universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Shuffle-AES (S-AES)

Cybersecurity MSc: Applied Cryptography

David Gomes Morais, 93147

Novembro, 2021
DETI, Universidade de Aveiro

Introdução

Serve o presente relatório para documentar e explicar as decisões tomadas na implementação de uma versão shuffled do algoritmo de *AES-128* (apelidado de *S-AES*), semelhante ao *AES* normal mas que usa uma chave extra de *128-bits*, criado no âmbito do primeiro projeto da cadeira de Criptografia Aplicada.

A linguagem escolhida para a implementação foi C, devido a ser uma linguagem de baixo nível, tendo por isso uma melhor *performance*, assim como a possibilidade de utilizar instruções *assembly* do *AES-NI Intel* por forma a melhorar ainda mais a *performance* da implementação em comparação a bibliotecas já existentes.

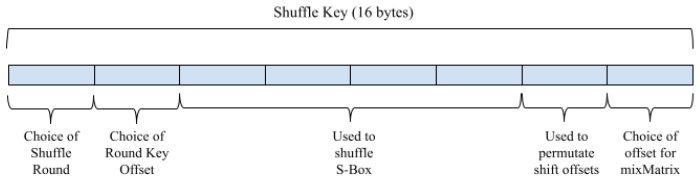
Para além do módulo de *S-AES*, foram também criados três programas: um *encrypt.c* que faz a cifra em modo *ECB* com *padding PKCS#7* de *input* a partir do *stdin*, e produz o *ciphertext* para o *stdout*; um *decrypt.c* que decifra em modo *ECB* com *PKCS#7 padding* um *input* provindo do *stdin* e produz o *plaintext* para o *stdout*; e um *speed.c*, que compara a *performance* relativa da implementação de *S-AES*, da implementação de *AES* e do *AES* da biblioteca *OpenSSL*.

Shuffle AES

Como foi mencionado na secção anterior, o *S-AES* é semelhante ao *AES* mas utiliza uma ronda modificada, escolhida a partir da *Shuffle Key* (*SK*), uma chave extra de *128-bits*. O algoritmo é composto por um método de inicialização do contexto de cifra, um método para aplicar *padding PKCS#7* ao *buffer* de *input* e métodos de cifra e decifra para cada modo de cifra.

Inicialização do Contexto

Iniciar o contexto do algoritmo pode ser feito através da função *SAES_init_ctx()*, que começa por fazer a *key expansion*, segundo os exemplos fornecidos na documentação pela *Intel* no *White Paper "AES New Instructions"* [1], que utiliza *_mm_aeskeygenassist_si128*, assim como uma função auxiliar. Esta *key expansion*, como o nome indica, expande uma chave de *16 bytes* para 11 *round keys* (sendo a primeira a própria chave), cada uma igualmente com *16 bytes*. Após isto, pode-se usar a *SK* para modificar a ronda extra da seguinte forma:



A ronda extra é escolhida com base nos primeiros dois *bytes* do *SK*, que possibilita a escolha de um inteiro entre 1 e 9, visto que a última ronda (10ª ronda) não deve ser escolhida uma vez que é diferente das restantes, tanto na cifra como na decifra.

```
314 |   // choice of shuffle round (1 to 9) based on the 2 first bytes of the SK
315 |   ctx->shuffleRoundNr = 1 + (((int) sk[0] | (sk[1] << 8)) % 9);
```

Após ser feito a expansão da chave e a seleção do *shuffle round*, é então passado a ser rodada a *round key* da ronda especial segundo um certo *offset*. Este *offset* é decidido com base no segundo e terceiro *bytes* da *Shuffle Key* e depois de aplicado o *shift* à direita do *offset* à chave, sendo que esta é guardada na *rotatedKey* do contexto do SAES.

```
317     // choice of the offset used to rotate the round key in the shuffle round, based on the
318     // 3th and 4th bytes of the SK
319     int offset = (int)(sk[2] | (sk[3] << 8))%16;
320     for (int i = 0; i < 16; i++)
321     {
322         ctx->rotatedKey[(i+offset)%16] = ctx->roundKey[16*ctx->shuffleRoundNr + i];
323         ctx->invRotatedKey[(i+offset)%16] = ctx->roundKey[16*ctx->shuffleRoundNr + i];
324     }
```

Em seguida, usam-se os seguintes 8 *bytes* da *SK* para fazer um *shuffle* da *Substitution Box* (*S-Box*) usada, através da função *shuffleSBox()* que usa uma *lookup table* da *S-Box* original, e vai trocando dois índices com base no primeiro e na *Shuffle Key*. O resultado vai ser uma *lookup table* de uma versão baralhada da *Substitution Box* original. Uma vez gerado o *shuffle* da S-Box, para poupar tempo de computação durante o *decrypt*, é feito uma pré-computação da tabela *lookup* da sua inversa, através da função *invertSBox()*. que percorre a *S-Box* e por cada valor *v* numa posição *p* da *S-Box*, coloca *p* como valor na posição *v* da *S-Box* inversa.

```
136   static void invertSBox(uint8_t* invSbox, const uint8_t* sbox)
137   {
138       uint8_t formedXored;
139       for (int p = 0; p < 256; p++)
140       {
141           formedXored = sbox[p];
142           invSbox[formedXored] = (uint8_t) p;
143       }
144
145   }
```

Os seguintes dois *bytes* da *SK* são usados para permutar o *array* [0, 1, 2, 3], que representa o *shift* que é para ser aplicado na fase de *shiftRow* da round especial, através de um gerador pseudo-aleatório onde dois índices da matriz são trocados.

```
333     // choice of the permutation of offsets to be applied in the shiftRow phase, based on the
334     // 13th and 14th bytes of the SK.
335     int shifRowByteShift[4] = {0, 1, 2, 3};
336     uint16_t seed = sk[12] | (sk[13] << 8);
337     srand((unsigned int) seed);
338     for (int i = 0; i < 4; i++)
339     {
340         int index = rand() % 4;
341         int temp = shifRowByteShift[i];
342         shifRowByteShift[i] = shifRowByteShift[index];
343         shifRowByteShift[index] = temp;
344     }
```

Por fim, são selecionados os últimos 2 *bytes* da *SK* para serem usados como *offset* a ser aplicado na matriz usada para fazer a operação de *mixColumns*, sendo que o *offset* gerado é um inteiro entre 0 e 3, sendo depois aplicado à matriz *mixColumnsMatrix* e a *invMixColumnsMatrix,* para esta última poder ser usada para fazer a operação inversa.

```
347     // choice of the offset to apply to the matrix used in the mixColumns phase of the special
348     // round, based on the 15th and 16th (last two) bytes of the SK.
349     offset = (int) (sk[14] | (sk[15] << 8)) % 4;
350     for (int i=0; i<4; i++)
351     {
352         for (int j=0; j<4; j++)
353         {
354             ctx->mixColumnMatrix[i][j] = mixColumnsMatrix[i][(j + offset)%4];
355             ctx->invMixColumnMatrix[i][j] = invMixColumnsMatrix[i][(j + offset)%4];
356         }
357     }
```

Por último, é aplicada a operação de *invMixColumns* usando a matriz inversa acabada de computar à *rotated key* que vai ser usada para fazer o *addRoundKey* na ronda especial da decifra, devido à natureza da *Equivalent Inverse Cipher* usada na decifra (abordado numa subseções posterior). É de

fazer notar que estes passos não são executados se uma *SK* não for passada à função, só sendo executada a expansão de chaves para ser usadas no fluxo do *AES* normal.

Operações de cifra

A operação de *encrypt* é feita para cada bloco de *128 bits* passado em *in*, sendo que o tamanho deve ser múltiplo de 16 *bytes* (podendo usar para isso o método *pkcs7_padder()*, que aplica *padding PKCS#7* ao *input*). Para cada bloco são então executadas as rondas de *AES* através de instruções *assembly AES-NI*, de acordo com os exemplos dados no *White Paper*. Se a *shuffle round* for para ser feita, quando chegamos à ronda especial é executado *aes_enc_shuffle_round*() seguido do *XOR* binário com a *rotated key* (este é feito usando operações *assembly* por forma a poupar um pouco de tempo de computação em comparação a uma implementação de um *addRoundKey* em C), em vez da instrução assembly *_mm_aesenc_si128()* que seria espectável.

Na operação *subBytes* da ronda especial, como uma *Shuffled Substitution Box* já foi computada, podemos simplesmente substituir cada *byte* da mensagem pelo *byte* respectivo na *Shuffled S-Box*. Isto é feito através de uma *lookup* table para a *S-Box* que, apesar de ser menos criptograficamente seguro do que fazer as operações *on the fly* devido ao facto de permitir ataques por *side-channels* (*timing* e *cache* por exemplo), é o que tem uma *performance* mais eficiente, visto os valores serem pré-computados na fase de inicialização do contexto.

No passo de *shiftRows*, com os *offsets* pré-permutados num array *shiftOffsets*, é simples iterar sobre cada *row* e fazer um *shift* à esquerda do número especificado de *bytes*, ditado pelo *array*. Esta operação tem espaço para melhorias de *performance* visto primeiro ser preciso iterar sobre os *16 bytes* por cada coluna 3 vezes, uma para sabermos que valores pertencem a uma determinada *row*, outra para fazer os *shifts* e uma última para substituir os *bytes* mudados na mensagem. Uma possível solução seria trabalhar com o estado em forma de matriz em vez de em forma de array, fazendo com que fosse mais fácil identificar que valores pertencem a que *row*.

Por fim, temos o *mixColumns*, que por cada coluna efetua a multiplicação matricial com a matriz que foi pré-calculada tendo em consideração o *offset* gerado pela *SK*. Esta multiplicação matricial, no entanto, utiliza aritmética modular, sendo que as adições são substituídas por *XORs* binários, e as multiplicações são efetuadas no *GF(2^8)* através da função *gmul()* [2].

Cada modo de cifra utiliza estes passos da mesma forma, mas tem em consideração diferentes cenários quando se trata do que fazer com o resultado e que *feedback* usar. Em *ECB*, no entanto, após ser analisada a *performance* relativa através de *speed.c*, percebeu-se que esta deixava um pouco a desejar visto estar a fazer a cifra de cada bloco de forma serializada. Como *ECB* cifra cada bloco de forma igual e não tem em consideração nenhum *feedback* de blocos anteriores ou posteriores, foi então implementado a cifra de blocos em paralelo através da instrução

```
428     #pragma omp parallel for
429     for (int i = 0; i < size; i++)
430     {
```

e compilando usando *-fopenmp*, o que permite que seja executado em paralelo o *for-loop* que itera e cifra cada bloco, aumentando assim a *performance* geral deste modo de cifra e decifra (visto na decifra acontecer o mesmo).

Operações de decifra

As operações de decifra são semelhantes às usadas na cifra, só que em vez de termos um *round i* modificado, temos dois (*r* e *r-1*, onde *r* = 10 - *i*), sendo que na round *r-1* iremos fazer o inverso de *subBytes* e de *shiftRows* com os parâmetros da ronda especial, e na ronda *r* fazemos o inverso de *mixMatrix* e de *addRoundKey* da ronda especial. As restantes operações são as que seriam feitas normalmente num *AES* normal, e isto é devido ao fluxo da decifra em comparação ao da decifra, havendo a necessidade de compensar o último bloco que é diferente.

Desta forma, na round *r-1* fazemos o *invShiftRows* com os *offsets* gerados com base na *SK*, com uma abordagem semelhante a *shiftRows*, só que em vez de subtrairmos o *offset* ao índice, somamos por forma a inverter a operação. Esta função, por ser tão semelhante, tem as mesmas limitações de *shiftRows* mencionadas na subseção acima. Ainda na round *r-1*, fazemos também o inverso da substituição de *bytes*, mas em vez de utilizarmos a versão *shuffled* da *S-Box*, utilizamos a sua inversa por forma a inverter a operação. As restantes operações desta ronda (*invMixColumns* e *addRoundKey*), como já foi mencionado, são feitas como seria de esperar no *flow* normal do *AES*. É de fazer notar que para, mais uma vez se melhorar ligeiramente a *performance*, aproveitou-se o facto de o *AES-NI* ter instruções em *assembly* para fazer estas operações - *_mm_xor_si128()* para o *addRoundKey* e *_mm_aesimc_si128()* para o *invMixColumns* -, sendo que esta última normalmente é usada para ajudar a preparar as round keys para o processo de decifra, mas como a única operação que efetua é inverter a *mixColumns* "normal" do AES, pode ser usada neste contexto).

Em relação à ronda *r*, as operações de *invShiftRows* e *invSubBytes* são efetuadas normalmente, usando os *offsets* a aplicar às *rows* e a *S-Box* original (respetivamente) enquanto que o *invMixColumns* é feito através da função *mixColumns*(), mas é-lhe passada a matriz inversa que foi previamente calculada, por forma a poder inverter a operação. Inicialmente tinha sido empregue a estratégia de fazer a *invMixColumns* através de três chamadas consecutivas a *mixColumns()*, visto que:

$$M^4 = I \quad \Rightarrow \quad M^3 = M^{-1},$$

no entanto esta estratégia provou-se infrutífera devido ao facto de que danificava demasiado a *performance* do algoritmo, optando-se por isso por pré-calcular a matriz de inversão com base nos *offsets* na inicialização do contexto.

Por último, é efetuado o *addRoundKey* com a *rotatedKey* invertida para a decifra. Esta *round key* invertida é feita devido à natureza da *Equivalent Inverse Cipher*, que permite trocarmos as ordens das operações de um bloco de *AES* normal, mas como passamos o addRoundKey para depois da operação de inversão de *mixColumns*, precisamos também de aplicar a *invMixColumns* à *round key* para que a inversão (e consequentemente a decifra) possa ser feita de forma correta.

Aplicações *encrypt*, *decrypt* e *speed*

Como foi mencionado, foram também feitas aplicações para a cifra e decifra em *ECB*, com *padding*, que lêem dados a partir do *stdin*, dando o output para o *stdout*, em que se exemplifica a utilização do algoritmo. Em ambos os casos, é gerada uma *Key* a partir de uma *password* passada como primeiro argumento, através do *PBKDF2*, e uma *Shuffle Key* gerada da mesma maneira a partir de uma

password passada como segundo argumento. Se apenas um argumento for passado, então efetua *AES* normal, isto é, sem *Shuffled Round*. O *padding* é feito através da função *pkcs7_padder()* [3], que enche o último bloco com lixo e retorna o número de *bytes* adicionados na cifra. Para fazer o *unpad* na decifra, basta ler o último *byte*, que representa o número de *pads* adicionados, e ler o *buffer* até *lineSize - pad*, para ignorar os *bytes* de *padding*.

Na aplicação de *speed*, é usado uma página de memória cheia de caracteres aleatórios gerados a partir de */dev/urandom*, e estes são cifrados e decifrados em modo *ECB* usando vários algoritmos, efetuando medições do tempo que cada um demora. Estas medições são apenas relativas às operações de cifra e decifra em si, foram efetuadas 1 000 000 medições e escolhida a menor, e estão a ser comparados o *S-AES* implementado, o *AES* implementado (isto é, o *S-AES* em que não é passada uma *SK*) e o *AES* do *OpenSSL* com precisão aos nanosegundos. Os resultados são os seguintes:

Algoritm	Encryption time (ns)	Decryption time (ns)
SAES	113218.000	116153.000
AES	19316.000	21421.000
OpenSSL AES	757.000	760.000

Conclusão

Foi implementada uma versão do *AES* relativamente boa, em comparação com o *OpenSSL*, visto que foram empregues instruções *AES-NI*. A *performance* do *S-AES* deixa um pouco a desejar comparado com o *AES* sendo que, como foi mencionado anteriormente, existem algumas melhoras a nível de *performance* que poderiam ser feitos, especialmente na fase de *shiftRows*, mesmo sendo usado (quando possível) instruções *assembly* para não se notar um impacto tão grande na *performance*.

Em termos de código poderia ter-se calculado a *S-Box* na fase de inicialização de contexto em vez de usar a *lookup* table da mesma *hardcoded* no início, de forma a deixar o código mais limpo. Por fim, limitações do projeto incluem, como foi mencionado, algumas falhas de segurança no que toca à possibilidade de análise via *side channels* (que foram ignoradas porque o objetivo do projeto recai sobre *performance*), a *performance* em comparação com bibliotecas tais como o *OpenSSL*, e o facto de o *S-AES* implementado só suportar os modos de cifra *ECB* e *CBC*, sendo que não deveria ser muito difícil a implementação dos restantes.

Referências

[1] Shay Gueron, “Intel® Advanced Encryption Standard (AES) New Instructions Set“ white paper, Intel (online), https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[2] fgrieu, “Constant time multiplication in GF(2^8)”, crypto.stackexchange.com (online), https://crypto.stackexchange.com/questions/82095/constant-time-multiplication-in-gf28

[3] bonybrown, “pkcs7_padding.c”, Github repository (online), https://github.com/bonybrown/tiny-AES128-C/blob/master/pkcs7_padding.c