Implementação eficiente e segura de cifras de fluxo em software

Daniel P. Cunha[1], **Rafael J. Cruz**[1], **Diego F. Aranha**[1]

[1]Instituto de Computação (IC) – Universidade Estadual de Campinas (Unicamp)

{dcunha,raju,dfaranha}@lasca.ic.unicamp.br

Abstract. *The number of devices able to connect to the Internet reaches new levels day after day, raising challenges about protecting the information they are capable of transmitting. Because of their smaller footprint, stream ciphers represent an ideal cryptographic primitive for confidentiality of in-transit data. This work presents efficient and constant time software implementations of two stream ciphers targeted at ARM processors. Our implementations of ChaCha20 and SOSEMANUK provide performance gains up to 30% and 43% against reference code, respectively, depending on the testing platform used.*

Resumo. *O crescente número de dispositivos que podem se conectar à Internet atinge novos patamares e traz consigo desafios ainda maiores quanto à segurança das informações transmitidas. Cifras de fluxo são uma primitiva criptográfica ideal para fornecer confidencialidade nesse cenário. Este trabalho apresenta implementações eficientes em software de duas cifras de fluxo na plataforma ARM, protegidas contra ataques de canal lateral de tempo. A implementações do ChaCha20 e SOSEMANUK apresentaram ganhos de desempenho de até 30% e 43% em relação ao código de referência, respectivamente, a depender da plataforma de testes utilizada.*

1. Introdução

Avanços tecnológicos permitem a progressiva inclusão de microprocessadores em praticamente qualquer objeto. Tal fato abre caminho para um cenário que representa uma nova era das comunicações, à medida que o número de dispositivos com capacidade de transmissão de dados aumenta rapidamente, a exemplo de eletrodomésticos e objetos de uso pessoal. Em linhas gerais, o cenário, batizado de "Internet das Coisas" ou *Internet of Things* – IoT, traz consigo a crescente necessidade de obtenção de ferramentas capazes de funcionar neste novo contexto. Desta forma, a proteção das informações pessoais coletadas por utensílios do cotidiano demanda a implementação eficiente de primitivas criptográficas. Alcançar esse objetivo pode ser desafiador, dado que a tendência de redução da recursos consumidos pelos dispositivos implica poder computacional restrito dos mesmos, especialmente aqueles alimentados por baterias.

O surgimento da criptografia de chave pública fomentou ativamente as pesquisas em implementação eficiente de primitivas criptográficas, onde a princípio o desempenho era essencialmente avaliado em termos do tempo de execução. O novo

 ©2016 SBC — Soc. Bras. de Computação

paradigma de IoT reforça e expande requisitos de desempenho para incluir consumo de energia e motiva a necessidade de algoritmos leves de criptografia.

Neste trabalho foram propostas otimizações das implementações de referência de dois cifradores de fluxo: o ChaCha [Bernstein 2005a], que é baseado no algoritmo do Salsa20 [Bernstein 2008], e o SOSEMANUK [Berbain et al. 2008] que faz uso das funções de transformação linear baseadas no SERPENT [Biham et al. 1998] e princípios de projeto do cifrador SNOW 2.0 [Ekdahl and Johansson 2002]. Tanto o ChaCha quanto o SOSEMANUK foram finalistas da competição internacional eSTREAM, que buscou no Perfil 1 cifradores de fluxo com características mais apropriadas para aplicações em software, com requisitos de alta vazão de dados. A implementação do ChaCha é intrinsecamente protegida contra ataque de canal lateral de tempo, mas o SOSEMANUK possui duas variantes: uma em tempo variável e outra em tempo de execução constante. A existência de 2 variantes para o SOSEMANUK advém de características do próprio cifrador, que faz uso de uma tabela de 256 posições, conforme especificação [Berbain et al. 2008], que pode introduzir variações na latência de resposta da hierarquia de memória úteis para um adversário capaz de montar ataques de canal lateral de tempo.

2. Notação e terminologia

A Tabela 1 estabelece o significado dos símbolos utilizados neste documento.

Tabela 1. Símbolos e seu respectivo significado neste documento.

Símbolo	Significado
$\lll n$	Rotação de n bits à esquerda
$\oplus$ ou $\wedge$	`XOR` bit a bit
$+$ ou $\boxplus$	Adição módulo 32 bits
$\ll n$	Deslocamento de n bits à esquerda
$\gg n$	Deslocamento de n bits à direita

Dada a escolha de plataforma alvo, o termo "palavra de memória", ou simplesmente "palavra" se refere a um elemento de 32 bits. Sempre que mencionado implementação de referência deve-se entender que a implementação em questão refere-se àquela encaminhada como última submissão à competição eSTREAM. No caso do ChaCha20, ver Salsa20 [eSTREAM 2004].

3. O algoritmo ChaCha20

O ChaCha20 [Bernstein 2005a] é uma variante da cifra de fluxo Salsa20, e esta por sua vez é construída sobre uma função pseudo-aleatória definida pelo autor como uma função de *hash* (ou confusão) baseada em operações ARX (*Add-Rotate-XOR*), com adição realizada em 32 bits. Mais especificamente, a definição do Chacha20 segue todas as especificações do Salsa20 a menos da função quarto-de-rodada, ou "*quarter-round*", e do posicionamento de cada elemento da matriz de estado, que consiste em 16 palavras de 4 bytes. Na matriz, metade das palavras representam a chave secreta, 4 palavras representam constantes e as últimas 4 são palavras de entrada sob controle do usuário do cifrador, representando o *nonce* ou vetor de

 ©2016 SBC — Soc. Bras. de Computação

inicialização. O número 20 que sucede o nome do algoritmo representa o número de iterações ou rodadas da função *hash*.

3.1. A função *quarter-round*

A função *quarter-round* (QR) é definida no Salsa20 considerando uma entrada constituída por uma sequência de 4 palavras de 32 bits. Para cada 4 ciclos finalizados desta função completa-se o equivalente a uma rodada. A *quarter-round* deve processar a entrada segundo a sequência de operações de 32 bits como no Algoritmo 1 apresentado a seguir para o Salsa20 e reprojetada para o ChaCha20 como no Algoritmo 2. Ambas as funções apresentam o mesmo número de operações de soma, XOR bit-a-bit e rotações.

Algoritmo 1 Função QR para o Salsa20

1: $b \leftarrow b \oplus (a + d) \lll 7$
2: $c \leftarrow c \oplus (b + a) \lll 9$
3: $d \leftarrow d \oplus (c + b) \lll 13$
4: $a \leftarrow a \oplus (d + c) \lll 18$

Algoritmo 2 Função QR para o Chacha20

1: $a \leftarrow a + b$, $d \leftarrow (d \oplus a) \lll 16$
2: $c \leftarrow c + d$, $b \leftarrow (b \oplus c) \lll 12$
3: $a \leftarrow a + b$, $d \leftarrow (d \oplus a) \lll 8$
4: $c \leftarrow c + d$, $b \leftarrow (b \oplus c) \lll 7$

A essência da cifração dos dados pode ser resumida em 3 funções: a função de *hash*, *quarter-round*, a função *row-round* e a função *column-round*. As duas últimas buscam selecionar os 16 elementos da matriz de estados para operação. A operação das duas funções ocorre de modo alternado entre linhas e colunas da matriz, de forma que a função *column-round* é composta de quatro chamadas à função *quarter-round*, cujos 4 parâmetros passados nesta chamada correspondem à posições na matriz de estados.

Analogamente, a função *row-round* realiza 4 chamadas à função *quarter-round*, onde os parâmetros agora correspondem a parâmetros selecionados um de cada linha da matriz de estados de modo que em cada chamada um único elemento de cada linha é utilizado. A aplicação de uma função *column-round* seguida de uma *row-round* é batizada de um *double-round*, que equivale a 2 rodadas do total de 20 rodadas propostas (uma rodada por função chamada).

Cabe mencionar que existem variações da implementação original, Salsa20/12, Salsa20/8 que correspondem a algoritmos que implementam, respectivamente, 12 e 8 rodadas da função, de forma que tais algoritmos representam duas dentre outras variantes com menor número de rodadas relativas ao ChaCha20. O ChaCha20/8 é também conhecido como ChaCha8, o ChaCha20/12 como ChaCha12 e assim por diante.

3.2. A matriz de estados do ChaCha20

A matriz do ChaCha20 possui 16 palavras de 32 bits ou um total de 64 bytes, dispostas conforme a Figura 1, que apresenta as mudanças propostas na matriz do Chacha20 em relação à matriz do Salsa20.

Após executadas as 20 rodadas da função de *hash* sobre a matriz de estados do ChaCha20 tais como descritas anteriormente, adiciona-se o resultado à uma cópia inalterada da matriz original, byte a byte, para compor os 64 bytes de

 ©2016 SBC — Soc. Bras. de Computação

$$\begin{pmatrix} constant & constant & constant & constant \\ key & key & key & key \\ key & key & key & key \\ input & input & input & input \end{pmatrix} \qquad \begin{pmatrix} constant & key & key & key \\ key & constant & input & input \\ input & input & constant & key \\ key & key & key & constant \end{pmatrix}$$

(a) Matriz de estados ChaCha20 **(b) Matriz de estados Salsa20**

Figura 1. Matriz de estados dos algoritmos ChaCha20 e Salsa20

cadeia de chave, que por sua vez será combinada através de um `XOR` sobre a mensagem original para produzir 64 bytes da mensagem cifrada. A especificação do Salsa20 [Bernstein 2005b] requer que os bytes de interesse sejam processados no formato *little-endian*. Dessa forma ao definir $Z_i = (z_0, z_1, \cdots, z_{15})$ como as 16 palavras da cadeia de chave, tem-se que o texto claro, (M_j), é transformado em texto cifrado, (C_j), pelo Salsa20 após conversão de representação por $C_j = M_j \oplus Z_j$, com $Z_j = (Y_i + X_i)$, onde $X_i = (x_0, x_1, \cdots, x_{15})$ são os elementos da matriz de estados original, da qual é guardada uma cópia antes da execução da função de *hash* e $Y_i = (y_0, y_1, \cdots, y_{15})$ representa a matriz de estados após as 20 rodadas da função de *hash*.

4. Descrição da implementação do ChaCha20

A implementação proposta foi desenvolvida em linguagem C, tal como a implementação de referência, fornecendo portabilidade e eficiência. A proposta de implementação leva em consideração a legibilidade do código, bem como a busca por operações ou combinações das mesmas que, mantendo as propriedades do cifrador, resultem em tempo de execução reduzido.

Aliado ao foco em uma implementação mais eficiente buscou-se uma implementação regular do ponto de vista do número de operações realizadas em cada um dos possíveis casos de execução, isto é, reescrever execuções condicionais de modo a torná-las equivalentes no que tange ao número de instruções executadas ao tomar ou não um determinado desvio condicional.

Serão apresentadas a princípio as estratégias adotadas que contribuíram com o ganho de desempenho do algoritmo, e em seguida as modificações que tornaram o código protegido contra ataques de canal lateral, isto é, que permitiram a execução do mesmo ocorrer em tempo constante para mensagens do mesmo tamanho.

O primeiro ganho obtido decorreu da observação de que as arquiteturas dos sistemas atuais é em sua grande maioria construída sobre arquiteturas nativamente *little-endian*, fato este que tende a se consolidar dada a crescente difusão de processadores desenvolvidos sobre as arquiteturas ARM e x86. Assim é possível evitar funções de conversão das entradas e resultado.

A Figura 2 mostra a aplicação das 20 rodadas da função de hash tanto na implementação proposta sob a nova estratégia quanto na implementação de referência. Vale ressaltar que na implementação de referência, além de computar a conversão, a função especificada(`U32TO8_LITTLE`) converte elementos de 4 bytes em 4 elementos de um byte.

Adicionalmente, a segunda estratégia adotada na implementação proposta

 ©2016 SBC — Soc. Bras. de Computação

```
static inline void salsa20_hash_chacha(
                        uint8_t vec[64],
                        uint32_t matrix[16]) {
  int i;
  uint32_t x[16] = { 0 };
  uint32_t sum;
  uint32_t *vec_32 = (uint32_t *)vec;

  for (i = 0; i < 16; i++)
    x[i] = matrix[i];

  for (i = 0; i < 10; i++)
    DOUBLE_ROUND(x);

  for (i = 0; i < 16; i++) {
    sum = x[i] + matrix[i];
    vec_32[i] = sum;
  }
}
```

(a) Chacha20 *hash* otimizado

```
static void salsa20_hash_chacha(
                uint8_t output[64],
                const uint32_t input[16]){
                uint32_t x[16];
  int i;

  for (i = 0;i < 16;++i)
    x[i] = input[i];

  for (i = 20;i > 0;i -= 2) {
    QUARTERROUND( 0, 4, 8,12)
    QUARTERROUND( 1, 5, 9,13)
    QUARTERROUND( 2, 6,10,14)
    QUARTERROUND( 3, 7,11,15)
    QUARTERROUND( 0, 5,10,15)
    QUARTERROUND( 1, 6,11,12)
    QUARTERROUND( 2, 7, 8,13)
    QUARTERROUND( 3, 4, 9,14)
  }

  for (i = 0;i < 16;++i)
    x[i] = PLUS(x[i],input[i]);
  for (i = 0;i < 16;++i)
    U32TO8_LITTLE(output + 4 * i,x[i]);
}
```

(b) Chacha20 *hash* referência

Figura 2. Comparação da implementação sob a nova estratégia com a implementação de referência.

refere-se ao modo como são realizadas as conversões das palavras (4 bytes) em elementos de um único byte. Ao contrário da implementação de referência que, a partir de uma palavra, executa deslocamentos, bitwise `AND`'s e `OR`'s para extração de cada um dos bytes de interesse, é proposta a inicialização de apontador de dimensão 32 bits (8 bits) por meio da conversão em um elemento de 8 bits (32 bits) de interesse caso a conversão seja de elementos de 1 byte (4 bytes) para elemento(s) de 4 (1) byte(s).

A segunda estratégia é apresentada na Figura 2, onde apesar de existir um consumo relativo maior de memória pela inicialização de 2 variáveis adicionais (`sum` e `*vec_32`), há o compromisso com o número de operações que é significativamente menor, visto que o parâmetro `vec` da função de *hash* pode, após aplicada a referida estratégia da inicialização dos auxiliares, ser visto como um vetor de 4 bytes, aproveitando melhor os acessos à memória realizados.

Ainda que esta estratégia tenha beneficiado as funções de expansão de chaves e inicialização, é na função de *hash* do algoritmo que se verifica o maior aproveitamento do uso destas estratégias, visto que as funções de inicialização podem ser utilizadas uma única vez para cada 2^{70} bytes a cifrar (período do cifrador), enquanto a função de *hash* pode ser utilizada sucessivamente a cada múltiplo de 64 bytes a cifrar, ainda sob o mesmo estado de inicialização caso respeitado o período mencionado. Por este motivo, as estratégias apresentadas, embora utilizadas nas funções de inicialização, não foram exibidas em termos de código implementado nas funções de expansão.

 ©2016 SBC — Soc. Bras. de Computação

5. O algoritmo SOSEMANUK

O SOSEMANUK é uma cifra de fluxo construída com base na estrutura geral do SNOW 2.0 [Ekdahl and Johansson 2002], baseado em um registrador de deslocamento de retroalimentação linear, ou LFSR – *Linear Feedback Shift Register* – e em uma máquina de estados finitos ou FSM – *Finite State Machine*. Adicionalmente, o SOSEMANUK agrega a transformação linear do cifrador de bloco SERPENT [Biham et al. 1998], candidato do concurso AES, para a inicialização do estado interno deste cifrador. A expansão de chaves também se baseia no SERPENT. Diferentemente do ChaCha que aceita chaves de 128 ou de 256 bits, o SOSEMANUK aceita chaves de tamanhos de 1 a 256 bits, porém chaves maiores do que 128 bits são recomendadas por razões de segurança. Para a expansão da chave, o SOSEMANUK faz uso de 8 S-Boxes – *Substitution Boxes* ou caixas de substituição – herdadas do SERPENT. O objetivo do SOSEMANUK é superar o SNOW 2.0 em termos de segurança e eficiência.

5.1. A inicialização do estado interno: populando o LFSR e a FSM

A inicialização do SOSEMANUK consiste em expandir a chave secreta. Antes da expansão, é realizado o preenchimento da chave para que esta tenha 256 bits, de modo a completar chaves menores do que 256 bits com um 1 no primeiro bit não utilizado da chave fornecida e os demais bits com 0 até atingir os 256 restantes. Em seguida, a expansão é realizada de modo a gerar 100 elementos de 4 bytes, ou 100 sub-chaves, pela interpolação em sequência especificada por uso parcial da sequência de caixas de substituição definidas no SERPENT [Biham et al. 1998].

A população inicial da FSM e do LFSR compõe a etapa seguinte da inicialização do algoritmo. A inicialização transforma as sub-chaves geradas anteriormente por meio do uso das S-Boxes associadas à subsequente transformação linear do SERPENT em todas as sub-chaves. As S-Boxes operam a chave expandida no modo *bitsliced*. Vale mencionar que em pontos específicos desta inicialização é que são populadas individualmente as 10 posições da LFSR e os 2 registradores da FSM, segundo especificação do cifrador [Berbain et al. 2008].

Este trabalho dará maior ênfase sobre a definição do LFSR, pelo fato de trazer contribuições significativas nessa estrutura do cifrador a fim de proteger o algoritmo contra ataques de canal lateral. A resistência é aprimorada pela eliminação do uso das tabelas internas do SOSEMANUK, ao calcular seus coeficientes durante a execução e apenas quando forem necessários. Além disso, outra contribuição nesse sentido foi o uso de estratégias na implementação estrutural da FSM no SOSEMANUK, para que instruções sujeitas a desvios condicionais sejam evitadas ou reformuladas para atingir execução em tempo constante.

5.2. A construção do LFSR: os polinômios base

A estrutura geral do LFSR é representada na Figura 3, onde o novo elemento do registrador de deslocamento, s_{t+9}, é gerado a partir de operações sobre α e α^{-1} e o valor que é deslocado para fora é utilizado para calcular com um `XOR` a saída do algoritmo.

Antes de mencionar o papel do coeficiente α e seu inverso, é necessário mencionar que os valores destes coeficientes são obtidos por operações envolvendo o

 ©2016 SBC — Soc. Bras. de Computação

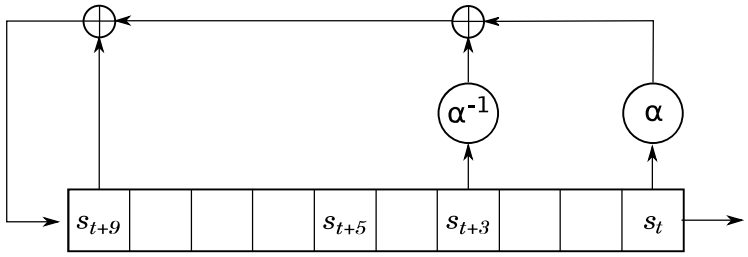

Figura 3. Esquema do LFSR

polinômio primitivo $Q(X) = X^8 + X^7 + X^5 + X^3 + 1$, que define um corpo binário de elementos em $GF(2)$ (os coeficientes de cada termo de $Q(X)$ são elementos iguais a 1 ou 0) cujas raízes, $\beta_i = (\beta^7, \beta^6, \cdots, \beta, 1)$, produzem os elementos inversíveis do corpo de extensão, representado por um segundo polinômio, $P(X) = X^4 + \beta^{23}X^3 + \beta^2 45X^2 + \beta^{48}X + \beta^{239}$. Esse último é composto por elementos em $GF(2^8)$ (coeficientes de $P(X)$ são bytes de valor entre 0 e 255) de modo que as raízes de $P(X)$ são representadas pela base $(\alpha_3, \alpha_2, \alpha, 1)$, e cada elemento representado por $P(X)$ possui na realidade um correspondente de 32 bits na representação de números inteiros, isto é, cada elemento representado por $P(X)$ é um elemento de 32 bits que irá compor o próximo elemento de s_{t+9} no LFSR.

A definição destes 2 corpos é mencionada com mais detalhes na especificação do algoritmo [Berbain et al. 2008], onde é definido que uma multiplicação por β corresponde a um deslocamento para esquerda de um bit na representação de inteiros, seguido de um `XOR` com uma máscara fixa se o bit mais significativo eliminado pelo deslocamento for 1. Vale ressaltar que a adição de elementos em $GF(2^{32})$ corresponde a um `XOR`. A multiplicação de um elemento em $GF(2^{32})$ por α equivale a um deslocamento à esquerda de 8 bits seguido de um `XOR` com uma máscara de 32 bits que depende apenas do byte mais significativo do elemento em $GF(2^{32})$. Uma divisão por α (multiplicação por α^{-1}) representa um deslocamento à direita de 8 bits seguido de um `XOR` com uma máscara que depende apenas do byte menos significativo de s_{t+3} em LFSR.

5.3. A atualização da FSM durante o processo de cifração

A FSM é atualizada, após ter sido inicializada, por meio de operações sobre $R1_{t-1}$ e $R2_{t-1}$, onde $t = 0$ representa o momento da inicialização, de forma que t é sempre maior ou igual a 1. O cálculo de $R1_t$ é como segue: $R1_t = (R2_{t-1} + mux(lsb(R1_{t-1}), s_{t+1}, s_{t+1} \oplus s_{t+8}))$ mod 2^{32}, sendo $lsb(x)$ o bit menos significativo de x e $mux(w, y, z)$ igual a y caso w seja igual a 0 ou igual a z caso w seja igual a 1. O cálculo de $R2_t$ é dado por $(M \cdot R1_{t-1} \mod 2^{32}) <<< 7$, onde M é constante e igual a `0x54655307`. Finalmente, a saída da FSM é dada por $f_t = (s_{t+9} + R1_t \mod 2^{32}) \oplus R2_t$.

6. Descrição da implementação do SOSEMANUK

A implementação protegida consiste, além da implementação em tempo constante de modo análogo ao descrito ao final da seção 4, no cálculo dos coeficientes das tabelas `mulAlpha` e `divAlpha` sob demanda, ou seja, conforme a execução do algoritmo necessitar destes coeficientes. Estas tabelas representam as máscaras fixas

 ©2016 SBC — Soc. Bras. de Computação

mencionadas ao final da seção 5.2, que são operadas por um `XOR` com o resultado do deslocamento de 8 bytes para a direita no elemento s_{t+3} do LFSR ou de 8 bytes para a esquerda no elemento s_t de LFSR.

A versão desprotegida contra ataques de canal lateral apresenta um compromisso com a diminuição no tempo de execução a custo de maior uso de memória interna associado ao estado do cifrador, visto que duas tabelas de 256 posições de 32 bits contendo as máscaras podem ser armazenadas localmente no algoritmo ou calculadas sob demanda, para as implementações desprotegidas ou protegidas, respectivamente. O cálculo das máscaras ocasiona, naturalmente, aumento bastante significativo no tempo de execução geral.

A Figura 4 apresenta o corpo de 2 das 3 funções utilizadas para o cálculo das máscaras (elementos das tabelas `mulAlpha` e `divAlpha`, que correspondem à multiplicação e divisão por alfa, respectivamente). A função `mul_galois8` opera no corpo binário, enquanto a `(mul/div)alpha_galois32` opera no corpo $GF(2^8)$ e concatena os resultados para obter elementos em $GF(2^{32})$, considerando o *carry*. A função `divAlpha_galois32`, não exibida na imagem, é análoga à função apresentada na função da Figura 4b, a menos das constantes passadas como primeiro parâmetro para a função `mul_galois8` e a menos da constante de verificação do byte menos significativo ao invés do mais significativo, que por sua vez implementa um simples `AND` lógico da constante `0xFF` com o valor b. Ainda na `divAlpha_galois32`, as constantes são: `0xCD`, `0x40`, `0x0F` e`0x18`, nessa ordem.

```
uint8_t mul_galois8(
            uint8_t a,
            uint8_t b,
            uint8_t *of) {
  uint8_t p = 0;
  int32_t i;

  for (i = 0; i < 8; i++) {
    // multiply  b_(i) * a
    p = p ^ ((b & 0x01) * a);
    /keeps every overflow bit before a *= 2
    *of |= ((a & 0x80) << (7 - i));
    // multiply a by 2
    a = (a << 1) ^ (((a & 0x80) >> 7) * 0xA9);
    // cat the  b_(i+1)
    b = (b >> 1);
  }

  b ^= *of;
  return p;
}
```

(a) Multiplicação no corpo $GF(2^8)$ operando sobre $Q(x)$.

```
static inline uint32_t mulAlpha_galois32(
                            uint32_t b,
                            uint8_t *overflow) {

  uint8_t p = ((b & 0xFF000000) >> 24);
  int32_t ans = 0;

  ans = ans | ((*overflow ^
        mul_galois8(0x13, p, overflow)) << 0);
  *overflow = 0;
  ans = ans | ((*overflow ^
        mul_galois8(0xCF, p, overflow)) << 8);
  *overflow = 0;
  ans = ans | ((*overflow ^
        mul_galois8(0x9F, p, overflow)) << 16);
  *overflow = 0;
  ans = ans | ((*overflow ^
        mul_galois8(0xE1, p, overflow)) << 24);
  *overflow = 0;

  return ans;
}
```

(b) Multiplicação no corpo $GF(2^{32})$ operando sobre $P(x)$.

Figura 4. Implementação das funções para cálculo dos coeficientes da tabela `mulAlpha`.

Para evitar execuções em tempo variável, foi adotada a definição da operação de seleção(mux) de acordo com uma implementação opcional sugerida na implementação de referência, que evita o desvio condicional que frequentemente recai em números distintos de operações executadas ou variância no tempo por interferência

 ©2016 SBC — Soc. Bras. de Computação

da predição de desvios. A implementação adotada é:

```
#define XMUX(c, x, y) (((signed int) ((c) << 31) >> 31) & (y)) ^ (x))
```

A macro seleciona entre os valores x ou $x \oplus y$, de forma que todas as operações serão executadas incondicionalmente, e será selecionado o resultado apropriado, a depender do valor de c.

As modificações realizadas na função de expansão da chave consistem em evitar o uso das funções `memcpy` e `memset` no momento em que se efetua o preenchimento da chave fornecida para 256 bits, visto que o tempo de execução dessas funções pode ser variável em alguns casos [Morrow 2004]. Por fim, foram realizadas alterações nos contextos das funções de inicialização da FSM e do LFSR para realizar conversões analogamente à estratégia adotada no ChaCha20: operar simultaneamente em bytes individuais armazenados em uma mesma palavra quando possível, para propiciar considerável economia de acessos à memória ou ainda evitar rotinas de conversão de tipos.

7. Resultados Experimentais

Esta seção apresenta uma comparação de desempenho entre implementações propostas para ambos os cifradores apresentados neste trabalho e as respectivas implementações de referência. Os testes foram realizados em diferentes plataformas ARM, com o propósito de medir, em ciclos por byte cifrado, os tempos de execução em cada plataforma, conforme Tabela 2.

O *benchmark* de nossas implementações foi realizado da seguinte forma nas respectivas plataformas:

- **Cortex-M3:** Arduino Due com processador Atmel SAM3X8E ARM Cortex-M3 84MHz. O compilador fornecido em última versão pelo Kit de Desenvolvimento Arduino, GCC 4.8.4, foi utilizado com as flags `-fno-schedule-insns-nostdlib -mcpu=cortex-m3 -mthumb`. O tempo de execução foi medido pela conversão da saída da função *micros()* do Arduino que mede ciclos em micro segundos através da simples multiplicação da frequência nominal.
- **Cortex-M4:** Teensy 3.2 é uma placa que contém o processador MK20DX256VLH7 Cortex-M4 de 72MHz. Utilizamos o mesmo compilador da plataforma Cortex-M3, porém com as flags `-fno-schedule-insns -nostdlib -mcpu=cortex-m4 -mthumb`. O tempo de execução foi medido através de um registrador para contagem de ciclos nativo à plataforma e do uso de um trecho de código em *assembly*.
- **Cortex-A7/15:** ODROID-XU4 é uma placa equipada com um Samsung Exynos5422 Cortex-A15 e um Cortex-A7 2GHz de oito núcleos. Instalamos a distribuição oficial do Arch Linux para a placa, que está equipada com o GCC 6.1.1 para plataformas ARM, usando as flags `-fno-schedule-insns -march=native`. O tempo de execução foi medido ao habilitar a leitura do registrador para contagem de ciclos(CCNT - Cycle CouNT register) no módulo de monitoramento de desempenho(Performance Monitor Unit - PMU) a nível de usuário.

 ©2016 SBC — Soc. Bras. de Computação

- **Cortex-A8:** BeagleBone Black é uma placa equipada com um AM335x 1GHz ARM Cortex-A8. Mesmas configurações de Sistema Operacional, compilador, flags e ativação do contador de ciclos pela PMU da ODROID-XU4.
- **Cortex-A53:** ODROID-C2 é uma placa contendo um processador Amlogic ARM Cortex- A53(ARMv8) com quadro núcleos de 2GHz. Mesmas configurações de Sistema Operacional, compilador, flags e ativação do contador de ciclos pela PMU da ODROID-XU4.
- **Core i7 Ivy Bridge:** Intel Core i7-3632Q de clock 2.20GHz. O GCC 6.1.1 foi utilizado novamente e com as flags `-fno-schedule-insns -march=native`. O registrador RDTSC foi utilizado para contagem de ciclos.

Para o SOSEMANUK habilitamos também a flag `-O3` em todas as plataformas, já para o ChaCha20 habilitamos apenas o parâmetro `-O1` pois demonstrou melhor custo benefício.

Tabela 2. Tempo de execução e tamanho de código para ChaCha20 e SOSEMANUK no Cortex-M3/M4/A7/A8/A15/A53 ARM e Core i7 x86 Ivy Bridge. As células apresentam o tempo médio para cifrar um único byte (CPB) para entradas de 128 bytes e 4096 bytes, com TC sendo a implementação em tempo constante. A letra N representa a implementação proposta nesse trabalho e R a implementação de referência.

		ChaCha20			SOSEMANUK			SOSEMANUK TC		
		Entrada de 128 bytes (CPB)	Entrada de 4096 bytes (CPB)	Tamanho (bytes)	Entrada de 128 bytes (CPB)	Entrada de 4096 bytes (CPB)	Tamanho (bytes)	Entrada de 128 bytes (CPB)	Entrada de 4096 bytes (CPB)	Tamanho (bytes)
Cortex - M3	N	**49,84**	**49,84**	1348	65,95	24,57	7696	212,30	168,07	13761
	R	**71,95**	**71,97**	1152	65,74	28,01	17428	–	–	–
Cortex - M4	N	36,56	34,96	1348	45,37	15,37	7836	153,47	123,42	14061
	R	47,53	44,98	1152	45,24	17,97	17856	–	–	–
Cortex-A7	N	47,77	30,08	1896	**37,09**	10,25	8760	133,80	92,65	16097
	R	51,31	38,96	1736	**51,49**	12,35	19251	–	–	–
Cortex-A8	N	24,72	23,80	1896	24,87	6,46	8760	71,66	52,66	16097
	R	31,49	31,01	1736	26,33	8,31	19251	–	–	–
Cortex - A15	N	14,96	14,64	1896	17,60	**4,00**	8760	56,79	43,29	16097
	R	19,00	18,68	1736	18,53	**7,02**	19251	–	–	–
Cortex - A53	N	24,54	24,02	1552	17,34	3,93	8504	51,59	37,98	12883
	R	31,95	30,54	1548	17,78	6,58	19779	–	–	–
i7 Ivy Bridge	N	9,37	8,88	1602	21,33	6,28	8716	67,38	50,62	15121
	R	12,54	11,93	1235	21,91	7,13	19160	–	–	–

Cabe mencionar que a plataforma Intel x86, Core i7 Ivy Bridge é utilizada apenas como um comparativo que faz alusão à uma plataforma externa ao contexto das plataformas ARM que foram utilizadas como plataformas alvo para medição, e, portanto, fora do foco central deste trabalho, servindo apenas como um ponto adicional de comparação.

Como a Tabela 2 nos permite constatar, a depender da plataforma obtivemos ganhos percentuais de até 43% para o SOSEMANUK executando em um Cortex-A15 cifrando 4096 bytes por vez, e 30% para o ChaCha20 executando num Cortex M3 cifrando 128 bytes ou 4096 bytes, de forma que esses processadores são característicos de sistemas embarcados. A penalidade causada pela implementação protegida do SOSEMANUK e consequente compromisso entre proteção e tempo de execução mencionados previamente neste trabalho é mensurado por picos de *perda* de 650% em ciclos por byte comparado à implementação de referência, não protegida, no Cortex-A7 cifrando 4096 bytes e para cifrar 128 bytes, uma *perda* de 239% no Cortex M4.

Não foram encontradas implementações alternativas além da referência do

 ©2016 SBC — Soc. Bras. de Computação

SOSEMANUK para comparação. A plataforma FELICS[1] possui diversas implementações do Chacha20, com destaque para uma implementação no Cortex-M3 que toma 55,43 ciclos por bytes na cifração de 128 bytes consecutivos e consome 740 bytes de ROM. A implementação descrita nesse trabalho é 12% mais rápida, porém quase 2 vezes maior. A plataforma SUPERCOP[2] possui implementações em *Assembly* ARM da Chacha20, portanto não diretamente comparáveis.

8. Conclusão

Este trabalho propõe otimizações às implementações de referência do ChaCha20 e do SOSEMANUK, onde, nas plataformas testadas, as implementações superam a de referência ou se igualam a esta considerando ±2% como flutuações de medição. As implementações trazem proteção adicional contra ataques de canal lateral de tempo por executarem em tempo constante. Vale ressaltar que as otimizações propostas ao ChaCha20 podem se estender às suas variantes com menor número de rodadas.

Agradecimentos

Os autores agradecem o apoio financeiro da empresa LG para financiamento dessa pesquisa, sob o projeto "*Efficient and Secure Cryptography for IoT*"

Referências

Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., and Sibert, H. (2008). Sosemanuk, a fast software-oriented stream cipher. In *The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 98–118. Springer.

Bernstein, D. J. (2005a). Chacha20, a variant of salsa20. `https://cr.yp.to/chacha/chacha-20080120.pdf`.

Bernstein, D. J. (2005b). Salsa20 specification. `https://cr.yp.to/snuffle/spec.pdf`.

Bernstein, D. J. (2008). The Salsa20 Family of Stream Ciphers. In *The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 84–97. Springer.

Biham, E., Anderson, R. J., and Knudsen, L. R. (1998). Serpent: A new block cipher proposal. In *FSE*, volume 1372 of *LNCS*, pages 222–238. Springer.

Ekdahl, P. and Johansson, T. (2002). A new version of the stream cipher SNOW. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 47–61. Springer.

eSTREAM (2004). the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`.

Morrow, M. (2004). Optimizing Memcpy improves speed. `www.embedded.com/design/configurable-systems/4024961/Optimizing-Memcpy-improves-speed`. Acessado: 09-09-2016.

[1] `https://www.cryptolux.org/index.php/FELICS_Stream_Ciphers_Brief_Results`
[2] `https://bench.cr.yp.to/results-stream.html`

 ©2016 SBC — Soc. Bras. de Computação