Implementação do 3DES em Sistemas Embarcados para Terminais de Ponto de Venda

Rodrigo Fontes Souto Vasco Roriz Elias Bechepeche Feliciano de Lima

Instituto Nokia de Tecnologia

RESUMO

Este artigo apresenta a implementação do algoritmo Triple Data Encryption Standard (3DES) em sistemas embarcados para terminais de Ponto de Venda (em inglês, Point of Sale, PoS). Deseja-se que o microcontrolador utilizado seja de baixo custo, o que reduz o preço final do PoS, e que tenha um desempenho satisfatório para o usuário, não tornando o produto lento. Terminais PoS são alvos freqüentes de ataques, principalmente por guardar e/ou transmitir informações sigilosas sobre seus usuários, e.g., senhas e números de cartões. Um dos pontos fracos destes terminais é quando um novo programa deve ser carregado em sua memória. Em cada PoS, há um firmware responsável pela gravação da nova aplicação. Portanto, o 3DES deve ser implementado para este firmware, que, por estar em uma região protegida de memória, tem uma disponibilidade de apenas 2Kb para seu código binário e de 1Kb para dados.

Palavras-Chave: Criptografia, Segurança, Sistemas Embarcados, *Bootloader*.

1. INTRODUÇÃO

Com o advento das telecomunicações, o mercado experimenta uma grande oferta em termos de *hardware* voltado para telecomunicações. Com o custo de tais produtos caindo cada vez mais, é possível inovar em aplicações inconcebíveis ate bem pouco tempo. Neste contexto é que foram aparecendo os aparelhos de terminais móveis com complexos sistemas embarcados. O maior expoente deste mercado é, sem dúvida, o aparelho celular. Entretanto, outros equipamentos foram surgindo, entre eles os terminais PoS.

Com a popularização dos terminais móveis surgiu outra necessidade: a segurança de tais equipamentos. Esta segurança vai desde o nível físico do aparelho (devido ao seu volume reduzido e à sua mobilidade), passando pelo nível lógico (como estes aparelhos têm acesso à Internet, também estão sujeitos aos ataques comuns da rede, e.g., vírus, *trojans*, invasões) e até ao nível de transmissão da informação (a informação muitas vezes é transmitida por meio de ondas eletromagnéticas e é acessível a todos com capacidade de interceptá-la). Desta forma, algumas medidas de segurança vêm sendo adotadas neste sentido, tais como: lacres que dão um aspecto visual de que o aparelho fora violado; o microcontrolador também pode estar ”vigiando” a integridade do aparelho; antivírus e *firewalls* mais seguros têm sido desenvolvidos. Protocolos mais seguros de comunicação na internet estão sendo adotados.

Implementações em *hardware* têm sido propostas em diversos trabalhos, principalmente em FPGA’s [1], [2]. Outras implementações são sugeridas para *hardware* dedicado [3] e para processadores Intel® [4]. Todavia, o custo de tais implementações é

elevado e seria refletido no preço final do produto, tornando-o inviável no mercado brasileiro. Optou-se, então, por uma solução de baixo custo utilizando microcontroladores de 8 bits. Diferentemente dos computadores pessoais, tais microcontroladores não possuem elevada capacidade de processamento e uma memória RAM e ROM bem limitadas, não ultrapassando 1Mb de espaço total, na maioria dos casos. Neste trabalho, o terminal de estudos será um terminal PoS, modelo FLEX®, com um microcontrolador ATMega8® embarcado. Em especial, o foco de interesse será uma aplicação bastante especial conhecida como *bootloader*. Tal aplicação, dadas as características do ATMega8®, não pode exceder o tamanho de 2Kb de programa e 1Kb de dados.

O *bootloader*, no entanto, por questões de segurança, deve ser capaz de executar algoritmos de criptografia. O objetivo deste trabalho é implementar o algoritmo 3DES no modo *Cipher Block Chaining* (CBC) com tamanho de chave de 56 bits, capaz de ser executado em tempo factível em um microcontrolador de 8 bits. O fabricante [5] disponibilizou em sua *home page* uma versão sugerida para *bootloader*. Contudo, tal versão encontra-se pouco otimizada em termos de desempenho e portabilidade. Portanto, deve-se melhorar o desempenho deste *firmware* (para que o usuário não sinta o produto como lento e sirva como um diferencial em relação à concorrência), deixando-o mais portátil, dividindo suas rotinas em módulos para que possam ser utilizadas por outra aplicação, e alterá-lo de forma a atender às especificidades do produto.

2. O ALGORITMO 3DES

O algoritmo *Data Encryption Standard* (DES), e sua variação, o 3DES, é uma das cifras de blocos mais utilizadas no mundo [1], [6]. Sua implementação pode ser vista com detalhes em [7], [8]. O 3DES foi projetado para encriptar blocos de 64 bits por meio de três chaves independentes de 64 bits, sendo que 8 destes bits são utilizados são utilizados para verificação de paridade. Portanto, o tamanho efetivo de cada chave é de 56 bits. Trata-se de uma cifra simétrica, i.e., a decriptação é realizada por meio das mesmas chaves utilizadas para a encriptação. Entretanto, para a decriptação, a ordem das rotinas é invertida. A funcionalidade do 3DES é mostrada de maneira simplificada na Figura 1. Matematicamente, o 3DES pode ser descrito como

$$C = DES_{k3}\left\{DES_{k2}^{-1}\{DES_{k1}(P)\}\right\},$$

denotando C como o texto cifrado, P como o texto em claro, DES_k como uma encriptação DES com a chave k e DES_k^{-1} como uma decriptação DES com a chave k.

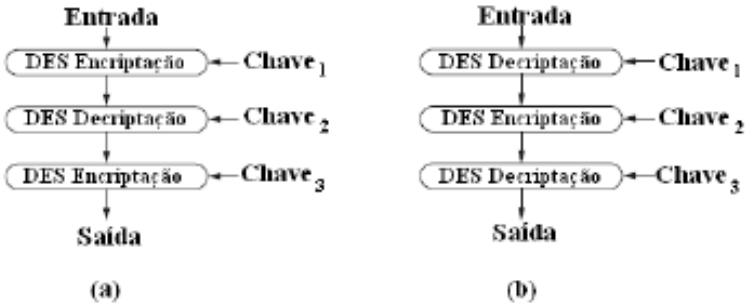

Figura 1. Fluxo do 3DES de (a) encriptação e (b) decriptação.

3. BOOTLOADER

Bootloader é uma aplicação com capacidade de gravar outra aplicação em um microcontrolador. Seu papel na segurança de um terminal é crucial. Há um grave risco de segurança no caso em que o *bootloader* permitisse a gravação de um programa invasor capaz de transmitir todos os dados da memória. O invasor teria acesso a senhas, números de cartões e de cheques, *Personal Identification Number* (PIN), IPs importantes etc, só para citar alguns casos de dados sigilosos. A figura 2 ilustra algumas fraquezas caso ocorra a gravação de um programa *bootloader* não autorizado.

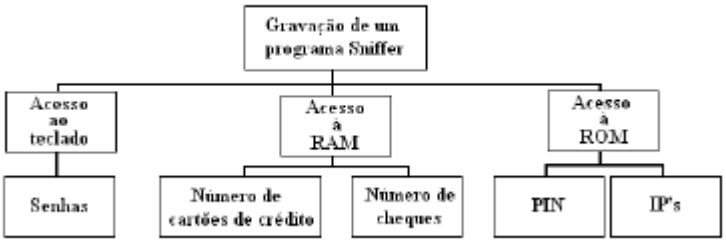

Figura 2. Algumas razões por que o bootloader deve ser seguro.

O *bootloader* em si é seguro. Ele é gravado em uma região segura da memória protegida por *hardware*. Uma vez ativada a proteção de tal região, não é permitido alterar nem ter acesso aos dados ali contidos. O problema é com a aplicação a ser gravada. Ela é insegura antes de ser gravada na memória *Flash* e antes de os bits de segurança serem corretamente configurados. Quando esta região precisa ser atualizada, pode ocasionar um acesso não autorizado por parte de um invasor.

Para se proteger deste ataque, a idéia consiste em encriptar os dados no servidor, enviá-los, e somente o *bootloader*, após ter completado o *download*, será capaz de decriptar os dados e conferir a integridade do pacote antes de gravar a nova aplicação. A chave de decriptação localiza-se em uma região de memória de forma que, uma vez que forem configurados os bits de segurança, sua leitura externa já não é mais possível, i.e, somente a aplicação interna, o *bootloader*, poderá acessá-la. A Figura 3 ilustra como o microcontrolador sai da fábrica e como é feito seu processo de atualização. O servidor encripta os dados como mostrado na Figura 1(a). Por sua vez, o microcontrolador deve decriptar os dados como mostrado na Figura 1(b).

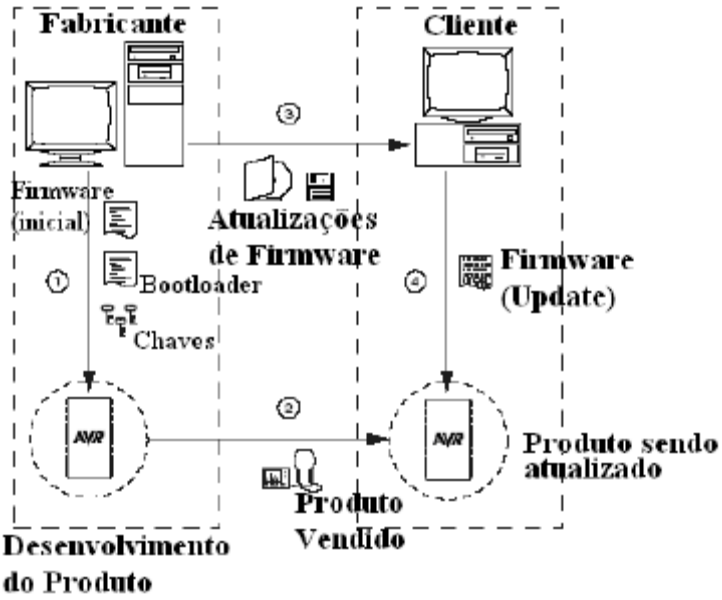

Figura 3. Exemplo de produção e procedimentos de atualização de aplicação.

4. PORTABILIDADE

4.1. MOTIVAÇÕES

Na solução proposta pelo fabricante, o código fora desenvolvido sem a preocupação de reuso do código. Em um terminal de ponto de venda, por se um alvo bastante visado, a questão da criptografia é bem explorada: diferentes algoritmos criptográficos são utilizados em diversos momentos, e.g., na transmissão de dados do teclado, na comunicação com *SIM cards* e *Smart cards* e para sistemas de encriptação e autenticação para *Virtual Private Networks* (VPN's), tais como o IPsec e SSL. Por motivos já expostos, sua utilização no *bootloader* também contribui para a segurança do aparelho. A Figura 4 ilustra alguns casos em que os algoritmos de criptografia são utilizados em terminais PoS.

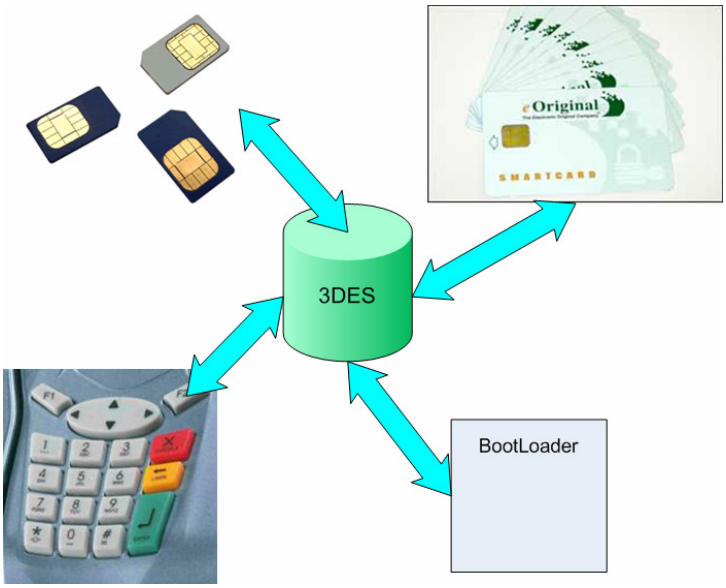

Figura 4. Funcionalidades que necessitam de algoritmos criptográficos.

Outro problema encontrado no código original do fabricante com relação à portabilidade foi com a chave utilizada. O código original utiliza uma chave fixa, uma vez que este código era para ser utilizado exclusivamente pelo *bootloader*. Tal decisão impede que outras aplicações de reutilizarem o código do 3DES.

4.2. RESULTADOS

Com as devidas modificações no código, foi possível passar de uma chave fixa, para uma chave variável, aumentando a segurança e a portabilidade do código.

Em relação a separação lógica, a opção de projeto foi separar um núcleo de rotinas básicas do modo de operação. O modo de operação utilizado no PoS é o CBC. Esta separação pode ser vista na Figura 5. Desta maneira, o código permite que outros modos de operação sejam utilizados. E melhor ainda, o algoritmo do 3DES está desacoplado, permitindo que seja utilizado por outras rotinas.

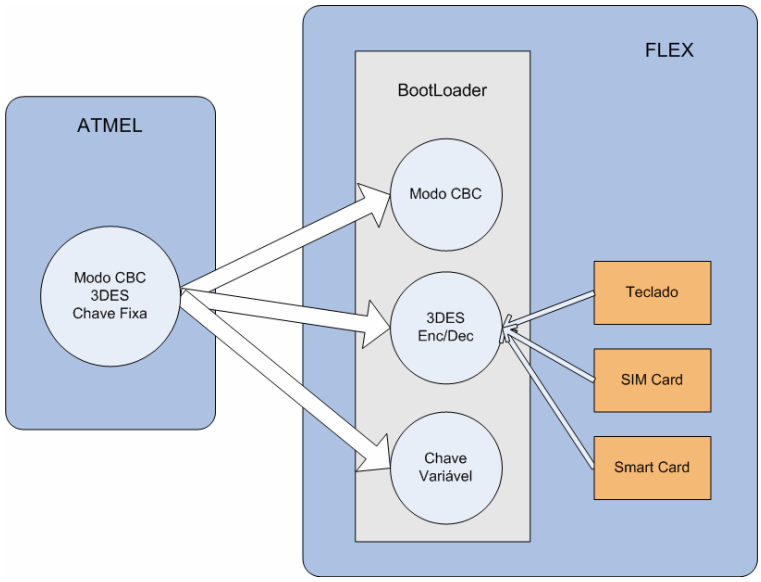

Figura 5. Separação lógica com o aproveitamento das rotinas criptográficas.

5. PERMUTAÇÕES

Por se tratar de código a ser colocado em um produto, há extrema preocupação em melhorar o desempenho do algoritmo, tanto para questões de *marketing* quanto em relação à fidelidade do cliente.

Desta forma, diversas melhorias foram propostas de maneira a reduzir o tempo de processamento do algoritmo. Só relembrando o *tradeoff* que existe entre tamanho e desempenho: na maioria dos casos, um aumento de velocidade implica em um aumento do espaço requerido em memória. E, no caso do *bootloader*, o espaço disponível é de apenas 2Kb, dificultando a melhora de desempenho do 3DES.

5.1. PERMUTAÇÕES NO 3DES

A maior parte do processamento do DES é representada pelas operações de permutação. A permutação é um mapeamento de bits onde um conjunto de bits de um vetor de entrada é mapeado para outro conjunto de bits em um vetor de saída. Existem ao todo 4 permutações na implementação original do DES:

- **Permutação inicial** - Entrada: 64 bits. Saída: 56 bits. Esta permutação retira os bits de paridade da chave de 64 bits e os reordena de uma outra forma;

- Permutação de expansão - Entrada: metade do bloco de texto em claro (32 bits). Saída: 48 bits.

- **Permutação da S-Box** - Entrada: 32 bits. Saída: 32 bits. Esta permutação é feita após a operação que envolve a chave da iteração, metade do bloco de texto em claro expandida e a S-Box.

- **Permutação final** - Inverso da permutação inicial.

Na implementação final foi acrescentada a permutação de compressão, utilizada para a geração da chave de iteração. Esta permutação não existe na implementação original porque a chave é fixa e as suas derivações para cada iteração já são calculadas antes da compilação.

O código fornecido pelo fabricante generalizou o processamento das permutações criando uma função que executa esta operação usando como parâmetro uma tabela específica para cada permutação. As permutações inicial e final contêm uma regularidade perceptível e, por economia de espaço, foram substituídas por funções que sintetizam as suas tabelas. A seguir, é apresentada a função genérica de permutação utilizada no código modificado e as funções auxiliares de acesso aos bits específicos.

```
// Permuta 'in' usando a tabela de permutação e coloca
// o resultado em 'out'
void permute( unsigned char *out,
unsigned char *in,
unsigned char (*func)(unsigned char),
unsigned char size )
{
do {
size--;
putBit(out, size, getBit(in, func(size)));
} while (size);
}
// Retorna o valor do bit 'index' na tabela de bits 'table'.
unsigned char getBit(unsigned char *table, unsigned char index) {
return (table[index >> 3] & (0x80 >> (index & 0x07)));
}
// Define o bit 'index' na tabela de bits 'table'
// considerando 'value'.
void putBit(unsigned char *table,
unsigned char index,
unsigned char value) {
unsigned char *p = &table[index >> 3];
unsigned char m = 0x80 >> (index & 0x07);
*p = (value) ? (m | *p) : (~m & *p);
}
```

A função genérica de permutação representa bem o *tradeoff* que existe entre tamanho de código e desempenho. Ela reduz o tamanho do código podendo ser utilizada para todas as

operações de permutação do DES com o uso de tabelas ou de funções que sintetizam as tabelas, o que for mais conveniente. Todavia, gastam-se muitos ciclos de CPU para efetuar a permutação de um único bit. O desafio principal foi encontrar uma implementação que pudesse substituir a função genérica mantendo o tamanho de código viável. A solução foi substituir todas as permutações por funções específicas implementadas com uso de instruções em *assembly*.

A permutação da S-Box que remapeia 32 bits, por exemplo, seria implementada com 32 operações de cópia de um bit no vetor de origem para outro bit no vetor de destino. A sua tabela de permutação é dada por

```
BOOTFLASH unsigned char pTable[32] = {
15,  6, 19, 20,
28, 11, 27, 16,
 0, 14, 22, 25,
 4, 17, 30,  9,
 1,  7, 23, 13,
31, 26,  2,  8,
18, 12, 29,  5,
21, 10,  3, 24 };
```

Pela notação utilizada, o bit 0 é o bit mais significativo (MSB) em um byte, e o bit 15 seria o bit 7 (bit menos significativo, LSB) do segundo byte (byte 1) (byte = 15/8 = 1; bit = 15%8=7). O primeiro elemento de valor 15 significa que o bit 0 no vetor de saída deverá ser o bit 15 no vetor de entrada. é importante perceber que a notação de bits da tabela não é a normalmente usada na programação, em que o bit 0 é o LSB.

Para ter desempenho utilizando o *assembly inline*, foi necessária a alocação prévia das variáveis em registradores. Sendo *r13* o segundo byte do vetor de entrada e *r11* o primeiro byte do vetor de saída, a permutação do primeiro bit da tabela foi implementada da forma

```
asm("BST r13,0"); //copia bit menos significativo de r13
asm("BLD r11,7"); //salva no bit mais significativo de r11
```

Esta operação é repetida para os outros 31 bits da tabela, gastando duas instruções por bit, ao todo 128 bytes. Neste caso, gastou-se mais que a tabela (32 bytes), se não considerarmos o esquema genérico de permutação extra. Isto ocorreu porque esta permutação é muito aleatória, não contendo nenhuma regularidade que possa representar simplificação em sua implementação.

Houve economia nas permutações inicial e final, bem como na permutação de expansão, de forma que o esquema genérico de permutação não foi utilizado e ainda assim o espaço foi suficiente.

5.2. OTIMIZAÇÃO ESPECIAL 1 – PERMUTAÇÕES INICIAL E FINAL

A permutação inicial possui um comportamento regular que permitiu a implementação de uma rotina com tamanho reduzido para executar sua função. Pode ser observado que a cada

linha da tabela de permutação o valor inicial é decrescido de 8. Uma implementação que sintetiza esta tabela é obtida por meio da função

```
unsigned char ip(unsigned char i)
{
return 57-((i & 0x07) << 3) + ((i >> 2) & 0x06) - ((i >> 5) & 0x01);
}
```

Esta rotina utiliza 34 bytes de memória, que é menor que a tabela de 64 bytes, porém precisa da rotina de permutação genérica. Como o esquema de permutação genérico foi abandonado por motivo de desempenho, o desafio foi criar uma rotina que não apenas sintetizasse a tabela de permutação inicial, mas já executasse esta permutação. Com o uso de 8 repetições de uma seqüência escrita em *assembly*, a rotina utilizou apenas 52 bytes. A permutação final (também chamada de inicial inversa) poderia ser implementada da mesma forma, porém, após inspeção nas tabelas, foi feita a constatação de que a rotina otimizada da permutação inicial pode ser utilizada para realizar a permutação final, com alguns procedimentos adicionais que serão descritos mais adiante.

Observa-se que a tabela de permutação inicial (ipTable) pode ser transformada na tabela de permutação final (iipTable) com operações de trocas de linhas e colunas. Numa tabela de permutação, a posição de um elemento está relacionada com a posição do bit no vetor de saída, enquanto o valor deste elemento representa a posição de um bit no vetor de entrada. Em ipTable, o primeiro elemento tem valor igual a 57. Isto significa que o bit 0 no vetor de saída terá o mesmo valor do bit 57 no vetor de entrada. As tabelas são mostradas a seguir.

```
// Tabela de permutação inicial (ipTable)
BOOTFLASH unsigned char ipTable[64] = {
57, 49, 41, 33, 25, 17, 9, 1,
59, 51, 43, 35, 27, 19, 11, 3,
61, 53, 45, 37, 29, 21, 13, 5,
63, 55, 47, 39, 31, 23, 15, 7,
56, 48, 40, 32, 24, 16, 8, 0,
58, 50, 42, 34, 26, 18, 10, 2,
60, 52, 44, 36, 28, 20, 12, 4,
62, 54, 46, 38, 30, 22, 14, 6 };
// Tabela de permutação final (iipTable)
BOOTFLASH unsigned char iipTable[64] = {
39, 7, 47, 15, 55, 23, 63, 31,
38, 6, 46, 14, 54, 22, 62, 30,
37, 5, 45, 13, 53, 21, 61, 29,
36, 4, 44, 12, 52, 20, 60, 28,
35, 3, 43, 11, 51, 19, 59, 27,
34, 2, 42, 10, 50, 18, 58, 26,
33, 1, 41, 9, 49, 17, 57, 25,
32, 0, 40, 8, 48, 16, 56, 24 };
```

Observe que permutando as linhas da tabela de permutação inicial na ordem {3 7 2 6 1 5 0 4}, ou seja, a linha 0 assume todos os valores da linha 3, a linha 1 assume os valores da linha 7 e assim por diante, a tabela ipTable se torna

```
BOOTFLASH unsigned char iipTableIntermediaria1[64] = {
63, 55, 47, 39, 31, 23, 15, 7,
62, 54, 46, 38, 30, 22, 14, 6
61, 53, 45, 37, 29, 21, 13, 5,
60, 52, 44, 36, 28, 20, 12, 4,
59, 51, 43, 35, 27, 19, 11, 3,
58, 50, 42, 34, 26, 18, 10, 2,
57, 49, 41, 33, 25, 17,  9, 1,
56, 48, 40, 32, 24, 16,  8, 0 };
```

Permutando agora as colunas da tabela de permutação iipTableIntermediaria1 na mesma ordem {3 7 2 6 1 5 0 4}, a tabela se torna:

```
BOOTFLASH unsigned char iipTableIntermediaria2[64] = {
39, 7, 47, 15, 55, 23, 63, 31,
38, 6, 46, 14, 54, 22, 62, 30
37, 5, 45, 13, 53, 21, 61, 29,
36, 4, 44, 12, 52, 20, 60, 28,
35, 3, 43, 11, 51, 19, 59, 27,
34, 2, 42, 10, 50, 18, 58, 26,
33, 1, 41,  9, 49, 17, 57, 25,
32, 0, 40,  8, 48, 16, 56, 24 };
```

Percebe-se que iipTableIntermediaria2 é idêntica à tabela de permutação final. Portanto, a tabela de permutação final foi obtida a partir da tabela de permutação inicial simplesmente permutando suas linhas e colunas.

Procede-se com a análise de que isto significa para a reutilização da rotina otimizada para servir às duas permutações. Nestas tabelas de permutação, uma troca de linha equivale a uma troca de um byte inteiro no vetor de saída, uma vez que a posição da tabela está relacionada ao vetor de saída. A troca de colunas é equivalente à troca de bits dentro de todos os bytes do vetor de saída. Permutar os bytes dentro do vetor de saída é uma operação simples, pois são apenas 8 bytes e as operações com bytes são mais eficientes que as operações com bits. Entretanto, permutar os bits dentro dos 8 bytes pode consumir um tempo maior, pois são 64 bits e as operações com bits são menos eficientes.

A simplificação ocorre quando se observa os valores contidos em cada coluna da tabela de permutação inicial: eles estão todos limitados a um byte único. Por exemplo, a primeira coluna possui valores entre 57 e 63 (byte 7 no vetor de entrada), a última coluna contém valores entre 0 e 7 (byte 0 no vetor de entrada) e isto ocorre com todas as colunas da tabela. Logo não é necessário permutar os bits do vetor de saída, basta permutar também os bytes do vetor de entrada. O algoritmo para realizar a permutação final é dado por

a) Permutar os bytes do vetor de entrada na ordem {3 7 2 6 1 5 0 4}.

b) Executa a mesma rotina modificada de permutação inicial.

c) Permutar os bytes do vetor de saída na mesma ordem {3 7 2 6 1 5 0 4}.

5.3. OTIMIZAÇÃO ESPECIAL 2: S-BOX

Um estágio importante do algoritmo do DES é implementado com o uso de uma tabela de substituição (S-Box). A cada chamada é utilizada uma S-Box diferente com 64 elementos, normalmente agrupados em 32 bytes, pois cada elemento está em um limite de 16 bits. Neste

estágio, o resultado da combinação da chave da iteração com o resultado da permutação de expansão, um vetor de 48 bits, é utilizado como entrada. Estes 48 bits são divididos em 8 palavras de 6 bits, que serão usadas para indexar a S-Box.

A fim de se evitar a separação dos grupos de 6 bits nos 48 bits, a permutação de expansão (32 bits → 48 bits) e a de compressão (56 bits → 48 bits) foram implementadas para terem como saída um vetor de 64 bits organizados em 8 bytes. Cada byte possui apenas 6 bits significativos, resultando nos 48 bits originais. A obtenção dos 6 bits de indexação da S-Box ficou então imediata, aumentando a velocidade e eliminando o código de extração dos 6 bits em 48 bits.

Outra otimização foi feita na forma de indexação da S-Box. Cada S-Box contém 64 elementos, organizados em 4 linhas e 16 colunas. Conforme o algoritmo, 6 bits são combinados de forma que os bits 0 e 5 indicam a linha e os bits 1, 2, 3 e 4 indicam a coluna. Na implementação original, é necessário um código extra que extrai linha e coluna e converte em um índice linear para acesso à S-Box. Por economia de memória, os elementos são endereçados em nibbles, tendo ainda que utilizar o bit 4 para decisão de *nibble* mais ou menos significativo. Optou-se por reorganizar a S-Box de forma que os 6 bits permitam uma indexação direta aos seus elementos. A reorganização nada mais é que uma permutação na S-Box, que não é fácil de perceber por causa da compactação dos *nibbles* em bytes feita para economia de código. Segue o exemplo para a S-Box 1.

```
// S-Box 1 Original
0xE4, 0xD1, 0x2F, 0xB8, 0x3A, 0x6C, 0x59, 0x07,
0x0F, 0x74, 0xE2, 0xD1, 0xA6, 0xCB, 0x95, 0x38,
0x41, 0xE8, 0xD6, 0x2B, 0xFC, 0x97, 0x3A, 0x50,
0xFC, 0x82, 0x49, 0x17, 0x5B, 0x3E, 0xA0, 0x6D.
//S-Box 1 Reorganizada
0xE0, 0x4F, 0xD7, 0x14, 0x2E, 0xF2, 0xBD, 0x81,
0x3A, 0xA6, 0x6C, 0xCB, 0x59, 0x95, 0x03, 0x78,
0x4F, 0x1C, 0xE8, 0x82, 0xD4, 0x69, 0x21, 0xB7,
0xF5, 0xCB, 0x93, 0x7E, 0x3A, 0xA0, 0x56, 0x0D.
```

5.4. RESULTADOS

O tempo de execução de cada etapa do algoritmo foi utilizado como parâmetro de desempenho do software. Foram feitas medidas de tempo para o código original e código modificado para efeito de comparação em um microcontrolador ATMega8®. O número de ciclos de máquina seria um critério de desempenho mais rigoroso, porém um pouco mais complexo de ser determinado em relação ao tempo, que pode ser facilmente medido com um osciloscópio e um pino de *Input/Output* (IO), praticamente sem causar nenhum overhead na medida. Os ganhos mostrados nas tabelas são calculados dividindo-se os valores do algoritmo original com o modificado. A Tabela 1 mostra a comparação entre os tempos de processamento de diferentes rotinas do 3DES para os algoritmos original e modificado.

Tabela 1 – Comparação entre os tempos do algoritmo original e modificado de diferentes rotinas.

Função	Tempo gasto pelo algoritmo		*Ganho*
	Original (us)	Modificado (us)	
Permutação inicial	720	40.4	17.82x

Permutação de Expansão	900	22.2	40.54x
Acesso à chave*	19.6	40.0	00.49x
Acesso à S-Box	424	38.8	10.93x
Permutação da S-Box	610	13.2	46.21x
XOR 4 bytes	12.4	12.4	01.00x
Permutação final	*720*	*42.5*	16.94x

*O acesso à chave na implementação original é simplesmente a operação XOR com a chave da iteração previamente calculada. No código modificado, a chave é calculada a cada iteração, acrescentando-se as operações de deslocamento e permutação de compressão.

Com exceção das permutações inicial e final, as outras etapas são executadas a cada uma das 16 iterações do DES. A comparação para as operações de ciframento/deciframento de um bloco de 64 bits é mostrada na Tabela 2. As taxas obtidas, em bits por segundo (bps), são mostradas na Tabela 3.

Tabela 2 - Comparação entre os tempos do algoritmo original e modificado para cifrar/decifrar blocos de 64 bits.

Função	Tempo gasto pelo algoritmo		*Ganho*
	Original (ms)	Modificado (ms)	
Cifrar/decifrar 64 bit (DES)	32.9	2.11	15.59x
Cifrar/decifrar 64 bit (3DES)	95.8	5.35	17.90x

Tabela 3 – Comparação entre as taxas do algoritmo original e modificado para cifrar/decifrar blocos de 64 bits.

Função	Taxa dos algoritmos		*Ganho*
	Original (Bps)	Modificado (Bps)	
Cifrar/decifrar 64 bit (DES)	243.16	3791.47	15.59x
Cifrar/decifrar 64 bit (3DES)	83.5	1495.33	17.90x

O resultado mais significativo, porém, é a redução do tempo para a atualização remota. No final, este é o tempo percebido pelo usuário ao ter seu *software* atualizado. A Tabela 4 mostra que o desempenho melhorou em 17.9 vezes, i.e., o tempo de atualização do *software* passou de 73.58s para 4.11s utilizando o 3DES, apresentado um resultado bastante satisfatório.

Tabela 4 – Comparação entre os tempos gastos do algoritmo original e modificado para a atualização remota.

Função	Tempo gasto pelo algoritmo (s)		*Ganho*
	Original	Modificado	
Cifrar/decifrar 6kB (DES)	25.27	1.62	15.60x
Cifrar/decifrar 6kB (3DES)	73.58	4.11	17.90x

A Tabela 5 ilustra os recursos utilizados por implementação. As medidas de tamanho de código foram retiradas do arquivo de mapa gerado após a compilação e montagem do código.

Tabela 5 – Comparação entre recursos de memória utilizados.

Recurso	Tamanho ocupado pelo algoritmo (Bytes).	
	Original	*Modificado*
Código	1935	2001
Dados	192	384
Máximo uso do *Stack*	49	36
Registradores Reservados	*0*	9

Quando a aplicação fizer uso das rotinas de DES já presentes no *bootloader*, deverá levar em conta o máximo uso do *stack*, sendo que a rotina de ciframento/deciframento deve ser chamada tendo no mínimo 36 bytes livres em *stack*. A aplicação deve ainda reservar 9 dos 32 registradores do microprocessador ATMega8®. Quando a rotina de ciframento/ deciframento não for utilizada, estes registradores podem ter outro uso.

6. CONCLUSÕES

A partir deste estudo, foi possível implementar o algoritmo do 3DES em um microcontrolador limitado tanto em capacidade de processamento tanto em memória disponível. Em especial, foi implementado para uma aplicação específica de um *bootloader*.

As modificações realizadas no código permitiram uma melhora de aproximadamente 18 vezes na velocidade de cálculo do algoritmo. Utilizando o código do fabricante, a atualização remota durou 73,58s. Todo o processo de atualização de versão passou para 4.11s, viabilizando seu uso em produtos de aplicação comercial.

Outra importante característica adicionada ao código foi a portabilidade, permitindo que outra aplicação pudesse utilizar as mesmas rotinas para criptografar outros dados, tais como informações relativas ao teclado, ao *SIM Card* e ao *Smart Card*.

5. CITAÇÕES

As citações devem ser apresentadas no texto segundo o formato sobrenome do autor e o ano da publicação. Por exemplo: ZELENY (1982).

7. REFERÊNCIAS

[1] HAMALAINEN, P.; HANNIKAINEN, M.; HAMALAINEN, T. e SAARINEN, J., "Configurable hardware implementation of triple-DES encryption algorithm for wireless local area network", IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), v. 2, p. 1221 - 1224, Maio 2001.

[2] CHIN MUN WEE; SUTTON, P.R. e BERGMANN, N.W., "An FPGA network architecture for accelerating 3DES - CBC", International Conference on Field Programmable Logic and Applications, p. 654 - 657, Agosto 2005. [3] Schaffer,T.; Glaser, A. e Franzon, P.D., "Chip-package Coimplementation of a triple DES Processor", IEEE Transactions on Advanced Packaging, v. 27, p. 194 - 202, Fevereiro 2004.

[4] SANG-SU LEE; MIN HO HAN; JEONG-NYEO KIM, “An implementation of 3DES and HMAC-MD5 in Intel IXP 2400”, The 7th International Conference on Advanced Communication Technology (ICACT), v. 1, p. 369 - 371, Fevereiro 2005.

[5] ATMEL, “Doc 2541 DES Bootloader”, http://www.atmel.com. Acessado em 24/07/2006.

[6] NADEEM, A.; JAVED, M.Y.; “A Performance Comparison of Data Encryption” First International Conference on Algorithms Information and Communication Technologies (ICICT), p.84 - 89, Agosto 2005.

[7] B. SCHNEIER, Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, 2a Ed., 1995.

[8] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, “Data Encryption Standard (DES)”, Federal Information Processing Standards (FIPS), Publication 46-7, EUA, 1999.