30,036               ( (uint32_t) FSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
   50,060               ( (uint32_t) FSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
   40,048               ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );
        .
   40,048       X3 = *RK++ ^ \
   20,024               ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]       ) ^
   40,048               ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
   40,048               ( (uint32_t) FSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
   30,036               ( (uint32_t) FSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );
        .
   80,096       PUT_UINT32_LE( X0, output,  0 );
  100,120       PUT_UINT32_LE( X1, output,  4 );
   80,096       PUT_UINT32_LE( X2, output,  8 );
   80,096       PUT_UINT32_LE( X3, output, 12 );
   60,072   }
        .   #endif /* !MBEDTLS_AES_ENCRYPT_ALT */
```

Fizemos um benchmark com uma única chamada de mbedtls_aes_setkey_enc e mbedtls_aes_encrypt para comparação dessas funções:

```c
#include "mbedtls/aes.h"

int main()
{
    const unsigned char key[16] = {'X'};
    const unsigned char input[16] = {'Y'};
    unsigned char output[16];
    mbedtls_aes_context ctx;
    mbedtls_aes_init(&ctx);
    mbedtls_aes_setkey_enc(&ctx, key);
    mbedtls_aes_encrypt(&ctx, input, output);
    return 0;
}
```

Os seguintes trechos tiveram uso alto de processamento:

Em *mbedtls_aes_setkey_enc*:

```
  .       for( i = 0; i < ( 128 /* keybits */ >> 5 ); i++ )
  .       {
 44           GET_UINT32_LE( RK[i], key, i << 2 );
  .       }
  .
  .       // switch( ctx->nr )
  .       // {
  .       //     case 10:
  .
  .               for( i = 0; i < 10; i++, RK += 4 )
  .               {
 10                   RK[4]  = RK[0] ^ RCON[i] ^
 20                   ( (uint32_t) FSb[ ( RK[3] >>  8 ) & 0xFF ]       ) ^
 46                   ( (uint32_t) FSb[ ( RK[3] >> 16 ) & 0xFF ] <<  8 ) ^
 90                   ( (uint32_t) FSb[ ( RK[3] >> 24 ) & 0xFF ] << 16 ) ^
 30                   ( (uint32_t) FSb[ ( RK[3]       ) & 0xFF ] << 24 );
  .
 28                   RK[5]  = RK[1] ^ RK[4];
 25                   RK[6]  = RK[2] ^ RK[5];
 24                   RK[7]  = RK[3] ^ RK[6];
  .               }
```

Em *mbedtls_aes_encrypt*:

```
  2       GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
  3       GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
```

```
  2       GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
  4       GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;
  .
  8       for( i = ( 10 /* ctx->nr */ >> 1 ) - 1; i > 0; i-- )
  .       {
192           AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
204           AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
  .       }
  .
 49       AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
  .
  4       X0 = *RK++ ^ \
  2               ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
  4               ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
  5               ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
  4               ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
  .
  5       X1 = *RK++ ^ \
  2               ( (uint32_t) FSb[ ( Y1       ) & 0xFF ]       ) ^
  4               ( (uint32_t) FSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
  5               ( (uint32_t) FSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
  4               ( (uint32_t) FSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );
  .
  4       X2 = *RK++ ^ \
  2               ( (uint32_t) FSb[ ( Y2       ) & 0xFF ]       ) ^
  3               ( (uint32_t) FSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
  5               ( (uint32_t) FSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
  4               ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );
  .
  4       X3 = *RK++ ^ \
  2               ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]       ) ^
  4               ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
  4               ( (uint32_t) FSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
  3               ( (uint32_t) FSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );
  .
  8       PUT_UINT32_LE( X0, output,  0 );
 10       PUT_UINT32_LE( X1, output,  4 );
  8       PUT_UINT32_LE( X2, output,  8 );
  8       PUT_UINT32_LE( X3, output, 12 );
```

2.7 Escolha de funções para implementação em hardware

Decidimos que o round de cifragem é um bom candidato a ser implementado em FPGA, pois é executado repetidamente e consome muito processamento. Para isso, as 4 Forward-Tables usadas nos primeiros 9 rounds estarão na FPGA. Decidimos que o round final, usando AddRoundKey, estará em hardware também, requerindo que a Forward-Sbox também esteja na FPGA.

A etapa de preparação de chaves poderia ser feita em software, pois consome menos processamento. No entanto, dado a Forward-Sbox já estar na FPGA, decidimos que ela será feita em hardware também,

requisitando apenas que seja adicionada a tabela de round constants em hardware (uma tabela pequena, de 10 constantes de 32 bits, sendo uma diferente usada por round do AES).

3. Conclusão

Considerando-se os resultados dos testes e do profiling realizados, podemos concluir que a primeira etapa do projeto foi realizada com sucesso.

Frisamos que o projeto pode sofrer pequenas alterações em etapas futuras caso seja identificado algum problema nas implementações das funções em hardware, ou se percebermos que parte da implementação deve ser movida para software por limite de memória da FPGA. Poderemos também investigar de implementar dois blocos de cifragem em paralelo, se a memória permitir.

A próxima etapa será a implementação das funções em SystemC e testes das mesmas no PicoBlaze utilizando-se *testbenches*.

4. Bibliografia

- MiBench. **MiBench Version 1.0**. Disponível em: http://vhosts.eecs.umich.edu/mibench/. Acesso em: 12/06/2017.
- Embarcados. **Análise de desempenho com GNU Profiler gprof**. Disponível em: https://www.embarcados.com.br/desempenho-gnu-profiler-gprof/. Acesso em: 12/06/21017.
- GNU Binutils. **GNU gprof**. Disponível em: https://sourceware.org/binutils/docs/gprof/. Acesso em: 12/06/2017.
- ARMbed. **AES source code**. Disponível em: https://tls.mbed.org/aes-source-code. Acesso em: 13/06/2017.
- Federal Information Processing Standards. **Publication 197 - Announcing the Advanced Encryption Standard (AES)**. Disponível em:http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf. Acesso em: 13/06/2017.
- National Institue of Standards and Technology. **Recommendation for Block Cipher Modes of Operation**. Disponível em: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf. Acesso em: 14/06/2017.

As imagens utilizadas estão em domínio público.