UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Gabriel Garcia Gava

ANÁLISE DA EFICIÊNCIA ENERGÉTICA DO CIFRADOR AES SUBMETIDO A DIFERENTES OTIMIZAÇÕES

Florianópolis - Santa Catarina

2014

Gabriel Garcia Gava

ANÁLISE DA EFICIÊNCIA ENERGÉTICA DO CIFRADOR AES SUBMETIDO A DIFERENTES OTIMIZAÇÕES

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciência da Computação para a obtenção do Grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Luiz Cláudio Villar dos Santos
Coorientador: Prof. Dr. Daniel Santana de Freitas

Florianópolis - Santa Catarina

2014

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Gabriel Garcia Gava

ANÁLISE DA EFICIÊNCIA ENERGÉTICA DO CIFRADOR AES SUBMETIDO A DIFERENTES OTIMIZAÇÕES

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciência da Computação.

Florianópolis - Santa Catarina, 02 de Dezembro 2014.

Prof. Dr. Renato Cislaghi
Coordenador

Banca Examinadora:

Prof. Dr. Luiz Cláudio Villar dos Santos
Orientador

Prof. Dr. Daniel Santana de Freitas
Coorientador

Prof. Dr. José Luís Almada Güntzel

AGRADECIMENTOS

Aos meus pais, Gilberto e Elisabete, e irmãos, Gustavo e Taiana, por sempre estarem ao meu lado e apoiarem minhas decisões. Por terem sido fundamentais na minha educação, e em tudo que sou hoje.

À Nathália, por sempre me acompanhar e ajudar em todos os momentos, e pela paciência e compreensão. Obrigado por fazer parte desses últimos 4 anos da minha vida.

À minha madrinha Margareth, padrinho Cesar e primo Maurício por participarem constantemente da minha formação.

À toda a minha família por sempre lembrarem de mim, que mesmo com tanta distância, faz questão de nos reunirmos sempre que possível.

Aos meus amigos que me acompanharam durante este curso e me ajudaram a chegar até aqui.

Ao meu orientador, Luiz Cláudio, que sempre confiou na minha capacidade e me motivou a seguir com o meu trabalho, mas nunca permitiu que minha vida profissional atrapalhasse a vida pessoal.

A todos os meus colegas de laboratório, especialmente ao Felipe e ao Rodrigo, que trabalharam comigo durante a execução deste trabalho, o qual não seria possível sem a ajuda deles.

The scariest moment is always just before you start.

Stephen King

RESUMO

Segurança sempre foi um tópico muito importante na história da Computação. De fato, os primeiros computadores eletrônicos surgiram com o objetivo de decifrar mensagens secretas. Por muito tempo, o foco das pesquisas em segurança computacional foi a qualidade da segurança e o em tempo de execução. Por outro lado, o grande crescimento do uso de sistemas embarcados alimentados a bateria fez com que outra restrição se fizesse importante: a eficiência energética.

Um dos algoritmos de criptografia mais usados atualmente é o AES. Sua estrutura permite implementações significativamente diferentes. Este trabalho estudou o gasto energético induzido ao executar este algoritmo, em suas diferentes implementações e quando submetido a diferentes otimizações de código.

Palavras-chave: Sistemas embarcados. Eficiência energética. Otimização de código. Criptografia. AES.

ABSTRACT

Security had always been a very significant subject in the history of Computer Science. Indeed, the first eletronic computers came with the purpose of breaking secret messages. For a long time, the research in computer security focused in the security level and execution time. On the other hand, the increasing use of the embedded systems powered by battery created another important constraint: the energy efficiency.
One of the most used encryption algorithm nowadays is AES. Its structure allow meaningly distinct implementations. This work studies the energy consumption induced by the execution of that algorithm, in its different implementationsand when subjected to different code optimizations.

Keywords: Embedded systems. Energy efficiency. Code optimization. Cryptography. AES.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

Os sistemas embarcados estão cada vez mais presentes no dia-a-dia das pessoas. Uma classe deles, os dispositivos móveis, é responsável por uma porcentagem considerável desses sistemas, e uma característica muito importante desses dispositivos, é que eles são alimentados por bateria. Esses equipamentos móveis sofrem fortes restrições de tempo, espaço, e energia. Além dessas restrições, a evolução da internet está direcionando esses sistemas a um ambiente totalmente conectado, o que é comprovado pelo fato que atualmente todos os novos celulares já têm acesso a internet. Isso cria uma nova restrição, tão importante quanto as outras, para esses aparelhos: a comunicação segura.

Porém, um dos maiores desafios para esses sistemas é a incompatibilidade entre os requisitos energéticos para a segurança e a capacidade das baterias atuais. Essa preocupação se dá pelo fato de que a taxa de comunicação está aumentando mais rápido do que a capacidade das baterias, e essa distância, chamada de *battery gap* entre a necessidade e a disponibilidade está aumentando (POTLAPALLY et al., 2006). Isso significa que, se a evolução das baterias continuar a mesma, será preciso encontrar contramedidas que possibilitem a continuação do crescimento do uso desses equipamentos móveis de forma segura. Essas medidas para redução de consumo energético não é só importante para reduzir o efeito do *battery gap*, mas também para o uso de dispositivos móveis com capacidades ainda mais limitadas, como as etiquetas RFID (ASHRY; SHARAF; IBRAHIM, 2009) (PILLAI et al., 2007), as quais também contém requisitos de segurança.

Analisando estes problemas, pode-se notar a necessidade de redução do consumo energético dos protocolos de segurança e algoritmos criptográficos, os quais disponibilizam a comunicação segura requerida. Os algoritmos criptográficos costumam ser divididos em duas classes: os simétricos e os assimétricos. Dentre os cifradores simétricos, há alguns que são mais usados, como por exemplo o AES, 3DES, Blowfish, Serpent, RC4. Neste trabalho, demos foco ao algoritmo AES, um dos mais importantes cifradores simétricos por conter várias garantias de segurança e ser o padrão adotado pelo governo dos Estados Unidos para suas comunicações (BURR, 2003). O objetivo deste trabalho é então, analisar os gastos energéticos causados pelo algoritmo AES no escopo de um dispositivo móvel. Não só obter os gastos sobre as implementações diretas do algoritmo, mas também a ele submetido a diferentes técnicas de otimização de código e, além disso, observar quais características do algoritmo ou do dispositivo móvel podem ser exploradas para obter uma melhor eficiência energética.

1.1 SISTEMAS EMBARCADOS

Um sistema embarcado é um sistema de microprocessadores que são construídos para alcançar objetivos específicos, e são desenvolvidos de forma que não sejam reprogramáveis, definição dada por (MARWEDEL, 2006). Em (HENNESSY; PATTERSON, 2012), pode-se encontrar um definição muito semelhante, mas ainda é colocado que dispositivos embarcados são mais limitados quanto a sofisticação de software e hardware, e que a habilidade de rodar software de terceiros é o que diferencia um computador não embarcado de um embarcado. Os dispositivos móveis conhecidos como *Personal Mobile Device*, ou PMD , como celulares e tablets, são muitas vezes considerados sistemas embarcados, porém eles podem ser reprogramáveis e também permite-se o uso de novos softwares. Isso faz com que os PMDs sejam tratados como um caso especial.

Os sistemas embarcados começaram a ser usados antes mesmo do surgimento dos PCs e são os consumidores majoritários dos microprocessadores atualmente. Eles costumam estar embutidos em vários equipamentos utilizados pelas pessoas no dia-a-dia sem que nem mesmo elas saibam, como em carros, ar-condicionados, e até mesmo em maquinas de lavar roupas. Mas uma classe deles que vem tomando muito espaço atualmente, são os dispositivos móveis, aqueles alimentados à bateria, pois estes trazem o benefício da ubiquidade, ou seja, podem ser usados em qualquer lugar, a qualquer momento, sem a necessidade de estarem conectados a uma rede elétrica. Não é por menos, podemos observar na Figura 1, como o uso dos dispositivos móveis vêm substituindo os computadores pessoais (PCs) .

Mas apesar desta vantagem, os dispositivos alimentados por bateria (DAB) trazem restrições mais rígidas de tempo, espaço e energia. Por exemplo, os celulares precisam ter sistemas de rádio para tratar das chamadas, o processamento envolvido por estes dispositivos precisa acontecer de forma rápida, para que possa ser possível manter uma comunicação agradável. Ao mesmo tempo, esses celulares precisam ser pequenos para ter uma boa mobilidade, mas isso faz com que se tenha menos espaço para o sistema de memória e processamento. Além dessas duas características, é importante que eles tenham um tempo de carga aceitável, pois ter que recarregar o equipamento a toda hora fere os benefícios da ubiquidade, e incomodam bastante o usuário final.

Atualmente, os DABs vêm sendo integrados à rede de computadores, quer dizer, cada vez mais eles estão se conectando à internet, criando uma rede fortemente conectada. Além disto, um novo paradigma vem sendo muito discutido, o Internet of Things (IoT) (ATZORI; IERA; MORABITO, 2010) (BASSI; HORN, 2008) . A ideia do IoT, é que cada objeto do mundo seja

unicamente identificado, e possam agir como agentes ativos, tomando suas próprias decisões, conectados aos contextos sociais de forma heterogênea, ou seja, objetos com funcionalidades totalmente distintas, mas pertencendo a um mesmo ambiente de comunicação. Isto requer que os objetos carreguem sistemas com eles, e a ideia mais natural para isso são as RFIDs , que são etiquetas com um sistema embarcado. Hoje em dia os RFIDs normalmente são passivos e costumam ser usados para sistemas anti-furtos, ou outras funcionalidades menores. Mas a utilização dessas etiquetas com o IoT traz novos desafios além daqueles de espaço e eficiência energética já esperados. Estando eles num ambiente em que todos podem se conectar entre si, situações críticas de segurança tornam-se muito importantes, como comunicações que envolvam transferência bancária, por exemplo. Resumindo, a integração dos DABs, da Internet, e da IoT implicam em uma outra restrição de extrema importância além daqueles citadas anteriormente, a segurança.

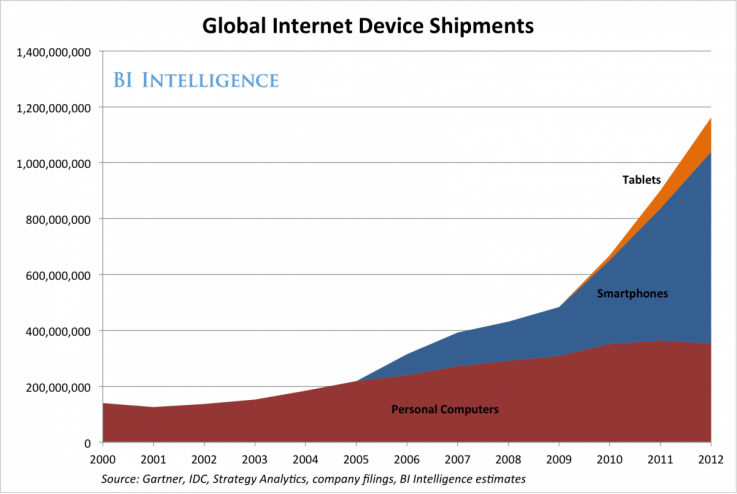


Figura 1 – Demanda de dispositivos com acesso a Internet.[1]

Neste trabalho, o enfoque escolhido foram os dispositivos móveis, como tablets e smartphones, já que estes vêm tomando o lugar dos PCs nos dias de hoje. Para isto, foi escolhido a arquitetura ARM como alvo do estudo, visto que este é o processador mais utilizado pelos dispositivos móveis atualmente. Uma breve introdução e características interessantes sobre estes

[1] http://www.businessinsider.com/2013–the-year-ahead-in-mobile-slide-deck-2013-12 Acessado em: 28 de Junho de 2013

processadores podem ser vistos a seguir.

1.1.1 A arquitetura ARM

Quando falamos de arquitetura ARM, na verdade estamos nos referindo aos *instruction set architectures* (ISAs) baseados na arquitetura RISC (Reduced Instruction Set Computing) que foram desenvolvidas pela empresa ARM holdings. Os processadores desenvolvidos usando esta tecnologia são usados em muitas classes de aplicações, como smartphones, microcontroladores e outros sistemas embarcados. As arquiteturas RISCs são importantes no mundo de embarcados pois precisam muito menos transistores que sistemas que usam arquitetura CISC (Complex Instruction Set Computing) , que é o caso da maioria dos desktops e servidores. Isto faz com que os RISCs sejam arquiteturas mais simples, possibilitando melhor eficiência energética que um CISC com funcionalidades equivalentes. Hoje, os processadores ARM lideram em número de processadores produzidos pelo mundo.

Uma das ISAs mais usadas hoje em dia nos smartphones é a ISA ARMv7-A (ARM, 2012), uma arquitetura RISC de 32 bits. A letra *A* significa que é uma arquitetura de ARMv7 com perfil para aplicações, que busca alto desempenho mantendo uma baixa potência. Um característica importante do ARMv7-A é a presença do Thumb-2, que é um thumb instruction set, isto é, um conjunto de instruções com tamanho reduzido, que possibilita a diminuição do tamanho de código, acelerando assim a execução de código por melhorar o uso da cache de instruções. Outras características importantes do ARMv7-A é a compatibilidade com instruções de ARMv6, a existência de um sistema de segurança em relação aos acessos dos componentes do sistema, suporte a ponto flutuante, multiprocessamento, entre outras. Dentro da arquitetura ARMv7-A ainda podemos encontrar várias implementações possíveis para este conjunto de instruções, como por exemplo o ARM Cortex-A9 e o ARM Cortex-A15.

Para finalizar essa seção, gostaria de citar algumas funcionalidades interessantes do conjunto de instruções do ARMv7. Neste conjunto há uma instrução de rotação para a direita, que rotaciona um número x de bits de um registrador. Rotacionar bits em um registrador é parecido com a operação de shift, porém os bits que sairiam do registrador, entram pelo outro lado. A figura 2 mostra um exemplo de rotação de 1 bit. Esta operação é interessante pois como veremos no trabalho, o algoritmo AES utiliza em vários momentos este tipo de operação, e como existe uma operação nativa de ARM que a realiza, temos um grande ganho de eficiência e tamanho de código se utilizarmos esta instrução.

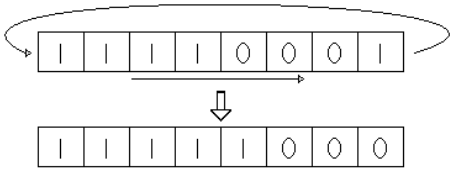


Figura 2 – Exemplo de rotação.

Uma segunda funcionalidade bastante útil para o AES, é a possibilidade de realizar operações de rotação ou shift como um preambulo de outra operação qualquer, isto é, podemos rotacionar o valor de um registrador antes que ele seja submetido a uma operação lógica ou de acesso a memória. Isto também acelera bastante o funcionamento do AES, pois muitas operações do AES consistem em rotacionar um valor, e acessar esta posição de memória logo em seguida, e com as funcionalidades do ARMv7 podemos fazer tudo isso em uma única instrução.

Pelo fato da arquitetura ARM ser a mais presente nos dias de hoje em relação a sistemas computacionais, e por disponibilizar muitas funcionalidades interessantes para o algoritmo AES. Esta foi a arquitetura escolhida para o escopo deste trabalho.

1.2 EFICIÊNCIA ENERGÉTICA

O conceito de eficiência energética é bastante vasto, e está relacionado à elaboração e execução de técnicas que reduzam a razão gasto energético por algum tipo de recurso, como o número de operações executadas. Sendo este um assunto bastante abrangente, o escopo deste trabalho é a melhora da eficiência energética nos dispositivos alimentados por bateria, que sofrem fortíssimas restrições de potência.

1.2.1 Importância

Por muito tempo da história da computação, as pesquisas nas áreas de sistemas e arquiteturas de computadores eram focadas na evolução do desempenho. E isto era feito de diversas maneiras, como o aumento da frequência do relógio do processador. O aumento da frequência faz sentido, como podemos observar na equação 1.1. Na equação, I significa quantidade de instruções executadas, CPI a média de ciclos por instrução, f a frequência e TE o tempo total de execução. Logo, o tempo de execução reduz linearmente com o aumento da frequência.

$$TE = \frac{I * CPI}{f} \tag{1.1}$$

Mas a frequência é um dos fatores relacionados ao consumo energético, como visto na equação 1.2. Nela, P_d representa a potência dinâmica, C a capacitância e Vdd a tensão de alimentação. Ou seja, a potência dinâmica aumenta linearmente em relação ao aumento da frequência. Uma das técnicas utilizadas para reduzir a potência, é a redução da alimentação, que tem uma importância quadrática, porém esta redução é limitada, pois tensões baixas de alimentação podem fazer com que o sinal não consiga ser propagado pelo sistema.

$$P_d = C * Vdd^2 * f \tag{1.2}$$

Na figura 3, vemos que a relação do aumento da potência em relação a frequência no passar dos anos. De 1982 a 2004 houve um aumento aproximado de 300 vezes na frequência (o que pela fórmula 1.1, significa que os programas passaram a executar 300 vezes mais rápido), mas a potência só teve um aumento de 30 vezes. Isso ocorreu pois houve uma diminuição do Vdd. Mesmo assim, existe um certo limite de potência que os equipamentos podem alcançar para conseguirem manter um resfriamento adequado, esse limite é chamado de barreira de potência.

Para solucionar isto, a indústria adotou o paradigma de multiprocessamento em chip, que consiste em vários núcleos de processamento em um mesmo processador trabalhando conjuntamente de forma paralela. Desta forma, foi possível manter o crescimento do desempenho, sem aumentar significativamente a frequência. Mas após o grande crescimento dos dispositivos alimentados por bateria, o consumo energético se tornou mais importante ainda, e apenas a técnica de multiprocessadores já não era mais suficiente. Neste momento, eficiência energética acabou se tornando um dos focos principais dentro do mundo de sistemas e arquiteturas de computadores.

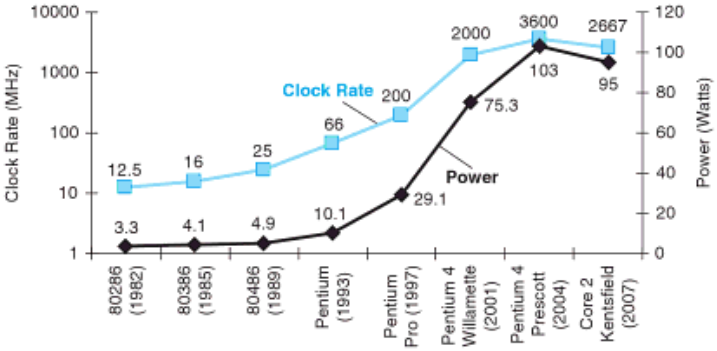


Figura 3 – Potência X Frequência.

1.2.2 Responsáveis

A tecnologia predominante na fabricação de circuitos integrados é chamada de CMOS. Esta tecnologia é responsável por uma dissipação de potência elétrica, e podemos dividí-la em dois tipos diferentes: dissipação estática e dissipação dinâmica.

$$P = P_e + P_d$$

A dissipação de potência estática acontece por imperfeições na parte física do circuito, que são intrínsecas desta tecnologia. Ela acontece quando há vazão de corrente da fonte de energia (V_{dd}) para o *ground* (*GND*). E a vazão de corrente é inversamente relacionada à dimensão dos transistores usados, quanto menores os transistores, maior a corrente, logo maior a potência dissipada. Como estamos aumentando o número de transistores em um sistema, e diminuindo sua dimensão, com o objetivo de melhor desempenho e tamanhos menores para os sistemas, a potência estática vem ocupando uma parcela considerável da potência total (HENNESSY; PATTERSON, 2012). Porém, a potência dinâmica ainda é a mais importante entre elas.

Com em torno de 70-90% da dissipação total, a potência dinâmica acontece na transição de estado lógico. Quando está havendo a troca de estado, o caminho V_{dd}-*GND* se fecha por alguns instantes, gerando corrente de curto-circuito. Além disso, existe uma quantia de energia que precisa ser

drenada da fonte para carregar a capacitância que mantém o estado lógico do circuito. Assim podemos notar que essa potência está relacionada com a atividade do circuito. Os outros fatores deste gasto são a capacitância e a tensão aplicada, como descrito na fórmula abaixo.

$$P_d = \delta \times \text{capacitância} \times V_{dd}^2 \times \text{frequência}$$

Onde δ é uma constante relacionada com a atividade de transição do circuito.

Explicado o que é dissipação de potência, podemos falar do gasto energético, que não deve ser confundida com dissipação de potência. O gasto energético é a potência dissipada em um certo intervalo de tempo, como na fórmula abaixo.

$$E = \int_0^T P(t)dt$$

Ou seja, para reduzirmos o gasto de energia causado por um certo algoritmo, precisamos reduzir seu tempo de computação, ou então reduzir a dissipação de potência gerada durante a execução. Porém uma análise ingênua pode dizer que aumentar o tempo de execução sempre irá aumentar o gasto de energia, mas isso não é verdade. Existem técnicas de redução de potência que aumentam o tempo de execução, mas de forma que o gasto energético ainda seja reduzido.

Antes de mostrarmos uma dessas técnicas, que faz parte do escopo deste trabalho, precisamos citar alguns subsistemas presentes na maioria dos embarcados, e suas contribuições na dissipação de potência do sistema total.

Podemos encontrar muitos modelos de consumo de energia na literatura, e muitos deles concordam em assumir que os maiores consumidores de energia em um sistema embarcado são o subsistema de memória, e o processador, como vemos em (STEINKE et al., 2001)(LI; HENKEL, 1998). Outros possíveis consumidores são os subsistemas de I/O e as interconexões, que normalmente ocupam uma parcela menos significativa do consumo total.

Em (TIWARI et al., 1996) podemos ver que uma instrução que usa apenas os registradores do processador sempre consome menos energia que uma instrução que precise acessar a memória, mesmo no caso da informação estar presente na cache. Neste caso, a diferença do consumo não é tão grande, mas caso a informação não esteja na cache, seja ela de dados ou de instrução, o gasto é muito mais significante. Isto ocorre pelo aumento da corrente necessária ao buscar um dado na memória principal, e a quantidade de ciclos necessária para a instrução ser buscada. Ou seja, podemos perceber que reduzir a frequência em que o sistema precisa acessar a memória principal pode ser uma otimização muito interessante quando estamos querendo uma melhor

eficiência energética.

Algumas pesquisas recentes dão ainda mais certeza de que o subsistema de memória é um local crítico quando queremos reduzir o gasto energético, como podemos ver em (DALLY et al., 2008), que mostra que em média 70% do gasto total de energia é proveniente da busca e escrita de dados e instruções na memória principal. E além disso, suprir uma instrução aritmética custa em torno de 15 a 50 vezes o custo de realmente executar a instrução.

Com isso, foi escolhido como escopo deste trabalho, o subsistema de memória a fim de tentar diminuir os gastos energéticos gerados por este. Ainda assim, podemos notar dois tipo de otimizações possíveis, técnicas de síntese de memória focando a aplicação alvo, isto é, o foco da otimização é o sistema de memória, ou otimizações de código, a qual o foco é o programa que será executado. A primeira classe é interessante quando sabemos que o sistema deverá executar uma única aplicação, o que é muito comum para os sistemas embarcados. Já a segunda classe é necessária quando o sistema alvo for responsável por suprir várias funcionalidades, como os dispositivos móveis, onde o sistema não pode ser modificado visando apenas uma aplicação. Sendo os dispositivos móveis o alvo deste trabalho, iremos trabalhar com a última classe de otimizações citada.

1.2.3 Otimizações de código

As otimizações de código podem impactar as memória de instruções e/ou de dados (HENKEL; PARAMESWARAN, 2007). Normalmente quando otimizações de desempenho são aplicadas, o número de instruções acessadas são reduzidas, assim reduzindo o consumo causado pela memória de instruções. Assim sendo, otimizações de desempenho costumam reduzir o gasto energético também. Mas podemos também tentar reduzir o número de vezes que a memória de dados é acessada, tentando melhorar a localidade dos dados, isto é, aumentando a porcentagem de vezes em que os dados são encontrados na cache ao invés de na memória.

Como será demonstrado melhor no próximo capítulo, algoritmos de criptografia simétrica, neste caso o AES, costumam requerer uma alta aleatoriedade espacial e temporal dos dados que está trabalhando. Isto faz com que seja muito difícil conseguir melhor localidade dos dados. Assim sendo, as otimizações que escolhemos e aplicamos são otimizações que se destinam a obter uma melhor relação de acertos na cache de instruções. Duas otimizações foram selecionadas, são elas: o *Loop unrolling* e *Loop pipelining*.

1.2.3.1 Loop unrolling

Uma das técnicas usadas para melhorar o desempenho de um processador, é chamada de *pipeline*. Em resumo, está técnica divide o processador em diferentes etapas de processamento, e assim mais de uma instrução pode estar sendo executada por vez no processador, pois quando uma está passando por uma parte do processador, as outras estão livres para receber outras instruções. Maiores explicações de um pipeline podem ser vistas em (HENNESSY; PATTERSON, 2012).

Porém, algumas instruções dependem do resultado produzido por outra instrução, e assim pode ser que essas instruções precisem esperar alguns ciclos de relógio até que seu dado esteja pronto. Para reduzir este problema, podemos reordenar as instruções, de forma a diminuir o número de ciclos em que o processador precisa ficar ocioso.

Para conseguirmos explorar mais ainda esta técnica de reordenamento, foi criada a técnica de *loop unrolling*, que nada mais é do que replicar o conteúdo de um loop algumas vezes, ajustando a condição de terminação do loop. Isto é interessante, pois podemos então reordenar instruções que pertenciam a diferentes iterações anteriormente, já que não há mais um desvio entre elas.

Não só isso, o loop unrolling reduz o *overhead* de instruções relacionadas ao desvio, pois teremos menos iterações, ou seja, menos instruções de incremento e desvio a serem executadas. E como explicado anteriormente, menos instruções significa menos acessos a cache de instruções, e eventualmente menos acessos a memória principal, reduzindo assim o gasto energético.

Por outro lado, a técnica de *loop unrolling* aumenta o tamanho do código conforme o número de laços desenrolados. Isto é uma grande desvantagem desta técnica, já que ela pode causar um alto crescimento da taxa de miss na cache de instruções, prejudicando a eficiência energética.

1.2.3.2 Loop pipelining

A técnica de *loop pipelining*, ou *software pipelining*, é uma técnica que busca melhorar o uso do pipeline, assim como o loop unrolling. A ideia principal é fazer com que instruções de diferentes iterações sejam executadas ao mesmo tempo, para explorar ao máximo o paralelismo disponível. Pode ser necessária a criação de um preâmbulo e pós-âmbulo ao corpo do loop, aumentando um pouco a quantidade de instruções, porém o ganho com o número de ciclos economizados dentro do loop pode ser muito recompensante.

É interessante perceber que o loop unrolling pode ser feito antes do

loop pipelining para criar mais possibilidades de reordenamento e intercalação de instruções de diferentes iterações. Mais informações sobre o loop pipelining e outras otimizações de código podem ser vistas em (MUCHNICK, 1997).

Como essas técnicas trabalham com a quantidade de iterações que irão ocorrer, seja executando mais de uma iteração por loop, ou executando iterações fora do corpo do loop, precisamos saber a quantidade de iterações que seriam executadas por aquele loop, para ajustar corretamente a condição de desvio. Desta forma, se um algoritmo em que não é possível saber a quantidade de iterações, essas otimizações não podem ser aplicadas.

1.3 CONTRIBUIÇÕES DESTA MONOGRAFIA

A realização deste trabalho permitiu a criação de um *overview* do gasto energético do algoritmo de cifragem do AES em subsistemas de memória equivalentes ao de uma arquitetura ARMv7-A. Não foi encontrado na literatura nada que comparasse energeticamente diferentes implementações e otimizações do AES, e esse trabalho pode ser usado como uma referência básica para esta questão.

Apesar de este trabalho ter sido realizado para um certo conjunto de tamanhos de cache, ele mostra tendências do gasto energético para outros tamanhos, para vários tipos de implementação.

Outra contribuição foi uma implementação inédita do cifrador do AES em ARM, contando com a otimização de loop pipelining, que teve gastos semelhantes ou levemente superiores que a implementação estado da arte do OpenSSL. Tal resultado supõe um lower bound energético da função de cifragem para os tamanhos de cache testados, por conta da simplicidade dessas implementações.

Por último, ainda foi criada uma plataforma de testes interessante para vários outros possíveis trabalhos, que só necessitam de um pequeno esforço para as alterações desejadas. Alguns exemplos de possíveis trabalhos que podem explorar esta plataforma seriam: realizar testes para diferentes tamanhos de cache, testar outras implementações, trocar a arquitetura alvo.

Essas contribuições ficaram mais claras no decorrer do trabalho, que está apresentado conforme descrito na próxima seção.

1.4 ORGANIZAÇÃO DESTA MONOGRAFIA

Após este primeiro capítulo introdutório, o restante do trabalho está estruturado da seguinte maneira: O capítulo 2 apresenta a criptografia simétrica e um exemplo dela, o AES, o algoritmo escolhido como foco deste trabalho. Após explicado o funcionamento do AES, no capítulo 3 é apresentada uma otimização de código que foi aplicada de forma inédita sobre o AES. O capítulo 4 descreve toda a infraestrutura que foi usada para realizar os experimentos e quais foram os códigos testados, e em seguida, o capítulo 5 contém os resultados e as análises realizadas. Por fim, o capítulo 6 traz as conclusões obtidas, e possíveis trabalhos futuros.

2 CRIPTOGRAFIA E O ALGORITMO AES

A segurança computacional é uma área bastante ampla, a qual envolve conceitos de redes, direito, matemática e elétrica, por exemplo. Uma das áreas mais importantes dela, é a criptografia, responsável pela transformação da informação em algo secreto, diferente da informação original.

A palavra criptografia vem da união dos fragmentos "cripto", que significa secreto, e "grafia", que significa escrita, ou seja, é a escrita de forma secreta. Acredita-se que a criptografia tenha surgido juntamente com a escrita, mas os primeiros vestígios reais dela foi no Egito antigo, onde algum egípcio usou alguns hieróglifos diferentes dos normais (BORGHOFF, 2010). Desde aquela época, a confidencialidade é a principal função da criptografia, ou seja, quando se precisa de comunicação segura, onde pessoas precisam se comunicar sem que terceiras consigam saber o que elas estão conversando. Não só para este objetivo, mas atualmente, com a tecnologia da informação, a criptografia também é necessária em outros tópicos, como integridade da informação, autenticação e não-repúdio (MENEZES; OORSCHOT; VANSTONE, 2010). Neste trabalho, o foco é a criptografia para confidencialidade, que ainda pode ser dividida em duas áreas: simétrica e assimétrica.

2.1 CRIPTOGRAFIA SIMÉTRICA

A criptografia simétrica se resume a duas funções. Uma função de cifragem C que transforma um texto t em uma cifra $c = C(t)$, e uma função de deciframento $D = C^{-1}$, que reconstitui a mensagem original, ou seja, $t = D(c)$. Para manter a confidencialidade, não deve ser possível descobrir t, sabendo de c, a menos que se conheça a função D. Isso acontece, para que dois indivíduos que conheçam as funções C e D consigam se comunicar sem que qualquer outro indivíduo que esteja ouvindo a conversa possa saber o que eles estão dizendo. Porém, esta abordagem faz com que sejam necessárias muitas funções para que os indivíduos conversem secretamente, o que é um grande problema, pois estas funções não são triviais de serem construídas. Tendo esta dificuldade em mente, kerckhoffs criou um teorema conhecido por princípio de kerckhoffs, que diz que *um sistema de criptografia deve ser seguro mesmo que tudo seja público, exceto a chave secreta* (KERCKHOFFS, 1978). O conceito de chave secreta vem com o intuito de solucionar o problema da criação de diferentes funções de cifragem para cada comunicação. Define-se então funções C_k e D_k parametrizadas, tal que $t = D_k(C_{k'}(t)) \iff k = k'$. Dessa forma, as funções podem ser conhecidas

por qualquer um, e somente a chave escolhida deve ser secreta.

Por último, ainda é importante dizer que os cifradores simétricos podem ser divididos em duas classes: os cifradores em bloco, e os cifradores em cadeia. Os cifradores em bloco cifram as mensagens, dividindo elas em blocos de um tamanho fixo de bits, e cifrando cada bloco separadamente através da execução repetida de rounds de transformação, e a maioria deles são ou cifradores de Feistel (Figura 4), como o DES (FIPS, 1999) , ou redes de substituição e permutação (Figura 5). Já os cifradores em cadeia vão recebendo os bits da mensagem e cifrando eles de acordo com um estado interno do algoritmo. Vale salientar que normalmente os cifradores em blocos, utilizam a saída do bloco anterior, para cifrar o próximo bloco, funcionando assim de uma forma similar aos cifradores em cadeia. Essa característica adicional faz parte de um modo de operação, que é uma camada de segurança aplicada em cima dos cifradores.

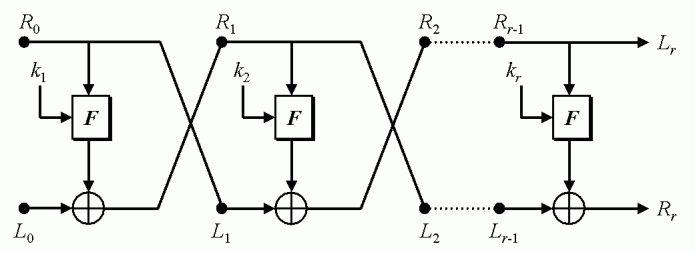


Figura 4 – Estrutura de Feistel.

Depois desta breve introdução da criptografia simétrica, será apresentado agora o algoritmo de criptografia chamado AES, um cifrador simétrico, em bloco e baseado em redes de substituição e permutação, padrão adotado pelo National Institute of Standards and Technology (NIST) (PUB, 2001).

2.2 O ALGORITMO AES

O algoritmo AES, é o algoritmo padrão em vários países para comunicação segura proposto pelo NIST, estabelecido em 2001 através de um concurso organizado por este instituto. O NIST aconselha que este algoritmo deve ser usado pelos órgãos governamentais dos Estados Unidos quando alguma informação sensível deve ser criptografada em função do sigilo da mesma. Por esta garantia dada por um órgão de tamanha importância, o AES

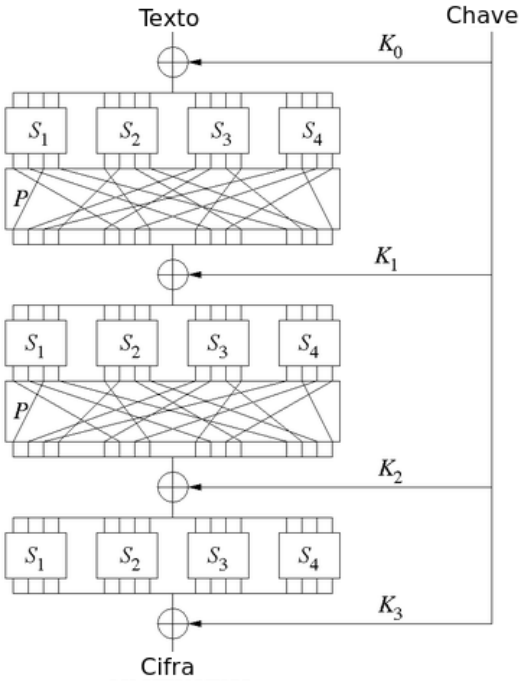


Figura 5 – Rede de Substituição e Permutação.

é um dos cifradores simétricos mais importantes atualmente.

O ganhador do concurso proposto pelo NIST, e escolhido para ser o AES, se chama Rijndael, criado por dois pesquisadores belgas, chamados Vincent Rijmen e Joan Daemen (DAEMEN; RIJMEN, 1998). Após escolhido, o Rijndael sofreu duas pequenas modificações para se tornar o AES: a fixação do tamanho do bloco em 128 bits, e do tamanho da chave em 128, 192 ou 256 bits, tendo 11, 13 ou 15 sub-chaves, respectivamente.

As sub-chaves usadas no processo de cifragem e deciframento do AES, são obtidas a partir de uma expansão sobre a chave secreta escolhida, mas este processo não será abordado neste trabalho, por se tratar de um processo executado apenas uma vez durante todo o processo de cifragem de todos os blocos da mensagem, e que ocupa uma parcela pequena do gasto energético ocasionado pela execução do AES , o foco deste trabalho. Já quanto às funções de cifragem e deciframento, por serem muito semelhantes, apresentarei aqui, apenas a função de cifragem, sobre a qual todos os experimentos foram reali-

zados. De fato, a função de deciframento é a mesma que a de cifragem, porém com as sub-chaves aplicadas no sentido inverso. O algoritmo de cifragem está definido a seguir.

O AES, trabalha com blocos de 32 bits, e os resultados parciais obtidos depois de cada round são chamados de estado. O estado pode ser visto como uma matriz de 4 linhas por 4 colunas de bytes. Vamos denotar o texto do bloco como

$$t_0 t_1 t_2 t_2 ... t_{15},$$

onde t_0 é o primeiro byte do bloco, e t_{15} o ultimo. A cifra é denotada de forma análoga por

$$c_0 c_1 c_2 c_2 ... c_{15}.$$

O estado é definido por

$$a_{i,j}, 0 \leq i, j < 4,$$

onde $a_{i,j}$ é o byte na linha i e coluna j. O mapeamento dos bytes de entrada para o estado inicial é feito pela seguinte fórmula:

$$a_{i,j} = t_{i+4j}, 0 \leq i, j < 4.$$

No final da cifragem, a cifra é obtido usando a seguinte conversão:

$$c_i = t_{i \bmod 4, i/4}, 0 \leq i < 16.$$

O algoritmo consiste em uma execução de N_r rounds, baseado no tamanho da chave, variando entre 10, 12 e 14 rounds. O primeiro passo do algoritmo é a execução do método AddRoundKey, usando a $key[0]$, depois a execução de $N_r - 1$ Rounds, onde o round i utiliza a $key[i]$. E por último a execução de um FINALROUND, a qual utiliza a chave $key[N_r]$. O algoritmo em alto nível pode ser visto em AESENCRYPT.

AESENCRYPT($Estado, key$)
1 AddRoundKey($Estado, key[0]$)
2 **for** $i = 1$ **to** $Nr - 1$
3 Round($Estado, key[i]$)
4 FinalRound($Estado, key[Nr]$)

O procedimento de Round é composto por uma sequência de funções simples e reversíveis chamadas de *step*, que serão explicadas mais adiante. Este procedimento está mostrado em ROUND. Já o FINALROUND, é praticamente igual, porém sem o *step* MixColumn.

ROUND($Estado, RoundKey$)

1 SubBytes($Estado$)
2 ShiftRows($Estado$)
3 MixColumn($Estado$)
4 AddRoundKey($Estado, RoundKey$)

FINALROUND($Estado, RoundKey$)

1 SubBytes($Estado$)
2 ShiftRows($Estado$)
3 AddRoundKey($Estado, RoundKey$)

2.2.1 SubBytes

O SubBytes é a primeira etapa da função de cifragem do AES. Esta etapa consiste na substituição não linear de cada byte do $Estado$. Primeiro, obtém-se a inversa multiplicativa de cada byte considerando que ele pertence ao corpo finito $GF(2^8)$[1]. Após isso, seja $x_0, x_1, ..., x_7$ os bits que compõem um desses bytes, aplica-se uma transformação dada pela função afim a seguir:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Toda essa sequência de passos pode ser substituída por uma tabela de substituição chamada S-Box. Como podemos perceber, esta etapa é equivalente à etapa de substituição das Redes de Substituição e Permutação.

2.2.2 ShiftRows

Na operação de ShiftRows ocorre uma rotação nas linhas 0,1,2 e 3 do $Estado$ em 0,1,2 e 3 bytes para a esquerda. Seja $x_{i,j}$ o byte da linha i e coluna j, a transformação que ocorre é a seguinte:

[1] Explicações sobre corpos finitos podem ser vistos no próprio paper de submissão do Rijndael (DAEMEN; RIJMEN, 1998)

$$ShiftRows(\begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}) = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,1} & x_{1,2} & x_{1,3} & x_{1,0} \\ x_{2,2} & x_{2,3} & x_{2,0} & x_{2,1} \\ x_{3,3} & x_{3,0} & x_{3,1} & x_{3,2} \end{bmatrix}$$

Esta etapa é a que corresponde à permutação das Redes de substituição e permutação.

2.2.3 MixColumn

A operação de MixColumn é a mais computacionalmente cara do processo de cifragem. Novamente cada byte do estado é visto como um elemento de $GF(2^8)$, e cada coluna é vista como um polinômio de terceiro grau. Cada coluna é submetida á uma multiplicação polinomial por um polinômio fixo c(x), módulo um outro polinômio fixo m(x). Estas operações polinomiais com corpos finitos também são explicadas no paper de subimissão do Rijndael (DAEMEN; RIJMEN, 1998), que mostra que essas operações são equivalentes à seguinte multiplicação matricial:

$$MixColumns(\begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}) = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 01 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}$$

Esta etapa é a etapa responsável pela variedade de implementações existentes do AES, que veremos em breve.

2.2.4 AddRoundKey

Por fim temos a etapa de AddRoundKey. Esta é a mais simples das etapas, mas muito importante, e consiste na realização de um XOR dos bytes do estado com os bytes das chaves gerada pela função de expansão da chave secreta escolhida.

2.2.5 Implementações do AES

Apesar de não serem operações simples, algumas dessas etapas podem ser substituída por tabelas de substituição, em vez da operação em si, já que estas operações ocorrem em corpos finitos. Essas modificações são responsáveis por diferentes versões de implementação do AES.

A operação SubBytes transforma um byte em outro byte, isto é, existem exatamente 2^8 possíveis transformações. Assim podemos criar uma tabela de substituição com $2^8 = 256$ bytes. Esta tabela costuma ser usada em todas as versões comuns por ser uma tabela pequena, em vez de termos que realizar operações matriciais. Tal tabela é chamada de S-Box. Então a operação de SubBytes em um estado é simplificada para:

$$SubBytes(\begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}) = \begin{bmatrix} S[x_{0,0}] & S[x_{0,1}] & S[x_{0,2}] & S[x_{0,3}] \\ S[x_{1,0}] & S[x_{1,1}] & S[x_{1,2}] & S[x_{1,3}] \\ S[x_{2,0}] & S[x_{2,1}] & S[x_{2,2}] & S[x_{2,3}] \\ S[x_{3,0}] & S[x_{3,1}] & S[x_{3,2}] & S[x_{3,3}] \end{bmatrix}$$

Em relação à operação ShiftRows, em vez de considerarmos uma rotação dos bytes, podemos selecionar os bytes desejados de cada coluna. Nas arquiteturas 32 bits (também possível em 64 bits), uma coluna do estado será uma word de $4bytes = 32bits$, e apenas é preciso um seleção de bytes de cada word para seguirmos para a próxima operação. Abaixo podemos ver o que seria a seleção da segunda coluna após a realização do ShiftRows.

$$\begin{bmatrix} \mathbf{S[x_{0,0}]} & S[x_{0,1}] & S[x_{0,2}] & S[x_{0,3}] \\ S[x_{1,0}] & \mathbf{S[x_{1,1}]} & S[x_{1,2}] & S[x_{1,3}] \\ S[x_{2,0}] & S[x_{2,1}] & \mathbf{S[x_{2,2}]} & S[x_{2,3}] \\ S[x_{3,0}] & S[x_{3,1}] & S[x_{3,2}] & \mathbf{S[x_{3,3}]} \end{bmatrix}$$

Em seguida, temos a operação de MixColumns, esta qual a mais complexa, e responsável por diferentes versões do AES. Para simplicidade, vamos focar apenas em uma coluna seleciona após ShiftRows. Relembrando que a operação MixColumns considera os bytes como elementos do corpo finito $GF(2^8)$ e que tal operação é dada por:

$$MixColumns(\begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}) = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 01 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}$$

Podemos transformá-la em,

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$S[x_{0,0}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} + S[x_{1,1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} + S[x_{2,2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} + S[x_{3,3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}$$

E então,

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$\begin{bmatrix} S[x_{0,0}] \bullet 02 \\ S[x_{0,0}] \\ S[x_{0,0}] \\ S[x_{0,0}] \bullet 03 \end{bmatrix} + \begin{bmatrix} S[x_{1,1}] \bullet 03 \\ S[x_{1,1}] \bullet 02 \\ S[x_{1,1}] \\ S[x_{1,1}] \end{bmatrix} + \begin{bmatrix} S[x_{2,2}] \\ S[x_{2,2}] \bullet 03 \\ S[x_{2,2}] \bullet 02 \\ S[x_{2,2}] \end{bmatrix} + \begin{bmatrix} S[x_{3,3}] \\ S[x_{3,3}] \\ S[x_{3,3}] \bullet 03 \\ S[x_{3,3}] \bullet 02 \end{bmatrix} \quad (2.1)$$

As operações $+$ e $\bullet$ são as operações de soma e multiplicação em $GF(2^8)$. O + é equivalente a um XOR binário, e $\bullet$ não possui instrução trivial equivalente, por isso diferentes modos de fazê-lo são descritos, como em (GLADMAN, 2001). A implementação destas operações vistas até agora sem nenhum uso de tabela além da S-Box é chamada de versão Sem Tabelas (0T).

Mas observando estas operações do MixColumns, podemos eliminar a operação $\bullet$ criando 4 tabelas de 256 entradas cada, que contém uma word (4 bytes) em cada posição. Isto é, 4 tabelas de 1kB cada. As tabelas criadas são:

$$T_0[x] = \begin{bmatrix} S[x] \bullet 02 \\ S[x] \\ S[x] \\ S[x] \bullet 03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \bullet 03 \\ S[x] \bullet 02 \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \bullet 03 \\ S[x] \bullet 02 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \bullet 03 \\ S[x] \bullet 02 \end{bmatrix}$$

Assim sendo, nossa equação 2.1 se transforma em:

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$T_0[x_{0,0}] + T_1[x_{1,1}] + T_2[x_{2,2}] + T_3[x_{3,3}] \quad (2.2)$$

Esta equação 2.2 é responsável pela versão chamada de Quatro Tabelas (4T). Ela elimina a operação complicada de multiplicação em $GF(2^8)$, porém adiciona 4kB de dados ao método de cifragem.

Agora note que o conteúdo de $T_1[x]$ é exatamente o conteúdo de $T_0[x]$ rotacionado um byte pra baixo, que no caso das words de 32 bits, representa a rotação de 8 bits para a direita (este passo é aproveitado pela instrução ROR presente na arquitetura ARM). O mesmo ocorre para T_2 e T_3, porém com 2 e 3 bytes, respectivamente. Definindo a operação $ror n(x)$ como a rotação de n bits da word x para a direita, podemos transformar a equação 2.2 em:

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$T_0[x_{0,0}] + ror8(T_0[x_{1,1}]) + ror16(T_0[x_{2,2}]) + ror24(T_0[x_{3,3}]) \quad (2.3)$$

Nesta versão não é mais necessário 4 tabelas, e sim apenas uma. Esta versão é chamada de Uma Tabela (1T).

Podemos observar que independente da versão, o número de acessos às tabelas é sempre o mesmo, e o que varia é apenas o tamanho e número de tabelas. Unindo isto ao fato de que o acesso a essas tabelas tem de ser o mais

aleatório possível, as diferente versões devem ter impactos muito distintos na cache de dados. A tendência que esperamos observar é que a 0T seja melhor que a 1T, que por sua vez é melhor que a 4T, por conta da quantidade de possíveis dados que podemos acessar em cada versão.

Por outro lado, esperamos observar o inverso em relação a cache de instruções, já que o número de instruções cresce de 4T para 1T, e 1T para 0T.

Citados as diferentes versões do AES, ainda podemos ter diferentes implementações de cada uma das versões. O OpenSSL implementa algumas delas, e será abordado na seção seguinte.

2.3 OPENSSL

O OpenSSL é um conjunto de ferramentas de código aberto que implementa o protocolo SSL (Secure Socket Layer) e o protocolo TLS (Transport Layer Security) , e oferece uma grande biblioteca de criptografia de propósitos gerais. O SSL é um protocolo de criptografia que tem como objetivo oferecer a comunicação segura nas redes de computadores.

O SSL oferece funções de certificação digital para saber com quem cada parte está se comunicando, criptografia assimétrica para possibilitar o acordo de chaves entre essas partes e criptografia simétrica para haver troca segura de mensagens. Diversos algoritmos são disponibilizados para realizar cada uma dessas funções, e entre os algoritmos implementados para realizar a criptografia simétrica, está o AES. O OpenSSL apresenta mais de uma implementação para o AES, inclusive implementações feitas em código de máquina específico para várias arquiteturas de computadores, como o ARM. Além disso, o OpenSSL também fornece implementações submetidas à otimização de código loop unrolling.

As implementações presentes no OpenSSL que foram usadas neste trabalho são: 4 tabelas com 2 laços desenrolados (4t-2u); 4 tabelas com todos os laços desenrolados (4t-full); e 1 tabela em assembly de ARM (openSSL-1t-asm).

2.4 A OTIMIZAÇÃO DO CÓDIGO DO ALGORITMO AES

Após esta breve introdução sobre cada um dos assuntos relevantes e necessários para o entendimento deste trabalho, é possível apresentar algumas características do algoritmo AES, e a otimização que foi possível desenvolver sobre este algoritmo.

Esta otimização foi realizada em assembly, tentando explorar um para-

lelismo existente entre iterações do algoritmo de cifragem do AES. A motivação e explicação desta implementação pode ser vista na seção seguinte.

2.4.1 Aplicação do loop pipelining

Nas seções anteriores, vimos quais as fases que compõe o algoritmo AES. Podemos perceber que dentro de cada round do AES, existe um grande nível de paralelismo possível e pouca dependência dos dados que estão sendo calculados. Ou seja, existe um grande liberdade de reordenamento de instruções dentro de um round, sem alterar o funcionamento do algoritmo. Mas além desta propriedade, existe um outro nível de paralelismo que pode ser explorado, o paralelismo entre rounds consecutivos.

Foi possível reordenar as instruções de forma a começar a executar uma iteração antes que a iteração corrente terminasse, porém, uma característica intrínseca do algoritmo AES proíbe que uma terceira iteração comece enquanto as duas anteriores não acabarem por completo. Ou seja, a cada dois rounds será necessário uma espera (e possivelmente um esvaziamento do pipeline) até que todas as operações tenham concluído.

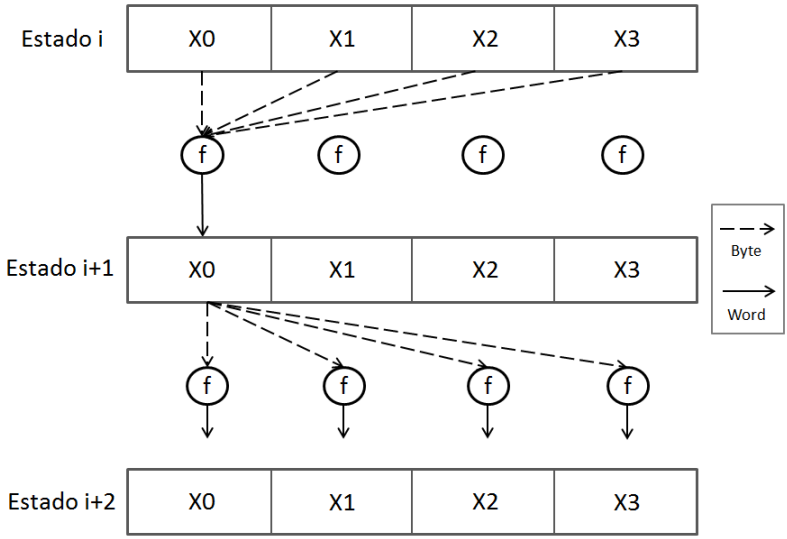


Figura 6 – Paralelismo entre rounds.

Essa integração de iterações é exatamente a técnica de loop pipelining. A ideia está ilustrada na figura 6. A variável X_j representa a coluna j do determinado estado. A função f é a função que constrói uma coluna de um estado $i+1$ a partir do estado i, isto é, ela é a sequência dos passos do round do AES, aplicados aos dados necessários para a criação de uma coluna. Vimos anteriormente que para isso, precisamos de um byte de cada coluna do estado i. Sendo assim, antes de terminarmos de calcular os outros X_j do estado $i+1$, já é possível começar as funções f que criarão as colunas do estado $i+2$.

Porém, mesmo identificando tal paralelismo, note que qualquer um dos X_j do estado $i+2$ só poderá ser concluído assim que todos os X_j do estado $i+1$ também forem. Assim sendo, haverá uma estagnação no final do cálculo do estado $i+2$, e precisaremos esperar ele ser concluído para então podermos começar o estado $i+3$.

Esta ideia foi implementada em assembly, usando a versão de implementação de uma tabela. Vamos chamar tal implementação de 1t-pipe.

3 INFRAESTRUTURA EXPERIMENTAL

Todos os experimentos foram realizados em um simulador de ARM em software. Essa escolha permitiu que fosse possível analisar várias partes do sistema independentemente, o que seria muito difícil em um circuito físico.

3.1 SIMULADOR GEM5

O simulador escolhido para realizar este trabalho foi o Gem5, (BINKERT et al., 2011). O gem5 é um simulador de arquiteturas de sistemas computacionais, e foi desenvolvido por grande parte pesquisadores da universidade de Wisconsin, e suporte de muitas grandes empresas de computação e eletrônica, como a ARM, AMD, HP, Google, Intel, entre outras.

Ele disponibiliza múltiplos modos de CPU, como CPUs in-order e out-of-order. O sistema de memória é implementado a partir de transmissão de eventos, o que permite um grande detalhamento das informações de operações de memória que estão sendo realizadas. Além disso, ele dá suporte a diferentes ISAs, como ARM e x86, e também ao uso de multiprocessadores. Por fim, o gem5 também dá suporte à análise de potência e energia envolvida na execução dos programas executados na plataforma escolhida.

A simulação completa de um processador ARM, mais a possibilidade de observar os detalhes de execução, tanto no processador como no subsistema de memória, são os motivos que levaram a escolher esta plataforma de simulação para a execução deste trabalho.

3.2 FERRAMENTA CACTI

O cacti é uma integração de modelos de tempo de acesso, tempo de ciclo, área, potência estática e dinâmica, em cache e memória. Ela foi desenvolvida pensando em explicitar os ganhos e perdas envolvidas em diferentes configurações de tempo, potência e área. Em outras palavras, é uma ferramenta que, informando-a sobre configurações de um subsistema de memória, cede informações relevantes de tempo, potência e área de um subsistema com essas características.

Uma descrição detalhada da versão que originou o cacti pode ser vista em (WILTON; JOUPPI, 1993). Como as tecnologias usadas nos circuitos digitais mudam muito rapidamente, como por exemplo, a redução da dimensão dos transistores, updates e novas versões do cacti são lançadas com bastante

frequência para manter suas informações fiéis ao estado da arte. Por isso, foi utilizada a versão mais recente existente no momento em que este trabalho foi realizado, o cacti 6.5 (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2009).

Apenas para exemplificar algumas das funcionalidades do cacti, citemos algumas delas. O cacti pode ceder informações do custo energético de um hit ou um miss na cache de nível 1 de 32kB, em um processador com tecnologia de 45nm, executando a 300 Mhz.

3.3 CÓDIGOS TESTADOS

Todas as implementações já citadas que são fornecidas pelo openSSL foram executadas e testadas neste trabalho, isto é, a implementação em C de 4 tabelas com 2 ou todas as iterações do loop desenroladas, e a implementação em assembly do ARMv7 com apenas 1 tabela. Vamos chamar essas implementações de 4t-2u, 4t-full e 1t-asm, respectivamente.

Além destas, foi implementada em C a versão sem tabelas sem desenrolar loops, desenrolando duas iterações, e desenrolando todas as iterações. Vamos chamá-las de 0t, 0t-2u e 0t-full. E também em C, as versões de uma tabela, que chamaremos de 1t, 1t-2u e 1t-full. Estas implementações foram baseadas na descrição de (GLADMAN, 2001). A implementação de uma tabela em assembly, utilizando a técnica de loop pipelining será chamada de 1t-pipe-asm.

3.4 ARQUITETURA DE TESTE

O foco deste trabalho são os dispositivos móveis, como citado anteriormente. Por isso, foi escolhida uma arquitetura e um conjunto de parâmetros que fossem fiéis ao que existe atualmente em relação a dispositivos móveis. Na plataforma Gem5, foi escolhida uma configuração semelhante a um ARM Cortex-A15. Em mais detalhes, foi usado a ISA ARMv7, com um processador de apenas um núcleo operando a 1GHz. O processador é capaz de fazer execução *out-of-order*, i.e., ele é capaz de reordenar as instruções em tempo de execução, a fim de obter melhor desempenho (normalmente de tempo de execução). Além disso, o processador também é capaz de emitir até 3 instruções por ciclo.

Quanto ao subsistema de memória, foi usada apenas caches de nível L1, uma para as instruções e outra para os dados, e uma memória principal do tipo LPDRAM. Os parâmetros usado podem ser vistos na tabela 1.

	Tipo	Capacidade	Blocksize (Bytes)	Associatividade	Tecnologia (nm)
Cache D	SRAM	4,8,16,32 kB	64	2	32
Cache I	SRAM	4,8,16,32 kB	64	2	32
Memória P	LPDRAM	256 mB	-		32

Tabela 1 – Configurações do subsistema de memória.

Os código feitos em C foram compilados usando a versão 4.8 do GCC, a mais atual até o momento. O cuidado em escolher a ultima versão do GCC foi para poder obter as melhores otimizações automáticas do compilador do estado da arte.

Como se trata de um algoritmo de criptografia simétrica, também é necessário escolher um texto para ser cifrado. Para submetermos todas as implementações ao mesmo esforço, um mesmo texto de 16KB foi cifrado em todas elas. A função de expansão de chave e a de deciframento não foram testadas e comparadas. O modo de blocos que foi usado foi o ECB, que apesar de pouco seguro, é uma escolha interessante pois é o modo que minimiza o número de operações entre blocos, assim reduzindo o *overhead* de instruções que não pertencem ao AES.

4 RESULTADOS EXPERIMENTAIS

Citados todos os códigos criados e executados, e qual foi a plataforma de testes escolhida para isso, podemos então analisar os resultados obtidos e discutir sobre a vantagem de cada otimização, seus comportamentos em cada configuração adotada, e também o potencial de cada código testado se executado em diferentes plataformas e configurações.

Antes de começarmos as análises, vamos citar o modelo de energia que foi adotado para a geração dos resultados. Como mostramos anteriormente, o subsistema de memória é o detentor da maior parcela de consumo energético. Assim sendo, foi definido um modelo de energia focado no gasto energético gerado por ele. Vamos dividir o gasto energético em 3 partes, o gasto causado pela cache de dados (cache D), pela cache de instruções (cache I), e pela memória principal (MP).

O gasto causado pela cache D é dado por

$$cacheD = readD + writeD + leakD$$

onde $readD$ e $writeD$ são calculados com

$$readD = nReadD * cReadD$$
$$writeD = nWriteD * cWriteD$$
$$leakD = pLeakD * eTime$$

Os valores $nReadD$, $cReadD$, $nWriteD$, $cWriteD$, $pLeakD$ representam o número de acessos de leitura na cache D, o custo energético de uma leitura na cache D, o número de acessos de escrita na cache D, o custo energético de uma escrita na cache D e pLeakD a potência dissipada estaticamente pela cache D, respectivamente. A variável eTime representa o tempo de execução total do algoritmo na plataforma testada (isto é, não o tempo real que levou para ser executado, mas o tempo que levaria se fosse executado na máquina ARM em questão. As informações de número de acessos são dadas pelo Gem5, e o custo energético e potência estática pela ferramenta CACTI.

O gasto energético gerado pela cache I é calculado de forma similar

$$cacheI = readI + leakI$$

Note que não faz sentido falar sobre escrita na cache I. Esse valores são cal-

culado por

$$readI = nReadI * cReadI$$
$$leakI = pLeakI * eTime$$

Aqui as variáveis tem significado análogo às variáveis da cache D.

E por último, o gasto energético gerado pela MP é dado por

$$memP = readP + writeP + leakP$$

e suas partes são dadas por

$$readP = nReadP * cReadP$$
$$writeP = nWriteP * cWriteP$$
$$leakP = pLeakP * eTime$$

A explicação destas variáveis também é análoga ao da cache D. Aqui é interessante falar que os valores de cReadP e cWriteP chegam a ter duas ordens de grandeza a mais que os custos de leitura e escrita nas caches, ou seja, otimizações de código que consigam reduzir o nReadP e o nWriteP provavelmente irão reduzir o gasto energético gerado pelo código. Esses acessos a memória principal irão potencialmente acontecer quando ocorrer um miss (dado não encontrado) em uma das caches, mas nem sempre que houver um miss irá ocorrer um acesso a MP, pois caso ocorra dois 'misses' muito próximos, para um mesmo bloco de dados, apenas o primeiro enviará uma requisição a MP, ou seja, dois 'misses' geraram apenas um acesso a MP.

E por fim, o gasto total da execução do código será dado por

$$tEnergy = cacheD + cacheI + memP$$

E a eficiência energética será dada em função do texto cifrado

$$E = tEnergy/textSize$$

ou seja, E contém a informação do custo energético por byte cifrado.

Pela grande quantidade de informações e resultados obtidos, vamos primeiramente fazer comparações dentro de cada modo de implementação do AES, isso é, a implementação sem tabelas, com 4 tabelas e com 1 tabela. Após essas análises separadas, vamos fazer uma análise integrada destes modos para concluir qual são as melhores a ser usadas em determinadas situações.

4.1 SEM TABELAS

Como dito anteriormente, as implementações da versão sem tabela do AES foram feitas na linguagem C e compiladas com o GCC 4.8 para a arquitetura ARMv7. As otimizações de loop unrolling realizadas para a versão 0t do AES não apresentaram grandes vantagens em relação a eficiência energética. Na verdade, para todos os tamanhos de cache a implementação sem desenrolar foi melhor ou muito próxima das outras duas, como pode ser visto na figura 7. O mais próximo que alguma otimização chegou em relação a implementação não otimizada foi o full unroll na cache de 32kB, onde a eficiência energética foi a mesma com $2.83nJ/B$.

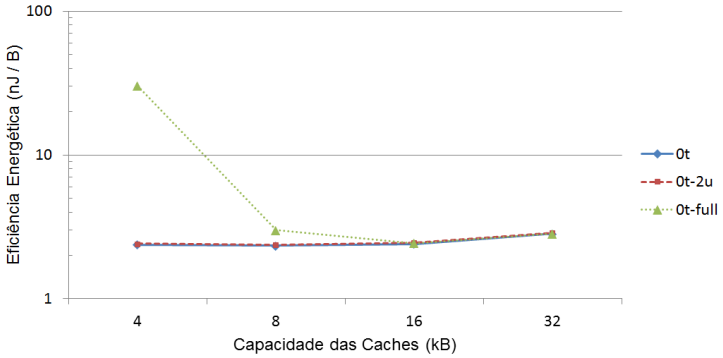


Figura 7 – Comparação das otimizações para diferentes tamanhos de cache - Sem Tabelas.

A versão full unroll foi muito pior na cache de 4kB por causa do número de instruções executadas ser muito grande, levando o miss rate da cache de instruções a valores altíssimos. O código de cifragem do full unroll na linguagem de máquina do ARMv7 tem aproximadamente 1800 instruções, o que dão 7200 bytes de instrução. Isto explica o miss rate de quase 100% para esta configuração, pois a quantidade de bytes desta função é quase o dobro do tamanho da cache, logo, quando uma chamada desta função estiver sendo concluída, a primeira metade dessa função já foi quase toda eliminada da cache I.

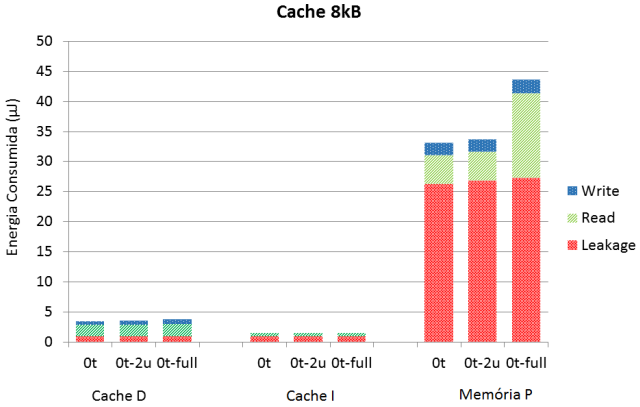


Figura 8 – Gasto energético nas memórias - Sem Tabelas 8kB.

Já nas figuras 8 e 9, podemos ver que não há quase vantagem no uso desta otimização, mesmo em uma cache grande o suficiente para armazenar todas as instruções deste método de cifragem. Podemos também observar a superioridade dos gastos com acesso a caches de 8kB, tanto em relação ao leakage, quanto em relação ao gasto dinâmico, mesmo com as execuções nas caches de 32kB sendo mais rápidas. Uma última observação que gostaria de fazer quanto a esse gráficos, é que na cache de 8kB, a implementação full unroll tem um grande aumento nas leituras feitas na memória principal em relação às outras implementações, isso acontece por causa do número elevado de misses na cache I. As figuras 10a e 10b comprovam esta observação, pois elas apresentam a contribuição energética de cada cache, isto é, elas contêm o gasto gerado por cada cache e pelo gasto gerado por tal cache na memória principal.

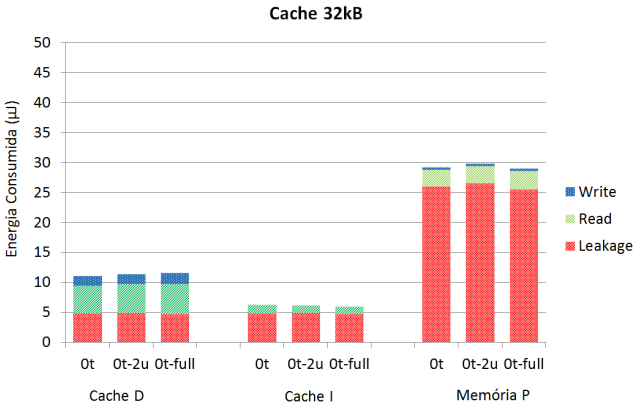
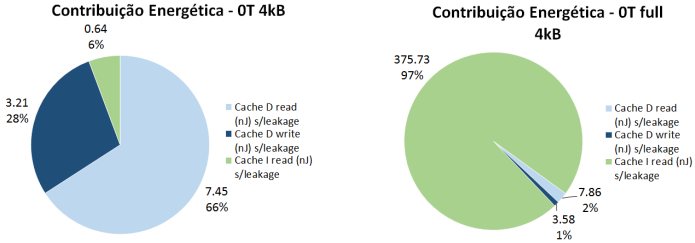


Figura 9 – Gasto energético nas memórias - Sem Tabelas 32kB.

Estas informações mostram que a aplicação do loop unrolling para a versão sem tabelas do AES não é eficaz, pois a versão sem tabelas é a versão que contém o maior número de instruções, já que necessita de um grande número de operações matemáticas, diferente das outras versões que acessam tabelas. Um cenário ótimo para a implementação normal do 0t no sentido do tamanho de caches, seria com uma cache I de tamanho um pouco superior a 1kB, pois o número de instruções de cifragem desta implementação é próximo de 300, o que dá 1200 bytes de instruções.



(a) (b)

Figura 10 – Contribuição energética - Sem Tabelas.

4.2 QUATRO TABELAS

As implementações de 4 tabelas foram em geral melhores que as implementações feitas sem tabela em quase todos os casos. As implementações da versão 4 tabelas foram adquiridas do openSSL. Apesar desta versão ser muito diferente da sem tabelas, o comportamento em relação ao tamanho de caches e otimização usada é muito semelhante. Primeiro note o comportamento em relação ao tamanho das caches na figura 11. A otimização full unroll novamente apresenta uma deterioração na cache de 4kB. Porém podemos perceber uma tendência bem diferente da versão sem tabelas, nesta versão quanto maior a cache, melhor a eficiência energética, ou seja, considerando uma implementação normal, a versão sem tabelas se faz mais vantajosa conforme diminuímos o tamanho das caches. Isto significa que o uso da versão sem tabelas deve ser mais interessante do que a 4 tabelas em sistemas com maiores restrições de energia, como uma rede de sensores.

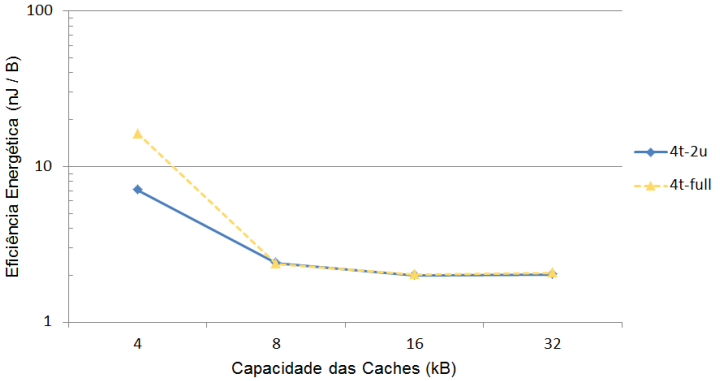


Figura 11 – Comparação das otimizações para diferentes tamanhos de cache - Quatro Tabelas.

O motivo que leva a versão 4 tabelas a ser tão ruim em caches menores é que a quantidade de dados que precisam ser acessados, por causa das 4 tabelas de 1kB cada. Como há a exigência de uma grande aleatoriedade no acesso a estes dados, por se tratar de um algoritmo de criptografia, deve haver pouca ou nenhuma localidade espacial ou temporal a ser explorada, logo não há forma eficiente de mapear estes dados na cache, assim podendo gerar um alto número de misses. Na figura 12, podemos ver o rápido crescimento no gasto energético de leituras da cache D conforme diminuímos o tamanho da

cache.

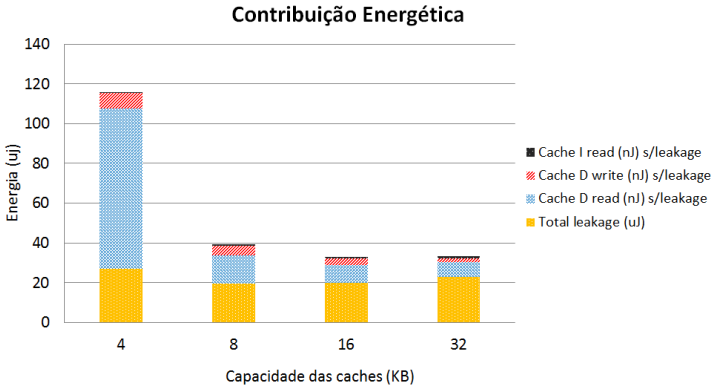


Figura 12 – Contribuição Energética - Quatro Tabelas.

Estas análises mostram que a versão 4 tabelas do AES não é muito interessante pois há muitos dados escritos em memória, e sendo um algoritmo criptográfico, não deve haver localidade no acesso a esses dados, assim, mesmo com caches grandes, existe um alto número de misses na leitura da cache D. Desta forma, um aumento no tamanho das caches acaba não compensando pois o custo de dissipação de potência começa a ser mais significante do que a lenta diminuição do miss rate na cache. E em relação ao full unroll, esta versão novamente apresenta o mesmo problema que a sem tabelas, onde o número de bytes de instruções do bloco básico de cifragem na cache de 4kB se torna quase duas vezes maior que o tamanho da cache, gerando um miss rate muito alto, levando a muitos acessos à memória principal.

4.3 UMA TABELA

Ainda temos a versão de uma tabela do AES para analisarmos. De forma geral, as implementações desta versão foram mais eficientes do que as das outras. Várias implementações foram feitas, e as mais interessantes serão analisadas. Primeiro, vamos analisar o gráfico de comparação das otimizações em relação aos tamanhos de cache, que está apresentado na figura 13. Considerando as implementações feitas em C, em geral foram inferiores, mas podemos ver como o compilador conseguiu explorar melhor suas otimizações conforme o código ia sendo desenrolado. Novamente o full un-

roll foi o pior nas caches de 4kB, mas a partir das caches de 8kB o compilador conseguiu ótimos resultados, se equiparando às implementações realizadas diretamente em assembly, porém, na cache de 32kB ele conseguiu obter o melhor resultado entre todas as implementações, mesmo por uma baixíssima diferença.

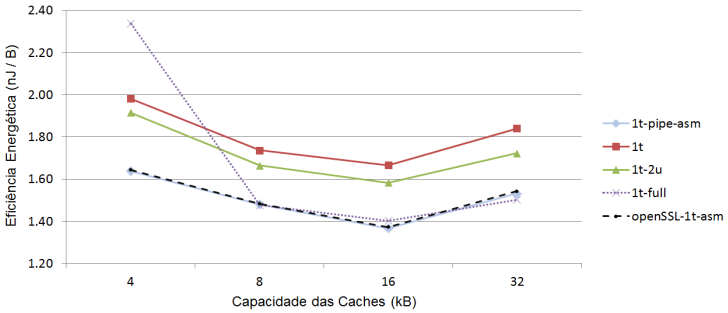
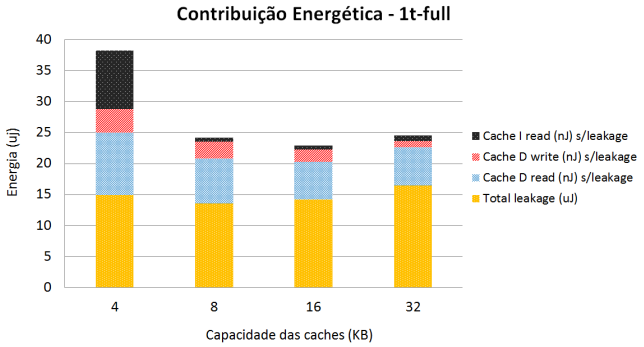
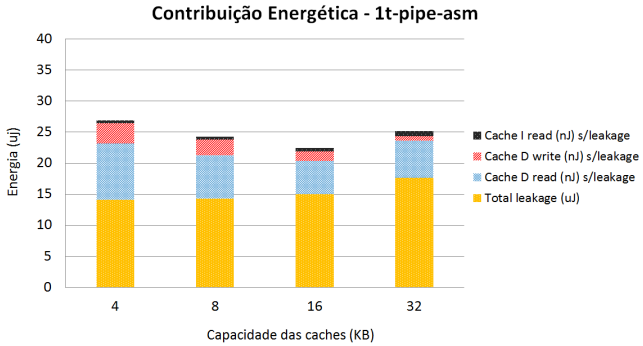


Figura 13 – Comparação das otimizações para diferentes tamanhos de cache - Uma Tabela.

Mas foram as implementações em assembly que obtiveram os melhores resultados. A implementação em assembly de uma tabela do OpenSSL é a implementação que é mais usada para a plataforma ARM. E como o protocolo OpenSSL é um dos mais usados, podemos considerar que esta implementação é o estado da arte do AES para ARM. Já a implementação de 1t-pipe foi feita usando a ideia explicada na seção 2.4. E isso na verdade mostra um resultado interessante. A implementação do 1t-pipe é muito diferente da versão do OpenSSL, ambos feitos em assembly, ou seja, esses códigos não são mais submetidos a nenhuma otimização, e os resultados de eficiência energética alcançados em ambos são praticamente idênticos, em qualquer tamanho de cache, com uma pequena superioridade do 1t-pipe. Isto traz uma das conclusões importantes deste trabalho, de que existe uma forte evidência de que esta eficiência alcançada é muito próxima de um lower bound implícito do cifrador do AES para a plataforma ARM.



(a)



(b)

Figura 14 – Contribuição energética - Uma Tabela.

Nas figuras 14a e 14b é possível perceber que o comportamento e contribuição das caches no gasto energético são muito semelhantes, mesmo sendo um diretamente feito e otimizado em assembly, e o outro programado em C e compilado pelo gcc. Apenas na cache de 4kB vemos uma grande diferença, mas isso acontece pelo grande número de instruções como nas outras implementações com full unroll.

Como veremos na seção a seguir, a versão de uma tabela é bastante superior às outras versões nos tamanhos de caches testados. Isto deve ocorrer pelo fato da versão de uma tabela equilibrar o número de instruções, tendo menos instruções que a versão sem tabelas, e ao mesmo tempo não há menos dados a serem armazenados e acessados, como na versão de 4 tabelas.

4.4 ANÁLISE GERAL

Por fim, vamos fazer uma comparação geral entre as três versões de implementação do AES. Para isso, escolhi a melhor implementação de cada uma das versões para fazermos as análises com um resumo de cada uma das versões. As implementações escolhidas foram a Sem Tabelas normal (0t), a Quatro Tabelas com duas iterações desenroladas (4t-2u), e a implementação da Uma Tabela em assembly com a aplicação do loop pipeline (1t-pipe-asm).

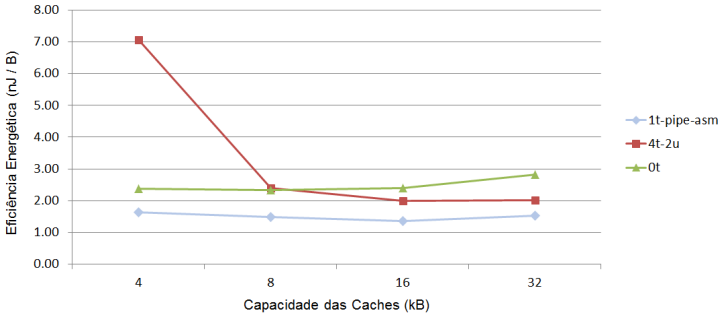


Figura 15 – Comparação das otimizações para diferentes tamanhos de cache - Análise Geral.

Na figura 15 estamos comparando as melhores implementações para cada tamanho de cache. Como foi dito anteriormente, a implementação de uma tabela foi superior às outras em todos os tamanhos de cache. A versão de 4 tabelas se aproxima bastante da versão de uma tabela conforme o tamanho das caches aumentam, porém 32kB já é um tamanho de cache bem alto para os dispositivos móveis da atualidade. E por outro lado, a implementação da versão sem tabelas mostra um comportamento interessante, é a única implementação que demonstra uma melhora na eficiência energética conforme diminuímos o tamanho das caches. Isto pode fazer com que em tamanhos ainda menores de caches a implementação sem tabelas supere a versão de uma tabela.

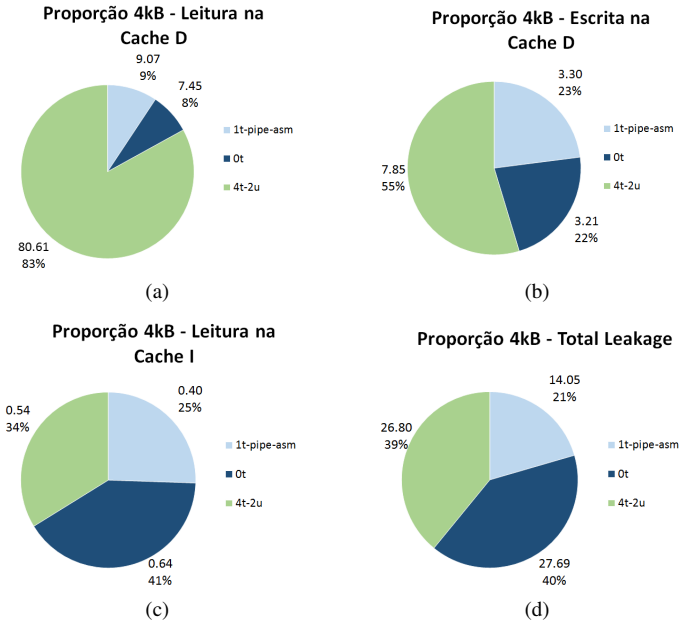


Figura 16 – Contribuição energética - Análise Geral.

Para uma comparação mais detalhada, vamos observar as figuras 16a, 16b, 16c e 16d. Estas figuras relacionam o gasto absoluto de energia por cada cache, e também pelo leakage total, de cada uma das versões, fixando o tamanho da cache em 4kB. Como era esperado, a versão sem tabelas vence as outras quando analisamos a cache D, porém perde na cache I. Mas o que realmente faz com que a 0t não ganhe da 1t é o maior tempo de execução, que eleva muito o leakage em relação ao 1t. E também, como esperado, podemos ver que a grande fraqueza da versão de 4 tabelas está na cache D. Assim sendo, podemos concluir que em geral a versão de 1 tabela é semelhante ou superior que as outras versões, mas com tamanhos menores de cache, a versão sem tabelas também pode ser interessante.

5 CONCLUSÕES E TRABALHOS FUTUROS

Com esta pesquisa foi possível obter um amplo *overview* do gasto energético de diferentes instâncias de implementação do AES para a ISA ARMv7, arquitetura muito comum em dispositivos celulares. Verificamos que a implementação mais usada do OpenSSL, isto é, a implementação de Uma Tabela diretamente em assembly e otimizado para tempo de execução, é superior a praticamente todas as implementações e configurações testadas no escopo deste trabalho. Uma diferente implementação usando a otimização loop pipelining foi proposta, desenvolvida e testada também diretamente em assembly, gerando um código muito diferente da versão padrão do OpenSSL. Os resultados obtidos com tal otimização foram muito parecidos com a do OpenSSL, porém, levemente superiores com menos de 1% de melhora. Este resultado é um forte indício de que estas implementações tem eficiência energética muito próxima ou equivalente a um lower bound para estas configurações de cache.

Outra conclusão importante que pudemos obter é que a versão Sem Tabela é a única versão que teve a eficiência energética melhorada enquanto reduzíamos o tamanho das caches. Este resultado é muito interessante caso fossemos pensar em sistemas com restrições mais pesadas de potência, como as redes de sensores, que, por conta do crescimento da exigência pela ubiquidade, são sistemas que estão se tornando cada vez mais presentes, e foco de pesquisas em sistemas de computação. Este tipo de sistema também tem exigências de segurança, e muitas vezes a troca de dados pode ser bastante alta, o que requer o uso de criptografia simétrica, por ser mais rápido e energeticamente mais eficiente que a outra alternativa, a criptografia assimétrica.

5.1 TRABALHOS FUTUROS

Durante todo este trabalho, observamos que há fortes indícios de que não há formas de reduzirmos o gasto gerado pelo método de cifragem do algoritmo AES, mesmo assim, vimos também que a versão Sem Tabelas do AES teve um potencial interessante para menores tamanhos de cache. Essa característica poderia ser analisada com mais profundidade realizando testes para tamanhos de cache menores que 4kB. Estes testes poderiam mostrar confirmar o indício de que a versão Sem Tabelas é mais eficiente em caches menores, e logo em sistemas de altíssimas restrições de potência. Caso estes resultados fossem confirmados, seria possível focar um trabalho de pesquisa em cima desse tipo de implementação para tentar encontrar possibilidades

de otimização de código que reduzissem o gasto energético gerado por esta versão.

Um segundo escopo possível nesta área, seria a busca de melhores desempenhos energéticos visando agora dispositivos com maiores restrições de potência, como as redes de sensores. Como discutido no início desta monografia, o conceito de Internet of Things (IoT) vem ganhando mais espaço nos tópicos de pesquisa computacional. Este conceitos está fortemente ligado ao uso de sensores, e mais amplamente ao uso de sistemas computacionais de baixíssimas capacidades energéticas (muitas vezes até mesmo sistemas sem uma fonte ativa de energia).

Neste mundo de IoT o comportamento dos indivíduos do sistemas em relação a comunicação é bastante diferente do convencional. A comunicação costuma ser muito breve, onde dois indivíduos normalmente precisam entrar em contato para trocar apenas um *confirmado* ou *negado*. Desta forma, a etapa de estabelecimento de comunicação se torna muito mais relevante. Dentro da criptografia, esta etapa costuma ser resolvida com a criptografia assimétrica, a qual é muito mais cara em desempenho energético e de tempo que a simétrica, pois envolve computação matemática pesada, em vez de operações lógicas simples. Ou seja, um trabalho futuro muito interessante é realizar este tipo de análise realizada neste trabalho em sistemas com maiores restrições de potência usando algoritmos de criptografia assimétrica, além de propor novas otimizações buscando melhor eficiência energética.

Como já demonstrado em várias pesquisas, algoritmos convencionais de criptografia assimétrica, como o RSA (STANDARD, 1998) são extremamente pesados, e consomem muita energia. Porém há uma área matemática que está sendo estudada e relacionado ao campo de criptografia simétrica, chamada de curvas elípticas. O uso de curvas elípticas para a criptografia assimétrica tem se mostrado muito mais eficiente do que os métodos convencionais, e vários estudos estão sendo feitos nessa área, como pode ser visto em (GURA et al., 2004) (YAN; SHI, 2006). Sendo este um campo em expansão e bastante promissor na área de segurança e eficiência energética para o mundo de IoT, há motivação mais do que suficiente para se realizar estudos futuros neste segmento, e possibilitar a solidificação do IoT na sociedade.

REFERÊNCIAS

ARM. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition. C, 2012.

ASHRY, A.; SHARAF, K.; IBRAHIM, M. A compact low-power UHF RFID tag. *Microelectronics Journal*, Elsevier, v. 40, n. 11, p. 1504–1513, nov. 2009. ISSN 00262692.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A survey. *Computer Networks*, v. 54, n. 15, p. 2787–2805, out. 2010. ISSN 13891286. Disponível em: <http://dx.doi.org/10.1016/j.comnet.2010.05.010>.

BASSI, A.; HORN, G. Internet of things in 2020: A roadmap for the future. *European Commission: Information Society and Media*, 2008.

BINKERT, N. et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, ACM, v. 39, n. 2, p. 1–7, 2011.

BORGHOFF, J. *Cryptanalysis of Lightweight Ciphers*. Tese (Doutorado) — Technical University of Denmark, December 2010.

BURR, W. E. Selecting the Advanced Encryption Standard. *Security & Privacy, IEEE*, v. 1, n. 2, p. 43–52, 2003.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. 1998.

DALLY, W. J. et al. Efficient Embedded Computing. *Computer*, v. 41, n. 7, p. 27–32, jul. 2008. ISSN 0018-9162. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4563875>.

FIPS, P. 46-3: Data encryption standard (des). *National Institute of Standards and Technology*, v. 25, n. 10, 1999.

GLADMAN, B. A specification for rijndael, the aes algorithm. *at fp. gladman. plus. com/cryptography_technology/rijndael/aes. spec*, v. 311, p. 18–19, 2001.

GURA, N. et al. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In: *Cryptographic Hardware and Embedded Systems-CHES 2004*. [S.l.]: Springer, 2004. p. 119–132.

HENKEL, J.; PARAMESWARAN, S. *Designing embedded processors: a low power perspective*. [S.l.]: Springer, 2007.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2012.

KERCKHOFFS, A. *La cryptographie militaire*. [S.l.]: University Microfilms, 1978.

LI, Y.; HENKEL, J. A framework for estimation and minimizing energy dissipation of embedded HW/SW systems. In: *Proceedings of the 35th annual conference on Design automation conference - DAC '98*. New York, New York, USA: ACM Press, 1998. p. 188–193. ISBN 0897919645. Disponível em: <http://dl.acm.org/citation.cfm?id=277044.277097>.

MARWEDEL, P. Embedded System Design. Springer-Verlag New York, Inc., mar. 2006. Disponível em: <http://dl.acm.org/citation.cfm?id=1121694>.

MENEZES, A. J.; OORSCHOT, P. C. V.; VANSTONE, S. A. *Handbook of applied cryptography*. [S.l.]: CRC press, 2010.

MUCHNICK, S. S. *Advanced compiler design implementation*. [S.l.]: Morgan Kaufmann, 1997.

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. P. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

PILLAI, V. et al. An Ultra-Low-Power Long Range Battery/Passive RFID Tag for UHF and Microwave Bands With a Current Consumption of 700 nA at 1.5 V. *IEEE Transactions on Circuits and Systems I: Regular Papers*, v. 54, n. 7, p. 1500–1512, jul. 2007. ISSN 1057-7122.

POTLAPALLY, N. R. et al. A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols. *Mobile Computing, IEEE Transactions on*, v. 5, n. 2, p. 128–143, 2006.

PUB, N. F. 197: Advanced encryption standard (aes). *Federal Information Processing Standards Publication*, v. 197, p. 441–0311, 2001.

STANDARD, R. C. Rsa public key cryptography standard 1 v. 2.0. *RSA Laboratories, Oct*, v. 5, p. 1, 1998.

STEINKE, S. et al. An accurate and fine grain instruction-level energy model supporting software optimizations. In: *Proc. of PATMOS*. [S.l.: s.n.], 2001.

TIWARI, V. et al. Instruction level power analysis and optimization of software. In: *Proceedings of 9th International Conference on VLSI Design*. IEEE Comput. Soc. Press, 1996. p. 326–328. ISBN 0-8186-7228-5. ISSN

1063-9667. Disponível em:
<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=489624>.

WILTON, S. J.; JOUPPI, N. P. An enhanced access and cycle time model for on-chip caches. Citeseer, 1993.

YAN, H.; SHI, Z. J. Studying software implementations of elliptic curve cryptography. In: IEEE. *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*. [S.l.], 2006. p. 78–83.

ANEXO A – Artigo sobre o TCC

Análise da eficiência energética do cifrador AES submetido a diferentes otimizações

Gabriel Garcia Gava[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Campus Universitário – Trindade – Florianópolis – SC – Brasil

gabrielgarciagava@gmail.com

Abstract. *Security had always been a very significant subject in the history of Computer Science. Indeed, the first eletronic computers came with the purpose of breaking secret messages. For a long time, the research in computer security focused in the security level and execution time. On the other hand, the increasing use of the embedded systems powered by battery created another important constraint: the energy efficiency.*
One of the most used encryption algorithm nowadays is AES. Its structure allow meaningly distinct implementations. This work studies the energy consumption induced by the execution of that algorithm, in its different implementationsand when subjected to different code optimizations. runtime.

Resumo. *Segurança sempre foi um tópico muito importante na história da Computação. De fato, os primeiros computadores eletrônicos surgiram com o objetivo de decifrar mensagens secretas. Por muito tempo, o foco das pesquisas em segurança computacional foi a qualidade da segurança e o em tempo de execução. Por outro lado, o grande crescimento do uso de sistemas embarcados alimentados a bateria fez com que outra restrição se fizesse importante: a eficiência energética.*
Um dos algoritmos de criptografia mais usados atualmente é o AES. Sua estrutura permite implementações significativamente diferentes. Este trabalho estudou o gasto energético induzido ao executar este algoritmo, em suas diferentes implementações e quando submetido a diferentes otimizações de código.

1. Introdução

Os sistemas embarcados estão cada vez mais presentes no dia-a-dia das pessoas. Uma classe deles, os dispositivos móveis, é responsável por uma porcentagem considerável desses sistemas, e uma característica muito importante desses dispositivos, é que eles são alimentados por bateria. Esses equipamentos móveis sofrem fortes restrições de tempo, espaço, e energia. Além dessas restrições, a evolução da internet está direcionando esses sistemas a um ambiente totalmente conectado, o que é comprovado pelo fato que atualmente todos os novos celulares já têm acesso a internet. Isso cria uma nova restrição, tão importante quanto as outras, para esses aparelhos: a comunicação segura.

Porém, um dos maiores desafios para esses sistemas é a incompatibilidade entre os requisitos energéticos para a segurança e a capacidade das baterias atuais.

Essa preocupação se dá pelo fato de que a taxa de comunicação está aumentando mais rápido do que a capacidade das baterias, e essa distância, chamada de *battery gap* entre a necessidade e a disponibilidade está aumentando [Potlapally et al. 2006]. Isso significa que, se a evolução das baterias continuar a mesma, será preciso encontrar contramedidas que possibilitem a continuação do crescimento do uso desses equipamentos móveis de forma segura. Essas medidas para redução de consumo energético não é só importante para reduzir o efeito do *battery gap*, mas também para o uso de dispositivos móveis com capacidades ainda mais limitadas, como as etiquetas RFID [Ashry et al. 2009] [Pillai et al. 2007], as quais também contém requisitos de segurança.

Analisando estes problemas, pode-se notar a necessidade de redução do consumo energético dos protocolos de segurança e algoritmos criptográficos, os quais disponibilizam a comunicação segura requerida. Os algoritmos criptográficos costumam ser divididos em duas classes: os simétricos e os assimétricos. Dentre os cifradores simétricos, há alguns que são mais usados, como por exemplo o AES, 3DES, Blowfish, Serpent, RC4. Neste trabalho, demos foco ao algoritmo AES, um dos mais importantes cifradores simétricos por conter várias garantias de segurança e ser o padrão adotado pelo governo dos Estados Unidos para suas comunicações [Burr 2003]. O objetivo deste trabalho é então, analisar os gastos energéticos causados pelo algoritmo AES no escopo de um dispositivo móvel. Não só obter os gastos sobre as implementações diretas do algoritmo, mas também a ele submetido a diferentes técnicas de otimização de código e, além disso, observar quais características do algoritmo ou do dispositivo móvel podem ser exploradas para obter uma melhor eficiência energética.

1.1. Sistemas embarcados

Um sistema embarcado é um sistema de microprocessadores que são construídos para alcançar objetivos específicos, e são desenvolvidos de forma que não sejam reprogramáveis, definição dada por [Marwedel 2006]. Em [Hennessy and Patterson 2012], pode-se encontrar um definição muito semelhante, mas ainda é colocado que dispositivos embarcados são mais limitados quanto a sofisticação de software e hardware, e que a habilidade de rodar software de terceiros é o que diferencia um computador não embarcado de um embarcado. Os dispositivos móveis conhecidos como *Personal Mobile Device*, ou PMD, como celulares e tablets, são muitas vezes considerados sistemas embarcados, porém eles podem ser reprogramáveis e também permite-se o uso de novos softwares. Isso faz com que os PMDs sejam tratados como um caso especial.

Os sistemas embarcados começaram a ser usados antes mesmo do surgimento dos PCs e são os consumidores majoritários dos microprocessadores atualmente. Eles costumam estar embutidos em vários equipamentos utilizados pelas pessoas no dia-a-dia sem que nem mesmo elas saibam, como em carros, ar-condicionados, e até mesmo em maquinas de lavar roupas. Mas uma classe deles que vem tomando muito espaço atualmente, são os dispositivos móveis, aqueles alimentados à bateria, pois estes trazem o benefício da ubiquidade, ou seja, podem ser usados em qualquer lugar, a qualquer momento, sem a necessidade de estarem conectados a uma rede elétrica. Não é por menos, podemos observar na Figura 1, como o uso dos dispositivos móveis

vêm substituindo os computadores pessoais (PCs).

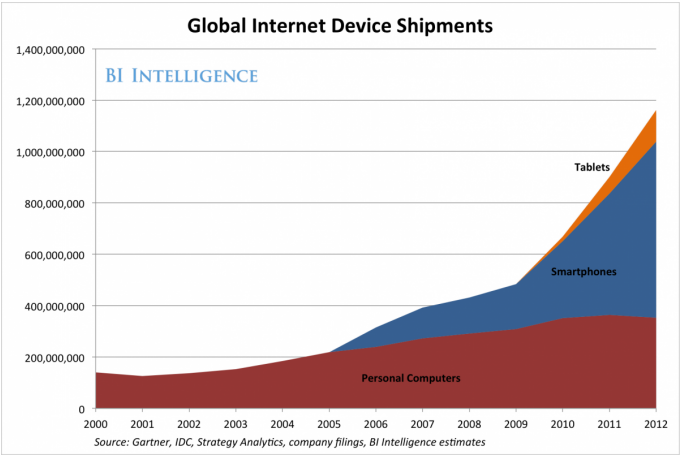


Figura 1. Demanda de dispositivos com acesso a Internet.[1]

Neste trabalho, o enfoque escolhido foram os dispositivos móveis, como tablets e smartphones, já que estes vêm tomando o lugar dos PCs nos dias de hoje. Para isto, foi escolhido a arquitetura ARM como alvo do estudo, visto que este é o processador mais utilizado pelos dispositivos móveis atualmente.

1.2. Eficiência Energética

O conceito de eficiência energética é bastante vasto, e está relacionado à elaboração e execução de técnicas que reduzam a razão gasto energético por algum tipo de recurso, como o número de operações executadas. Sendo este um assunto bastante abrangente, o escopo deste trabalho é a melhora da eficiência energética nos dispositivos alimentados por bateria, que sofrem fortíssimas restrições de potência.

Podemos encontrar muitos modelos de consumo de energia na literatura, e muitos deles concordam em assumir que os maiores consumidores de energia em um sistema embarcado são o subsistema de memória, e o processador, como vemos em [Steinke et al. 2001][Li and Henkel 1998]. Outros possíveis consumidores são os subsistemas de I/O e as interconexões, que normalmente ocupam uma parcela menos significativa do consumo total.

Em [Tiwari et al. 1996] podemos ver que uma instrução que usa apenas os registradores do processador sempre consome menos energia que uma instrução que precise acessar a memória, mesmo no caso da informação estar presente na cache. Neste caso, a diferença do consumo não é tão grande, mas caso a informação não esteja na cache, seja ela de dados ou de instrução, o gasto é muito mais significante.

[1]http://www.businessinsider.com/2013–the-year-ahead-in-mobile-slide-deck-2013-12 Acessado em: 28 de Junho de 2013

Isto ocorre pelo aumento da corrente necessária ao buscar um dado na memória principal, e a quantidade de ciclos necessária para a instrução ser buscada. Ou seja, podemos perceber que reduzir a frequência em que o sistema precisa acessar a memória principal pode ser uma otimização muito interessante quando estamos querendo uma melhor eficiência energética.

Algumas pesquisas recentes dão ainda mais certeza de que o subsistema de memória é um local crítico quando queremos reduzir o gasto energético, como podemos ver em [Dally et al. 2008], que mostra que em média 70% do gasto total de energia é proveniente da busca e escrita de dados e instruções na memória principal. E além disso, suprir uma instrução aritmética custa em torno de 15 a 50 vezes o custo de realmente executar a instrução.

Com isso, foi escolhido como escopo deste trabalho, o subsistema de memória a fim de tentar diminuir os gastos energéticos gerados por este. Ainda assim, podemos notar dois tipo de otimizações possíveis, técnicas de síntese de memória focando a aplicação alvo, isto é, o foco da otimização é o sistema de memória, ou otimizações de código, a qual o foco é o programa que será executado. A primeira classe é interessante quando sabemos que o sistema deverá executar uma única aplicação, o que é muito comum para os sistemas embarcados. Já a segunda classe é necessária quando o sistema alvo for responsável por suprir várias funcionalidades, como os dispositivos móveis, onde o sistema não pode ser modificado visando apenas uma aplicação. Sendo os dispositivos móveis o alvo deste trabalho, iremos trabalhar com a última classe de otimizações citada.

As otimizações de código podem impactar as memória de instruções e/ou de dados [Henkel and Parameswaran 2007]. Normalmente quando otimizações de desempenho são aplicadas, o número de instruções acessadas são reduzidas, assim reduzindo o consumo causado pela memória de instruções. Assim sendo, otimizações de desempenho costumam reduzir o gasto energético também. Mas podemos também tentar reduzir o número de vezes que a memória de dados é acessada, tentando melhorar a localidade dos dados, isto é, aumentando a porcentagem de vezes em que os dados são encontrados na cache ao invés de na memória.

Como será demonstrado melhor no próximo capítulo, algoritmos de criptografia simétrica, neste caso o AES, costumam requerer uma alta aleatoriedade espacial e temporal dos dados que está trabalhando. Isto faz com que seja muito difícil conseguir melhor localidade dos dados. Assim sendo, as otimizações que escolhemos e aplicamos são otimizações que se destinam a obter uma melhor relação de acertos na cache de instruções. Duas otimizações foram selecionadas, são elas: o *Loop unrolling* e *Loop pipelining*[Muchnick 1997] .

2. Criptografia e o algoritmo AES

A segurança computacional é uma área bastante ampla, a qual envolve conceitos de redes, direito, matemática e elétrica, por exemplo. Uma das áreas mais importantes dela, é a criptografia, responsável pela transformação da informação em algo secreto, diferente da informação original.

A palavra criptografia vem da união dos fragmentos "cripto", que significa secreto, e "grafia", que significa escrita, ou seja, é a escrita de forma secreta. Acredita-

se que a criptografia tenha surgido juntamente com a escrita, mas os primeiros vestígios reais dela foi no Egito antigo, onde algum egípcio usou alguns hieróglifos diferentes dos normais [Borghoff 2010]. Desde aquela época, a confidencialidade é a principal função da criptografia, ou seja, quando se precisa de comunicação segura, onde pessoas precisam se comunicar sem que terceiras consigam saber o que elas estão conversando. Não só para este objetivo, mas atualmente, com a tecnologia da informação, a criptografia também é necessária em outros tópicos, como integridade da informação, autenticação e não-repúdio [Menezes et al. 2010]. Neste trabalho, o foco é a criptografia para confidencialidade, que ainda pode ser dividida em duas áreas: simétrica e assimétrica.

2.1. O algoritmo AES

O algoritmo AES é um cifrador simétrico e em blocos. Ele é o algoritmo padrão em vários países para comunicação segura proposto pelo NIST, estabelecido em 2001 através de um concurso organizado por este instituto. O NIST aconselha que este algoritmo deve ser usado pelos órgãos governamentais dos Estados Unidos quando alguma informação sensível deve ser criptografada em função do sigilo da mesma. Por esta garantia dada por um órgão de tamanha importância, o AES é um dos cifradores simétricos mais importantes atualmente.

O ganhador do concurso proposto pelo NIST, e escolhido para ser o AES, se chama Rijndael, criado por dois pesquisadores belgas, chamados Vincent Rijmen e Joan Daemen [Daemen and Rijmen 1998]. Após escolhido, o Rijndael sofreu duas pequenas modificações para se tornar o AES: a fixação do tamanho do bloco em 128 bits, e do tamanho da chave em 128, 192 ou 256 bits, tendo 11, 13 ou 15 sub-chaves, respectivamente.

O algoritmo consiste em uma execução de N_r rounds, baseado no tamanho da chave, variando entre 10, 12 e 14 rounds. O primeiro passo do algoritmo é a execução do método AddRoundKey, usando a $key[0]$, depois a execução de $N_r - 1$ Rounds, onde o round i utiliza a $key[i]$. E por último a execução de um FinalRound, a qual utiliza a chave $key[N_r]$. O algoritmo em alto nível pode ser visto em AESencrypt.

AESencrypt($Estado, key$)
1 AddRoundKey($Estado, key[0]$)
2 **for** $i = 1$ **to** $Nr - 1$
3 Round($Estado, key[i]$)
4 FinalRound($Estado, key[Nr]$)

O procedimento de Round é composto por uma sequência de funções simples e reversíveis chamadas de *step*, que serão explicadas mais adiante. Este procedimento está mostrado em Round. Já o FinalRound, é praticamente igual, porém sem o *step* MixColumn.

Round($Estado, RoundKey$)
1 SubBytes($Estado$)
2 ShiftRows($Estado$)
3 MixColumn($Estado$)
4 AddRoundKey($Estado, RoundKey$)

FinalRound($Estado, RoundKey$)
1 SubBytes($Estado$)
2 ShiftRows($Estado$)
3 AddRoundKey($Estado, RoundKey$)

Apesar de não serem operações simples, algumas dessas etapas podem ser substituída por tabelas de substituição, em vez da operação em si, já que estas operações ocorrem em corpos finitos. Essas modificações são responsáveis por diferentes versões de implementação do AES.

A operação SubBytes transforma um byte em outro byte, isto é, existem exatamente 2^8 possíveis transformações. Assim podemos criar uma tabela de substituição com $2^8 = 256$ bytes. Esta tabela costuma ser usada em todas as versões comuns por ser uma tabela pequena, em vez de termos que realizar operações matriciais. Tal tabela é chamada de S-Box. Então a operação de SubBytes em um estado é simplificada para:

$$SubBytes(\begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}) =$$

$$\begin{bmatrix} S[x_{0,0}] & S[x_{0,1}] & S[x_{0,2}] & S[x_{0,3}] \\ S[x_{1,0}] & S[x_{1,1}] & S[x_{1,2}] & S[x_{1,3}] \\ S[x_{2,0}] & S[x_{2,1}] & S[x_{2,2}] & S[x_{2,3}] \\ S[x_{3,0}] & S[x_{3,1}] & S[x_{3,2}] & S[x_{3,3}] \end{bmatrix}$$

Em relação à operação ShiftRows, em vez de considerarmos uma rotação dos bytes, podemos selecionar os bytes desejados de cada coluna. Nas arquiteturas 32 bits (também possível em 64 bits), uma coluna do estado será uma word de $4bytes = 32bits$, e apenas é preciso um seleção de bytes de cada word para seguirmos para a próxima operação. Abaixo podemos ver o que seria a seleção da segunda coluna após a realização do ShiftRows.

$$\begin{bmatrix} \mathbf{S[x_{0,0}]} & S[x_{0,1}] & S[x_{0,2}] & S[x_{0,3}] \\ S[x_{1,0}] & \mathbf{S[x_{1,1}]} & S[x_{1,2}] & S[x_{1,3}] \\ S[x_{2,0}] & S[x_{2,1}] & \mathbf{S[x_{2,2}]} & S[x_{2,3}] \\ S[x_{3,0}] & S[x_{3,1}] & S[x_{3,2}] & \mathbf{S[x_{3,3}]} \end{bmatrix}$$

Em seguida, temos a operação de MixColumns, esta qual a mais complexa, e responsável por diferentes versões do AES. Para simplicidade, vamos focar apenas em uma coluna seleciona após ShiftRows. Relembrando que a operação MixColumns considera os bytes como elementos do corpo finito $GF(2^8)$ e que tal operação é dada por:

$$MixColumns(\begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}) = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 01 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_{0,0} \\ x_{1,1} \\ x_{2,2} \\ x_{3,3} \end{bmatrix}$$

Podemos transformá-la em,

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$S[x_{0,0}] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} + S[x_{1,1}] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} + S[x_{2,2}] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} + S[x_{3,3}] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}$$

E então,

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) =$$

$$\begin{bmatrix} S[x_{0,0}] \bullet 02 \\ S[x_{0,0}] \\ S[x_{0,0}] \\ S[x_{0,0}] \bullet 03 \end{bmatrix} + \begin{bmatrix} S[x_{1,1}] \bullet 03 \\ S[x_{1,1}] \bullet 02 \\ S[x_{1,1}] \\ S[x_{1,1}] \end{bmatrix} + \begin{bmatrix} S[x_{2,2}] \\ S[x_{2,2}] \bullet 03 \\ S[x_{2,2}] \bullet 02 \\ S[x_{2,2}] \end{bmatrix} + \begin{bmatrix} S[x_{3,3}] \\ S[x_{3,3}] \\ S[x_{3,3}] \bullet 03 \\ S[x_{3,3}] \bullet 02 \end{bmatrix} \quad (1)$$

As operações + e $\bullet$ são as operações de soma e multiplicação em $GF(2^8)$. O + é equivalente a um XOR binário, e $\bullet$ não possui instrução trivial equivalente, por isso diferentes modos de fazê-lo são descritos, como em [Gladman 2001]. A implementação destas operações vistas até agora sem nenhum uso de tabela além da S-Box é chamada de versão Sem Tabelas (0T).

Mas observando estas operações do MixColumns, podemos eliminar a operação $\bullet$ criando 4 tabelas de 256 entradas cada, que contém uma word (4 bytes) em cada posição. Isto é, 4 tabelas de 1kB cada. As tabelas criadas são:

$$T_0[x] = \begin{bmatrix} S[x] \bullet 02 \\ S[x] \\ S[x] \\ S[x] \bullet 03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \bullet 03 \\ S[x] \bullet 02 \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \bullet 03 \\ S[x] \bullet 02 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \bullet 03 \\ S[x] \bullet 02 \end{bmatrix}$$

Assim sendo, nossa equação 1 se transforma em:

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) = T_0[x_{0,0}] + T_1[x_{1,1}] + T_2[x_{2,2}] + T_3[x_{3,3}] \quad (2)$$

Esta equação 2 é responsável pela versão chamada de Quatro Tabelas (4T). Ela elimina a operação complicada de multiplicação em $GF(2^8)$, porém adiciona 4kB de dados ao método de cifragem.

Agora note que o conteúdo de $T_1[x]$ é exatamente o conteúdo de $T_0[x]$ rotacionado um byte pra baixo, que no caso das words de 32 bits, representa a rotação de 8 bits para a direita (este passo é aproveitado pela instrução ROR presente na arquitetura ARM). O mesmo ocorre para T_2 e T_3, porém com 2 e 3 bytes, respectivamente. Definindo a operação $ror n(x)$ como a rotação de n bits da word x para a direita, podemos transformar a equação 2 em:

$$MixColumns(\begin{bmatrix} S[x_{0,0}] \\ S[x_{1,1}] \\ S[x_{2,2}] \\ S[x_{3,3}] \end{bmatrix}) = T_0[x_{0,0}] \ + \ ror8(T_0[x_{1,1}]) \ + \ ror16(T_0[x_{2,2}]) \ + \ ror24(T_0[x_{3,3}]) \quad (3)$$

Nesta versão não é mais necessário 4 tabelas, e sim apenas uma. Esta versão é chamada de Uma Tabela (1T).

Podemos observar que independente da versão, o número de acessos às tabelas é sempre o mesmo, e o que varia é apenas o tamanho e número de tabelas. Unindo isto ao fato de que o acesso a essas tabelas tem de ser o mais aleatório possível, as diferente versões devem ter impactos muito distintos na cache de dados. A tendência que esperamos observar é que a 0T seja melhor que a 1T, que por sua vez é melhor que a 4T, por conta da quantidade de possíveis dados que podemos acessar em cada versão.

Por outro lado, esperamos observar o inverso em relação a cache de instruções, já que o número de instruções cresce de 4T para 1T, e 1T para 0T.

3. A otimização do código do algoritmo AES

Após esta breve introdução sobre cada um dos assuntos relevantes e necessários para o entendimento deste trabalho, é possível apresentar algumas características do algoritmo AES, e a otimização que foi possível desenvolver sobre este algoritmo.

Esta otimização foi realizada em assembly, tentando explorar um paralelismo existente entre iterações do algoritmo de cifragem do AES. A motivação e explicação desta implementação pode ser vista na seção seguinte.

Nas seções anteriores, vimos quais as fases que compõe o algoritmo AES. Podemos perceber que dentro de cada round do AES, existe um grande nível de paralelismo possível e pouca dependência dos dados que estão sendo calculados. Ou seja, existe um grande liberdade de reordenamento de instruções dentro de um round, sem alterar o funcionamento do algoritmo. Mas além desta propriedade, existe um outro nível de paralelismo que pode ser explorado, o paralelismo entre rounds consecutivos.

Foi possível reordenar as instruções de forma a começar a executar uma iteração antes que a iteração corrente terminasse, porém, uma característica intrínseca do algoritmo AES proíbe que uma terceira iteração comece enquanto as duas anteriores não acabarem por completo. Ou seja, a cada dois rounds será necessário uma espera (e possivelmente um esvaziamento do pipeline) até que todas as operações tenham concluído.

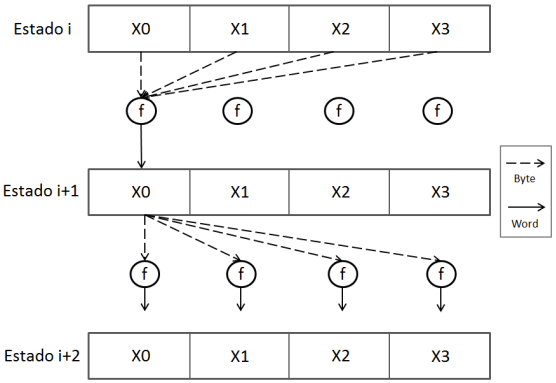


Figura 2. Paralelismo entre rounds.

Essa integração de iterações é exatamente a técnica de loop pipelining. A ideia está ilustrada na figura 2. A variável X_j representa a coluna j do determinado estado. A função f é a função que constrói uma coluna de um estado $i+1$ a partir do estado i, isto é, ela é a sequência dos passos do round do AES, aplicados aos dados necessários para a criação de uma coluna. Vimos anteriormente que para isso, precisamos de um byte de cada coluna do estado i. Sendo assim, antes de terminarmos de calcular os outros X_j do estado $i+1$, já é possível começar as funções f que criarão as colunas do estado $i+2$.

Porém, mesmo identificando tal paralelismo, note que qualquer um dos X_j do estado $i+2$ só poderá ser concluído assim que todos os X_j do estado $i+1$ também forem. Assim sendo, haverá uma estagnação no final do cálculo do estado $i+2$, e precisaremos esperar ele ser concluído para então podermos começar o estado $i+3$.

Esta ideia foi implementada em assembly, usando a versão de implementação de uma tabela. Vamos chamar tal implementação de 1t-pipe.

4. Infraestrutura experimental

Todos os experimentos foram realizados em um simulador de ARM em software. Essa escolha permitiu que fosse possível analisar várias partes do sistema independentemente, o que seria muito difícil em um circuito físico.

4.1. Códigos testados

Todas as implementações já citadas que são fornecidas pelo openSSL foram executadas e testadas neste trabalho, isto é, a implementação em C de 4 tabelas com 2 ou todas as iterações do loop desenroladas, e a implementação em assembly do ARMv7 com apenas 1 tabela. Vamos chamar essas implementações de 4t-2u, 4t-full e 1t-asm, respectivamente.

Além destas, foi implementada em C a versão sem tabelas sem desenrolar loops, desenrolando duas iterações, e desenrolando todas as iterações. Vamos chamá-las de 0t, 0t-2u e 0t-full. E também em C, as versões de uma tabela, que chamaremos de 1t, 1t-2u e 1t-full. Estas implementações foram baseadas na descrição de [Gladman 2001]. A implementação de uma tabela em assembly, utilizando a técnica de loop pipelining será chamada de 1t-pipe-asm.

4.2. Arquitetura de teste

O foco deste trabalho são os dispositivos móveis, como citado anteriormente. Por isso, foi escolhida uma arquitetura e um conjunto de parâmetros que fossem fiéis ao que existe atualmente em relação a dispositivos móveis. Na plataforma Gem5, foi escolhida uma configuração semelhante a um ARM Cortex-A15. Em mais detalhes, foi usado a ISA ARMv7, com um processador de apenas um núcleo operando a 1GHz. O processador é capaz de fazer execução *out-of-order*, i.e., ele é capaz de reordenar as instruções em tempo de execução, a fim de obter melhor desempenho (normalmente de tempo de execução). Além disso, o processador também é capaz de emitir até 3 instruções por ciclo.

Quanto ao subsistema de memória, foi usada apenas caches de nível L1, uma para as instruções e outra para os dados, e uma memória principal do tipo LPDRAM. Os parâmetros usado podem ser vistos na tabela 1.

	Tipo	Capacidade	Blocksize (Bytes)	Associatividade	Tecnologia (nm)
Cache D	SRAM	4,8,16,32 kB	64	2	32
Cache I	SRAM	4,8,16,32 kB	64	2	32
Memória P	LPDRAM	256 mB	-		32

Tabela 1. Configurações do subsistema de memória.

Os código feitos em C foram compilados usando a versão 4.8 do GCC, a mais atual até o momento. O cuidado em escolher a ultima versão do GCC foi para poder obter as melhores otimizações automáticas do compilador do estado da arte.

Como se trata de um algoritmo de criptografia simétrica, também é necessário escolher um texto para ser cifrado. Para submetermos todas as implementações ao mesmo esforço, um mesmo texto de 16KB foi cifrado em todas elas. A função de expansão de chave e a de deciframento não foram testadas e comparadas. O modo

de blocos que foi usado foi o ECB, que apesar de pouco seguro, é uma escolha interessante pois é o modo que minimiza o número de operações entre blocos, assim reduzindo o *overhead* de instruções que não pertencem ao AES.

5. Resultados

Citados todos os códigos criados e executados, e qual foi a plataforma de testes escolhida para isso, podemos então analisar os resultados obtidos e discutir sobre a vantagem de cada otimização, seus comportamentos em cada configuração adotada, e também o potencial de cada código testado se executado em diferentes plataformas e configurações.

5.1. Sem Tabelas

Como dito anteriormente, as implementações da versão sem tabela do AES foram feitas na linguagem C e compiladas com o GCC 4.8 para a arquitetura ARMv7. As otimizações de loop unrolling realizadas para a versão 0t do AES não apresentaram grandes vantagens em relação a eficiência energética. Na verdade, para todos os tamanhos de cache a implementação sem desenrolar foi melhor ou muito próxima das outras duas, como pode ser visto na figura 3. O mais próximo que alguma otimização chegou em relação a implementação não otimizada foi o full unroll na cache de 32kB, onde a eficiência energética foi a mesma com $2.83nJ/B$.

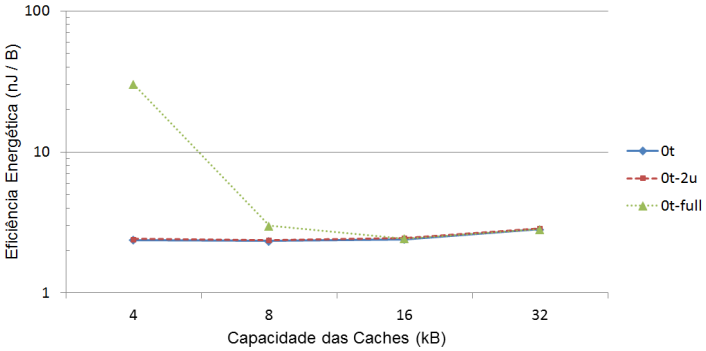


Figura 3. Comparação das otimizações para diferentes tamanhos de cache - Sem Tabelas.

A versão full unroll foi muito pior na cache de 4kB por causa do número de instruções executadas ser muito grande, levando o miss rate da cache de instruções a valores altíssimos. O código de cifragem do full unroll na linguagem de máquina do ARMv7 tem aproximadamente 1800 instruções, o que dão 7200 bytes de instrução. Isto explica o miss rate de quase 100% para esta configuração, pois a quantidade de bytes desta função é quase o dobro do tamanho da cache, logo, quando uma chamada desta função estiver sendo concluída, a primeira metade dessa função já foi quase toda eliminada da cache I.

5.2. Quatro Tabelas

As implementações de 4 tabelas foram em geral melhores que as implementações feitas sem tabela em quase todos os casos. As implementações da versão 4 tabelas foram adquiridas do openSSL. Apesar desta versão ser muito diferente da sem tabelas, o comportamento em relação ao tamanho de caches e otimização usada é muito semelhante. Primeiro note o comportamento em relação ao tamanho das caches na figura 4. A otimização full unroll novamente apresenta uma deterioração na cache de 4kB. Porém podemos perceber uma tendência bem diferente da versão sem tabelas, nesta versão quanto maior a cache, melhor a eficiência energética, ou seja, considerando uma implementação normal, a versão sem tabelas se faz mais vantajosa conforme diminuímos o tamanho das caches. Isto significa que o uso da versão sem tabelas deve ser mais interessante do que a 4 tabelas em sistemas com maiores restrições de energia, como uma rede de sensores.

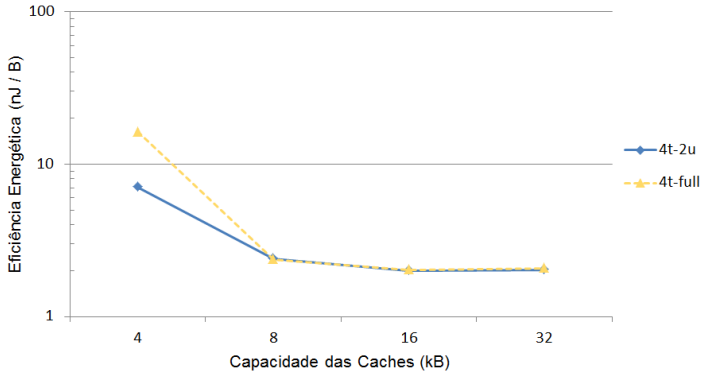


Figura 4. Comparação das otimizações para diferentes tamanhos de cache - Quatro Tabelas.

Estas análises mostram que a versão 4 tabelas do AES não é muito interessante pois há muitos dados escritos em memória, e sendo um algoritmo criptográfico, não deve haver localidade no acesso a esses dados, assim, mesmo com caches grandes, existe um alto número de misses na leitura da cache D. Desta forma, um aumento no tamanho das caches acaba não compensando pois o custo de dissipação de potência começa a ser mais significante do que a lenta diminuição do miss rate na cache. E em relação ao full unroll, esta versão novamente apresenta o mesmo problema que a sem tabelas, onde o número de bytes de instruções do bloco básico de cifragem na cache de 4kB se torna quase duas vezes maior que o tamanho da cache, gerando um miss rate muito alto, levando a muitos acessos à memória principal.

5.3. Uma Tabela

Ainda temos a versão de uma tabela do AES para analisarmos. De forma geral, as implementações desta versão foram mais eficientes do que as das outras. Várias

implementações foram feitas, e as mais interessantes serão analisadas. Primeiro, vamos analisar o gráfico de comparação das otimizações em relação aos tamanhos de cache, que está apresentado na figura 5. Considerando as implementações feitas em C, em geral foram inferiores, mas podemos ver como o compilador conseguiu explorar melhor suas otimizações conforme o código ia sendo desenrolado. Novamente o full unroll foi o pior nas caches de 4kB, mas a partir das caches de 8kB o compilador conseguiu ótimos resultados, se equiparando às implementações realizadas diretamente em assembly, porém, na cache de 32kB ele conseguiu obter o melhor resultado entre todas as implementações, mesmo por uma baixíssima diferença.

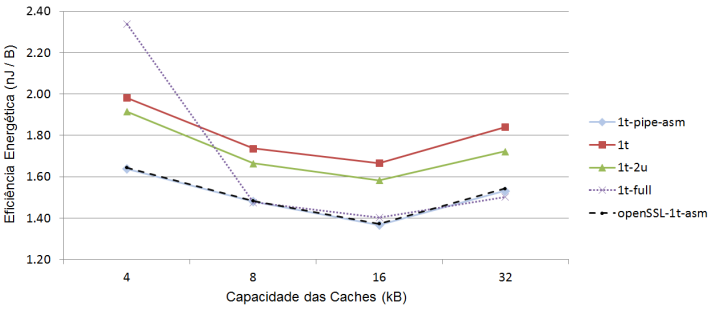


Figura 5. Comparação das otimizações para diferentes tamanhos de cache - Uma Tabela.

Mas foram as implementações em assembly que obtiveram os melhores resultados. A implementação em assembly de uma tabela do OpenSSL é a implementação que é mais usada para a plataforma ARM. E como o protocolo OpenSSL é um dos mais usados, podemos considerar que esta implementação é o estado da arte do AES para ARM. Já a implementação de 1t-pipe foi feita usando a ideia explicada na seção **??**. E isso na verdade mostra um resultado interessante. A implementação do 1t-pipe é muito diferente da versão do OpenSSL, ambos feitos em assembly, ou seja, esses códigos não são mais submetidos a nenhuma otimização, e os resultados de eficiência energética alcançados em ambos são praticamente idênticos, em qualquer tamanho de cache, com uma pequena superioridade do 1t-pipe. Isto traz uma das conclusões importantes deste trabalho, de que existe uma forte evidência de que esta eficiência alcançada é muito próxima de um lower bound implícito do cifrador do AES para a plataforma ARM.

Como veremos na seção a seguir, a versão de uma tabela é bastante superior às outras versões nos tamanhos de caches testados. Isto deve ocorrer pelo fato da versão de uma tabela equilibrar o número de instruções, tendo menos instruções que a versão sem tabelas, e ao mesmo tempo não há menos dados a serem armazenados e acessados, como na versão de 4 tabelas.

5.4. Ánalise Geral

Por fim, vamos fazer uma comparação geral entre as três versões de implementação do AES. Para isso, escolhi a melhor implementação de cada uma das versões para fazermos as análises com um resumo de cada uma das versões. As implementações escolhidas foram a Sem Tabelas normal (0t), a Quatro Tabelas com duas iterações desenroladas (4t-2u), e a implementação da Uma Tabela em assembly com a aplicação do loop pipeline (1t-pipe-asm).

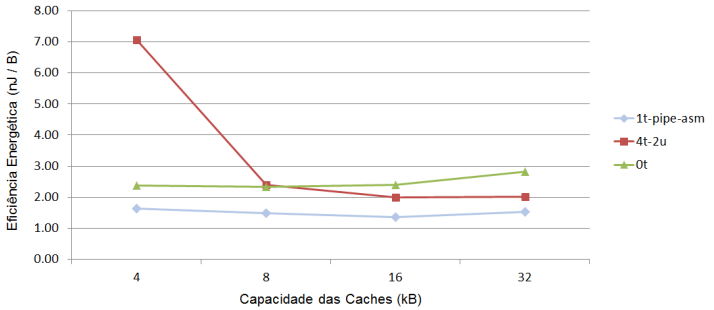


Figura 6. Comparação das otimizações para diferentes tamanhos de cache - Análise Geral.

Na figura 6 estamos comparando as melhores implementações para cada tamanho de cache. Como foi dito anteriormente, a implementação de uma tabela foi superior às outras em todos os tamanhos de cache. A versão de 4 tabelas se aproxima bastante da versão de uma tabela conforme o tamanho das caches aumentam, porém 32kB já é um tamanho de cache bem alto para os dispositivos móveis da atualidade. E por outro lado, a implementação da versão sem tabelas mostra um comportamento interessante, é a única implementação que demonstra uma melhora na eficiência energética conforme diminuímos o tamanho das caches. Isto pode fazer com que em tamanhos ainda menores de caches a implementação sem tabelas supere a versão de uma tabela.

6. Conclusões e Trabalhos Futuros

Com esta pesquisa foi possível obter um amplo *overview* do gasto energético de diferentes instâncias de implementação do AES para a ISA ARMv7, arquitetura muito comum em dispositivos celulares. Verificamos que a implementação mais usada do OpenSSL, isto é, a implementação de Uma Tabela diretamente em assembly e otimizado para tempo de execução, é superior a praticamente todas as implementações e configurações testadas no escopo deste trabalho. Uma diferente implementação usando a otimização loop pipelining foi proposta, desenvolvida e testada também diretamente em assembly, gerando um código muito diferente da versão padrão do OpenSSL. Os resultados obtidos com tal otimização foram muito parecidos com a

do OpenSSL, porém, levemente superiores com menos de 1% de melhora. Este resultado é um forte indício de que estas implementações tem eficiência energética muito próxima ou equivalente a um lower bound para estas configurações de cache.

Outra conclusão importante que pudemos obter é que a versão Sem Tabela é a única versão que teve a eficiência energética melhorada enquanto reduzíamos o tamanho das caches. Este resultado é muito interessante caso fossemos pensar em sistemas com restrições mais pesadas de potência, como as redes de sensores, que, por conta do crescimento da exigência pela ubiquidade, são sistemas que estão se tornando cada vez mais presentes, e foco de pesquisas em sistemas de computação. Este tipo de sistema também tem exigências de segurança, e muitas vezes a troca de dados pode ser bastante alta, o que requer o uso de criptografia simétrica, por ser mais rápido e energeticamente mais eficiente que a outra alternativa, a criptografia assimétrica.

6.1. Trabalhos Futuros

Durante todo este trabalho, observamos que há fortes indícios de que não há formas de reduzirmos o gasto gerado pelo método de cifragem do algoritmo AES, mesmo assim, vimos também que a versão Sem Tabelas do AES teve um potencial interessante para menores tamanhos de cache. Essa característica poderia ser analisada com mais profundidade realizando testes para tamanhos de cache menores que 4kB. Estes testes poderiam mostrar confirmar o indício de que a versão Sem Tabelas é mais eficiente em caches menores, e logo em sistemas de altíssimas restrições de potência. Caso estes resultados fossem confirmados, seria possível focar um trabalho de pesquisa em cima desse tipo de implementação para tentar encontrar possibilidades de otimização de código que reduzissem o gasto energético gerado por esta versão.

Um segundo escopo possível nesta área, seria a busca de melhores desempenhos energéticos visando agora dispositivos com maiores restrições de potência, como as redes de sensores. Como discutido no início desta monografia, o conceito de Internet of Things (IoT) vem ganhando mais espaço nos tópicos de pesquisa computacional. Este conceitos está fortemente ligado ao uso de sensores, e mais amplamente ao uso de sistemas computacionais de baixíssimas capacidades energéticas (muitas vezes até mesmo sistemas sem uma fonte ativa de energia).

Neste mundo de IoT o comportamento dos indivíduos do sistemas em relação a comunicação é bastante diferente do convencional. A comunicação costuma ser muito breve, onde dois indivíduos normalmente precisam entrar em contato para trocar apenas um *confirmado* ou *negado*. Desta forma, a etapa de estabelecimento de comunicação se torna muito mais relevante. Dentro da criptografia, esta etapa costuma ser resolvida com a criptografia assimétrica, a qual é muito mais cara em desempenho energético e de tempo que a simétrica, pois envolve computação matemática pesada, em vez de operações lógicas simples. Ou seja, um trabalho futuro muito interessante é realizar este tipo de análise realizada neste trabalho em sistemas com maiores restrições de potência usando algoritmos de criptografia assimétrica, além de propor novas otimizações buscando melhor eficiência energética.

Como já demonstrado em várias pesquisas, algoritmos convencionais de crip-

tografia assimétrica, como o RSA [Standard 1998] são extremamente pesados, e consomem muita energia. Porém há uma área matemática que está sendo estudada e relacionado ao campo de criptografia simétrica, chamada de curvas elípticas. O uso de curvas elípticas para a criptografia assimétrica tem se mostrado muito mais eficiente do que os métodos convencionais, e vários estudos estão sendo feitos nessa área, como pode ser visto em [Gura et al. 2004] [Yan and Shi 2006]. Sendo este um campo em expansão e bastante promissor na área de segurança e eficiência energética para o mundo de IoT, há motivação mais do que suficiente para se realizar estudos futuros neste segmento, e possibilitar a solidificação do IoT na sociedade.

Referências

Ashry, A., Sharaf, K., and Ibrahim, M. (2009). A compact low-power UHF RFID tag. *Microelectronics Journal*, 40(11):1504–1513.

Borghoff, J. (2010). *Cryptanalysis of Lightweight Ciphers*. PhD thesis, Technical University of Denmark.

Burr, W. E. (2003). Selecting the Advanced Encryption Standard. *Security & Privacy, IEEE*, 1(2):43–52.

Daemen, J. and Rijmen, V. (1998). Aes proposal: Rijndael.

Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., Park, J., and Sheffield, D. (2008). Efficient Embedded Computing. *Computer*, 41(7):27–32.

Gladman, B. (2001). A specification for rijndael, the aes algorithm. *at fp. gladman. plus. com/cryptography_technology/rijndael/aes. spec*, 311:18–19.

Gura, N., Patel, A., Wander, A., Eberle, H., and Shantz, S. C. (2004). Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer.

Henkel, J. and Parameswaran, S. (2007). *Designing embedded processors: a low power perspective*. Springer.

Hennessy, J. L. and Patterson, D. A. (2012). *Computer architecture: a quantitative approach*. Elsevier.

Li, Y. and Henkel, J. (1998). A framework for estimation and minimizing energy dissipation of embedded HW/SW systems. In *Proceedings of the 35th annual conference on Design automation conference - DAC '98*, pages 188–193, New York, New York, USA. ACM Press.

Marwedel, P. (2006). Embedded System Design.

Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (2010). *Handbook of applied cryptography*. CRC press.

Muchnick, S. S. (1997). *Advanced compiler design implementation*. Morgan Kaufmann.

Pillai, V., Heinrich, H., Dieska, D., Nikitin, P. V., Martinez, R., and Rao, K. V. S. (2007). An Ultra-Low-Power Long Range Battery/Passive RFID Tag for UHF

and Microwave Bands With a Current Consumption of 700 nA at 1.5 V. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(7):1500–1512.

Potlapally, N. R., Member, S., Ravi, S., Raghunathan, A., Member, S., and Jha, N. K. (2006). A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols. *Mobile Computing, IEEE Transactions on*, 5(2):128–143.

Standard, R. C. (1998). Rsa public key cryptography standard 1 v. 2.0. *RSA Laboratories, Oct*, 5:1.

Steinke, S., Knauer, M., Wehmeyer, L., and Marwedel, P. (2001). An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc. of PATMOS*.

Tiwari, V., Malik, S., Wolfe, A., and Lee, M.-C. (1996). Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design*, pages 326–328. IEEE Comput. Soc. Press.

Yan, H. and Shi, Z. J. (2006). Studying software implementations of elliptic curve cryptography. In *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, pages 78–83. IEEE.

ANEXO B – Código fonte do assembly

```
#include "arm_arch.h"
.text
.code 32

.type AES_Te,%object
.align 5
AES_Te:
.word 0xc66363a5, 0xf87c7c84, 0xee777799, 0xf67b7b8d
.word 0xfff2f20d, 0xd66b6bbd, 0xde6f6fb1, 0x91c5c554
.word 0x60303050, 0x02010103, 0xce6767a9, 0x562b2b7d
.word 0xe7fefe19, 0xb5d7d762, 0x4dababe6, 0xec76769a
.word 0x8fcaca45, 0x1f82829d, 0x89c9c940, 0xfa7d7d87
.word 0xeffafa15, 0xb25959eb, 0x8e4747c9, 0xfbf0f00b
.word 0x41adadec, 0xb3d4d467, 0x5fa2a2fd, 0x45afafea
.word 0x239c9cbf, 0x53a4a4f7, 0xe4727296, 0x9bc0c05b
.word 0x75b7b7c2, 0xe1fdfd1c, 0x3d9393ae, 0x4c26266a
.word 0x6c36365a, 0x7e3f3f41, 0xf5f7f702, 0x83cccc4f
.word 0x6834345c, 0x51a5a5f4, 0xd1e5e534, 0xf9f1f108
.word 0xe2717193, 0xabd8d873, 0x62313153, 0x2a15153f
.word 0x0804040c, 0x95c7c752, 0x46232365, 0x9dc3c35e
.word 0x30181828, 0x379696a1, 0x0a05050f, 0x2f9a9ab5
.word 0x0e070709, 0x24121236, 0x1b80809b, 0xdfe2e23d
.word 0xcdebeb26, 0x4e272769, 0x7fb2b2cd, 0xea75759f
.word 0x1209091b, 0x1d83839e, 0x582c2c74, 0x341a1a2e
.word 0x361b1b2d, 0xdc6e6eb2, 0xb45a5aee, 0x5ba0a0fb
.word 0xa45252f6, 0x763b3b4d, 0xb7d6d661, 0x7db3b3ce
.word 0x5229297b, 0xdde3e33e, 0x5e2f2f71, 0x13848497
.word 0xa65353f5, 0xb9d1d168, 0x00000000, 0xc1eded2c
.word 0x40202060, 0xe3fcfc1f, 0x79b1b1c8, 0xb65b5bed
.word 0xd46a6abe, 0x8dcbcb46, 0x67bebed9, 0x7239394b
.word 0x944a4ade, 0x984c4cd4, 0xb05858e8, 0x85cfcf4a
.word 0xbbd0d06b, 0xc5efef2a, 0x4faaaae5, 0xedfbfb16
.word 0x864343c5, 0x9a4d4dd7, 0x66333355, 0x11858594
.word 0x8a4545cf, 0xe9f9f910, 0x04020206, 0xfe7f7f81
.word 0xa05050f0, 0x783c3c44, 0x259f9fba, 0x4ba8a8e3
.word 0xa25151f3, 0x5da3a3fe, 0x804040c0, 0x058f8f8a
.word 0x3f9292ad, 0x219d9dbc, 0x70383848, 0xf1f5f504
.word 0x63bcbcdf, 0x77b6b6c1, 0xafdada75, 0x42212163
.word 0x20101030, 0xe5ffff1a, 0xfdf3f30e, 0xbfd2d26d
.word 0x81cdcd4c, 0x180c0c14, 0x26131335, 0xc3ecec2f
.word 0xbe5f5fe1, 0x359797a2, 0x884444cc, 0x2e171739
.word 0x93c4c457, 0x55a7a7f2, 0xfc7e7e82, 0x7a3d3d47
.word 0xc86464ac, 0xba5d5de7, 0x3219192b, 0xe6737395
.word 0xc06060a0, 0x19818198, 0x9e4f4fd1, 0xa3dcdc7f
.word 0x44222266, 0x542a2a7e, 0x3b9090ab, 0x0b888883
.word 0x8c4646ca, 0xc7eeee29, 0x6bb8b8d3, 0x2814143c
.word 0xa7dede79, 0xbc5e5ee2, 0x160b0b1d, 0xaddbdb76
.word 0xdbe0e03b, 0x64323256, 0x743a3a4e, 0x140a0a1e
.word 0x924949db, 0x0c06060a, 0x4824246c, 0xb85c5ce4
.word 0x9fc2c25d, 0xbdd3d36e, 0x43acacef, 0xc46262a6
.word 0x399191a8, 0x319595a4, 0xd3e4e437, 0xf279798b
```

```
52  .word 0xd5e7e732, 0x8bc8c843, 0x6e373759, 0xda6d6db7
53  .word 0x018d8d8c, 0xb1d5d564, 0x9c4e4ed2, 0x49a9a9e0
54  .word 0xd86c6cb4, 0xac5656fa, 0xf3f4f407, 0xcfeaea25
55  .word 0xca6565af, 0xf47a7a8e, 0x47aeaee9, 0x10080818
56  .word 0x6fbabad5, 0xf0787888, 0x4a25256f, 0x5c2e2e72
57  .word 0x381c1c24, 0x57a6a6f1, 0x73b4b4c7, 0x97c6c651
58  .word 0xcbe8e823, 0xa1dddd7c, 0xe874749c, 0x3e1f1f21
59  .word 0x964b4bdd, 0x61bdbddc, 0x0d8b8b86, 0x0f8a8a85
60  .word 0xe0707090, 0x7c3e3e42, 0x71b5b5c4, 0xcc6666aa
61  .word 0x904848d8, 0x06030305, 0xf7f6f601, 0x1c0e0e12
62  .word 0xc26161a3, 0x6a35355f, 0xae5757f9, 0x69b9b9d0
63  .word 0x17868691, 0x99c1c158, 0x3a1d1d27, 0x279e9eb9
64  .word 0xd9e1e138, 0xebf8f813, 0x2b9898b3, 0x22111133
65  .word 0xd26969bb, 0xa9d9d970, 0x078e8e89, 0x339494a7
66  .word 0x2d9b9bb6, 0x3c1e1e22, 0x15878792, 0xc9e9e920
67  .word 0x87cece49, 0xaa5555ff, 0x50282878, 0xa5dfdf7a
68  .word 0x038c8c8f, 0x59a1a1f8, 0x09898980, 0x1a0d0d17
69  .word 0x65bfbfda, 0xd7e6e631, 0x844242c6, 0xd06868b8
70  .word 0x824141c3, 0x299999b0, 0x5a2d2d77, 0x1e0f0f11
71  .word 0x7bb0b0cb, 0xa85454fc, 0x6dbbbbd6, 0x2c16163a
72  @ Te4[256]
73  .byte 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5
74  .byte 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76
75  .byte 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0
76  .byte 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0
77  .byte 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc
78  .byte 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15
79  .byte 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a
80  .byte 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75
81  .byte 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0
82  .byte 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84
83  .byte 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b
84  .byte 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf
85  .byte 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85
86  .byte 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8
87  .byte 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5
88  .byte 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2
89  .byte 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17
90  .byte 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73
91  .byte 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88
92  .byte 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb
93  .byte 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c
94  .byte 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79
95  .byte 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9
96  .byte 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08
97  .byte 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6
98  .byte 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a
99  .byte 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e
100 .byte 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e
101 .byte 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94
102 .byte 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf
103 .byte 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68
```

```
104 .byte 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
105 @ rcon[]
106 .word 0x01000000, 0x02000000, 0x04000000, 0x08000000
107 .word 0x10000000, 0x20000000, 0x40000000, 0x80000000
108 .word 0x1B000000, 0x36000000, 0, 0, 0, 0, 0, 0
109 .size AES_Te,.-AES_Te
110
111 @ void AES_encrypt(const unsigned char *in, unsigned char *out,
112 @       const AES_KEY *key) {
113 .global AES_encrypt
114 .type   AES_encrypt,%function
115 .align  5
116 AES_encrypt:
117   sub r3,pc,#8     @ AES_encrypt
118   stmdb   sp!,{r1,r4-r12,lr}
119   mov r12,r0      @ inp
120   mov r11,r2
121   sub r10,r3,#AES_encrypt-AES_Te  @ Te
122 #if __ARM_ARCH__<7
123   ldrb  r0,[r12,#3] @ load input data in endian-neutral
124   ldrb  r4,[r12,#2] @ manner...
125   ldrb  r5,[r12,#1]
126   ldrb  r6,[r12,#0]
127   orr r0,r0,r4,lsl#8
128   ldrb  r1,[r12,#7]
129   orr r0,r0,r5,lsl#16
130   ldrb  r4,[r12,#6]
131   orr r0,r0,r6,lsl#24
132   ldrb  r5,[r12,#5]
133   ldrb  r6,[r12,#4]
134   orr r1,r1,r4,lsl#8
135   ldrb  r2,[r12,#11]
136   orr r1,r1,r5,lsl#16
137   ldrb  r4,[r12,#10]
138   orr r1,r1,r6,lsl#24
139   ldrb  r5,[r12,#9]
140   ldrb  r6,[r12,#8]
141   orr r2,r2,r4,lsl#8
142   ldrb  r3,[r12,#15]
143   orr r2,r2,r5,lsl#16
144   ldrb  r4,[r12,#14]
145   orr r2,r2,r6,lsl#24
146   ldrb  r5,[r12,#13]
147   ldrb  r6,[r12,#12]
148   orr r3,r3,r4,lsl#8
149   orr r3,r3,r5,lsl#16
150   orr r3,r3,r6,lsl#24
151 #else
152   ldr r0,[r12,#0]
153   ldr r1,[r12,#4]
154   ldr r2,[r12,#8]
155   ldr r3,[r12,#12]
```

```
156 #ifdef __ARMEL__
157   rev r0,r0
158   rev r1,r1
159   rev r2,r2
160   rev r3,r3
161 #endif
162 #endif
163   bl  _armv4_AES_encrypt
164
165   ldr r12,[sp],#4   @ pop out
166 #if __ARM_ARCH__>=7
167 #ifdef __ARMEL__
168   rev r0,r0
169   rev r1,r1
170   rev r2,r2
171   rev r3,r3
172 #endif
173   str r0,[r12,#0]
174   str r1,[r12,#4]
175   str r2,[r12,#8]
176   str r3,[r12,#12]
177 #else
178   mov r4,r0,lsr#24    @ write output in endian-neutral
179   mov r5,r0,lsr#16    @ manner...
180   mov r6,r0,lsr#8
181   strb  r4,[r12,#0]
182   strb  r5,[r12,#1]
183   mov r4,r1,lsr#24
184   strb  r6,[r12,#2]
185   mov r5,r1,lsr#16
186   strb  r0,[r12,#3]
187   mov r6,r1,lsr#8
188   strb  r4,[r12,#4]
189   strb  r5,[r12,#5]
190   mov r4,r2,lsr#24
191   strb  r6,[r12,#6]
192   mov r5,r2,lsr#16
193   strb  r1,[r12,#7]
194   mov r6,r2,lsr#8
195   strb  r4,[r12,#8]
196   strb  r5,[r12,#9]
197   mov r4,r3,lsr#24
198   strb  r6,[r12,#10]
199   mov r5,r3,lsr#16
200   strb  r2,[r12,#11]
201   mov r6,r3,lsr#8
202   strb  r4,[r12,#12]
203   strb  r5,[r12,#13]
204   strb  r6,[r12,#14]
205   strb  r3,[r12,#15]
206 #endif
207 #if __ARM_ARCH__>=5
```

```
208     ldmia sp!,{r4-r12,pc}
209 #else
210     ldmia   sp!,{r4-r12,lr}
211     tst lr,#1
212     moveq pc,lr      @ be binary compatible with V4, yet
213     .word 0xe12fff1e       @ interoperable with Thumb ISA:-)
214 #endif
215 .size AES_encrypt,.-AES_encrypt
216
217 .type   _armv4_AES_encrypt,%function
218 .align  2
219 _armv4_AES_encrypt:
220     str lr,[sp,#-4]!     @ push lr
221     ldmia r11!,{r4-r7}
222
223     eor r0,r0,r4
224     ldr r12,[r11,#240-16]
225     eor r1,r1,r5
226     eor r2,r2,r6
227     eor r3,r3,r7
228     sub r12,r12,#1
229     mov r12,r12,lsr#1 @ dividir por dois pois vamos fazer metade
            das iteracoes
230     mov lr,#255
231
232
233         @ ret 0  usa so r4 e r5,  salva em r6
234     mov r4,r0,lsr#24
235     ldr r4,[r10,r4,lsl#2] @ r4= X00
236     and r5,lr,r1,lsr#16
237     ldr r5,[r10,r5,lsl#2]
238     eor r4,r4,r5,ror#8  @ r4= XOO^X11
239     and r5,lr,r2,lsr#8
240     ldr r5,[r10,r5,lsl#2]
241     eor r4,r4,r5,ror#16 @ r4= XOO^X11^X22
242     and r5,lr,r3
243     ldr r5,[r10,r5,lsl#2]
244     eor r4,r4,r5,ror#24 @ r4= XOO^X11^X22^X33
245     ldr r5,[r11],#32     @ pegando chave[0] e incrementando em 8
            posicoes...
246     eor r6,r4,r5     @ t3 <- y
247
248 .Lenc_loop:
249         @ circ 0  r7, r8, r9, r6 estarao ocupados a partir de
                agora
250         @ ret 1  usa so r4 e r5, salva em r5 (r6 ja esta ocupado)
251     mov r4,r1,lsr#24
252     and r5,lr,r2,lsr#16
253     ldr r4,[r10,r4,lsl#2] @ r4= X01
254     ldr r5,[r10,r5,lsl#2]
255     and r7,lr,r6
256     eor r4,r4,r5,ror#8  @ r4= X01^X12
```

```
257     ldr r7,[r10,r7,lsl#2] @ i1 tem X30
258     and r5,lr,r3,lsr#8
259     and r8,lr,r6,lsr#8
260     ldr r5,[r10,r5,lsl#2]
261     ldr r8,[r10,r8,lsl#2] @ i2 tem X20
262     eor r4,r4,r5,ror#16 @ r4= X01^X12^X23
263     and r9,lr,r6,lsr#16
264     and r5,lr,r0
265     ldr r9,[r10,r9,lsl#2] @ i3 tem X10
266     ldr r5,[r10,r5,lsl#2]
267     mov r6,r6,lsr#24
268     eor r4,r4,r5,ror#24 @ r4= X01^X12^X23^X30
269     ldr r6,[r10,r6,lsl#2] @ t3 tem X00
270     ldr r5,[r11,#-16]    @ pegando chave[4]
271     mov r1,r1,lsl#16
272     eor r6,r6,r5
273     orr r1,r1,r3,lsr#16 @ agora r3 esta livre pra ser usado. r1
            contem [X21, X31, X03, X13]
274     ldr r5,[r11,#-28]    @ pegando chave[1]
275     ror r0,r0,#8
276     ror r2,r2,#8     @ liberando r2 tambÃ©m.
277     eor r5,r4,r5     @ t2 <- y
278     lsl r0,r0,#16
279
280         @ circ 1 @ r4, r2 e r3 estao livres, usar r4 como temp no
                circ
281         @ ret 2  usar r3/r2 como temps.
282     and r4,lr,r5,lsr#16
283     orr r0,r0,r2,lsr#16 @ r0 contem [X10, X20, X32, X02]
284     ldr r4,[r10,r4,lsl#2]
285     and r3,lr,r0
286     eor r6,r6,r4,ror#8  @ t3 tem X00 ^(X11 ror8)
287     ldr r3,[r10,r3,lsl#2] @ r3= X02
288     mov r4,r5,lsr#24
289     and r2,lr,r1
290     ldr r4,[r10,r4,lsl#2]
291     ldr r2,[r10,r2,lsl#2]
292     eor r7,r4,r7,ror#24 @ i1 tem (ror24 X30) ^ X01
293     eor r3,r3,r2,ror#8  @ r3= X02^X13
294     and r4,lr,r5,lsr#8
295     and r2,lr,r0,lsr#16
296     ldr r4,[r10,r4,lsl#2]
297     ldr r2,[r10,r2,lsl#2]
298     eor r9,r9,r4,ror#8  @ i3 tem X10 ^(X21 ror8)
299     eor r3,r3,r2,ror#16 @ r3= X02^X13^X20
300     and r4,lr,r5
301     and r2,lr,r1,lsr#16
302     ldr r4,[r10,r4,lsl#2]
303     ldr r2,[r10,r2,lsl#2]
304     eor r8,r8,r4,ror#8  @ i2 tem X20 ^(X31 ror8)
305     eor r3,r3,r2,ror#24 @ r3= X02^X13^X20^X31
306     ldr r4,[r11,#-12]    @ pegando chave[5]
```

```
307     ldr r2,[r11,#-24]   @ pegando chave[2]
308     eor r7,r4,r7
309     eor r3,r3,r2     @ s3 <- y
310
311
312         @ circ 2 r4, r2 e r5 estao livres, usar r4 como temp no
                circ
313         @ ret 3  usar r2/r5 como temps.
314     and r4,lr,r3,lsr#8
315     and r5,lr,r1,lsr#8
316     ldr r4,[r10,r4,lsl#2]
317     ldr r5,[r10,r5,lsl#2] @ r5= X03
318     eor r6,r6,r4,ror#16 @ t3 tem X00 ^(X11 ror8) ^ (X22 ror16)
319     mov r2,r0,lsr#24
320     and r4,lr,r3,lsr#16
321     ldr r2,[r10,r2,lsl#2]
322     ldr r4,[r10,r4,lsl#2]
323     eor r5,r5,r2,ror#8  @ r5= X03^X10
324     eor r7,r7,r4,ror#8  @ i1 tem (X30 ror24) ^ X01 ^ (X12 ror 8)
325     mov r2,r1,lsr#24
326     and r4,lr,r3
327     ldr r2,[r10,r2,lsl#2]
328     ldr r4,[r10,r4,lsl#2]
329     eor r5,r5,r2,ror#16 @ r5= X03^X10^X21
330     eor r9,r9,r4,ror#16 @ i3 tem X10 ^(X21 ror8)^(X32 ror16)
331     and r2,lr,r0,lsr#8
332     mov r4,r3,lsr#24
333     ldr r2,[r10,r2,lsl#2]
334     ldr r4,[r10,r4,lsl#2]
335     eor r5,r5,r2,ror#24 @ r5= X03^X10^X21^X32
336     eor r8,r4,r8,ror#16 @ i2 tem ((X20 ^(X31 ror8))ror16) ^ X02
337     ldr r2,[r11,#-20]   @ pegando chave[3]
338     ldr r4,[r11,#-8]    @ pegando chave[6]
339     eor r5,r5,r2     @ t2 <- y
340     eor r8,r4,r8
341
342
343         @ circ 2 r4, e r3 estao livres, usar r3 como temp no circ
344         @ fixed agora. mas a alocaÃ§Ã£o dos registradores aqui
                pode ser melhorada creio eu.
345         @ ret 0  usa soh r4 e r2,  salva em r6 la no final soh //
                nao deve ter problema usar t3,
346         @ pq ele vai ser substituido logo de cara no comeco...
347     and r0,lr,r5
348     ldr r3,[r11,#-4]    @ pegando chave[7]
349     ldr r0,[r10,r0,lsl#2]
350     and r1,lr,r5,lsr#8
351     eor r0,r6,r0,ror#24 @ t3 tem X00 ^(X11 ror8) ^ (X22 ror8) ^ (
            X33 ror24)
352     mov r4,r0,lsr#24
353     ldr r1,[r10,r1,lsl#2]
354     ldr r4,[r10,r4,lsl#2] @ r4= X00
```

```
355     eor r1,r7,r1,ror#16 @ i1 tem (X30 ror24) ^ X01 ^ (X12 ror8) ^
            (X23 ^ ror16)
356     ldr r6,[r11],#32    @ pegando chave[0] e incrementando em 16
            posicoes...
357     and r2,lr,r5,lsr#16
358     and r7,lr,r1,lsr#16
359     ldr r2,[r10,r2,lsl#2]
360     ldr r7,[r10,r7,lsl#2]
361     eor r2,r8,r2,ror#8  @ i2 tem ((X20 ^(X31 ror8))ror16) ^ X02 ^
            (X13 ror8)
362     eor r4,r4,r7,ror#8  @ r4= XOO^X11
363     and r8,lr,r2,lsr#8
364     mov r5,r5,lsr#24
365     ldr r8,[r10,r8,lsl#2]
366     ldr r5,[r10,r5,lsl#2]
367     eor r4,r4,r8,ror#16 @ r4= XOO^X11^X22
368
369     eor r9,r5,r9,ror#8  @ i3 tem (X10 ^(X21 ror8)^(X32 ror16))ror8
            ^ X03
370     eor r3,r3,r9
371     and r8,lr,r3
372     ldr r8,[r10,r8,lsl#2]
373     eor r4,r4,r8,ror#24 @ r4= XOO^X11^X22^X33
374     eor r6,r4,r6    @ t3 <- y
375
376     subs  r12,r12,#1
377     bne .Lenc_loop
378
379
380        @ ret 1 final:  usa soh r4 e r5, salva em r5
381
382     mov r4,r1,lsr#24
383     mov r7,r2,lsr#24
384     mov r12,r3,lsr#24
385     ldr r4,[r10,r4,lsl#2] @ r4= X01
386     ldr r7,[r10,r7,lsl#2] @ r7= X02
387     ldr r12,[r10,r12,lsl#2] @ r12= X03
388     and r5,lr,r2,lsr#16
389     and r9,lr,r3,lsr#16
390     and r8,lr,r0,lsr#16
391     ldr r5,[r10,r5,lsl#2]
392     ldr r9,[r10,r9,lsl#2]
393     ldr r8,[r10,r8,lsl#2]
394     eor r4,r4,r5,ror#8  @ r4= X01^X12
395     eor r7,r7,r9,ror#8  @ r7= X02^X13
396     eor r12,r12,r8,ror#8  @ r5= X03^X10
397     and r5,lr,r3,lsr#8
398     and r9,lr,r0,lsr#8
399     and r8,lr,r1,lsr#8
400     ldr r5,[r10,r5,lsl#2]
401     ldr r9,[r10,r9,lsl#2]
402     ldr r8,[r10,r8,lsl#2]
```

```
403     eor r4,r4,r5,ror#16 @ r4= X01^X12^X23
404     eor r7,r7,r9,ror#16 @ r7= X02^X13^X20
405     eor r12,r12,r8,ror#16 @ r5= X03^X10^X21
406     and r5,lr,r0
407     and r9,lr,r1
408     and r8,lr,r2
409     ldr r5,[r10,r5,lsl#2]
410     ldr r9,[r10,r9,lsl#2]
411     ldr r8,[r10,r8,lsl#2]
412     eor r4,r4,r5,ror#24 @ r4= X01^X12^X23^X30
413     eor r7,r7,r9,ror#24 @ r7= X02^X13^X20^X31
414     mov r0,r6
415     eor r12,r12,r8,ror#24 @ r5= X03^X10^X21^X32
416     ldr r5,[r11,#-28]   @ pegando chave[1]
417     ldr r9,[r11,#-24]   @ pegando chave[2]
418     ldr r8,[r11,#-20]   @ pegando chave[3]
419     eor r1,r4,r5    @ t1 <- y0 //primeira coluna
420     eor r2,r9,r7    @ i1 <- y
421     eor r3,r12,r8   @ rounds <- y
422
423     add r12,r10,#2
424     and r7,lr,r0
425     and r8,lr,r0,lsr#8
426     and r9,lr,r0,lsr#16
427     mov r0,r0,lsr#24
428
429     ldrb  r4,[r12,r7,lsl#2] @ Te4[s0>>0]
430     and r7,lr,r1,lsr#16 @ i0
431     ldrb  r5,[r12,r8,lsl#2] @ Te4[s0>>8]
432     and r8,lr,r1
433     ldrb  r6,[r12,r9,lsl#2] @ Te4[s0>>16]
434     and r9,lr,r1,lsr#8
435     ldrb  r0,[r12,r0,lsl#2] @ Te4[s0>>24]
436     mov r1,r1,lsr#24
437
438     ldrb  r7,[r12,r7,lsl#2] @ Te4[s1>>16]
439     ldrb  r8,[r12,r8,lsl#2] @ Te4[s1>>0]
440     ldrb  r9,[r12,r9,lsl#2] @ Te4[s1>>8]
441     eor r0,r7,r0,lsl#8
442     ldrb  r1,[r12,r1,lsl#2] @ Te4[s1>>24]
443     and r7,lr,r2,lsr#8  @ i0
444     eor r5,r8,r5,lsl#8
445     and r8,lr,r2,lsr#16 @ i1
446     eor r6,r9,r6,lsl#8
447     and r9,lr,r2
448     ldrb  r7,[r12,r7,lsl#2] @ Te4[s2>>8]
449     eor r1,r4,r1,lsl#24
450     ldrb  r8,[r12,r8,lsl#2] @ Te4[s2>>16]
451     mov r2,r2,lsr#24
452
453     ldrb  r9,[r12,r9,lsl#2] @ Te4[s2>>0]
454     eor r0,r7,r0,lsl#8
```

```
455     ldrb    r2,[r12,r2,lsl#2] @ Te4[s2>>24]
456     and     r7,lr,r3        @ i0
457     eor     r1,r1,r8,lsl#16
458     and     r8,lr,r3,lsr#8  @ i1
459     eor     r6,r9,r6,lsl#8
460     and     r9,lr,r3,lsr#16 @ i2
461     ldrb    r7,[r12,r7,lsl#2] @ Te4[s3>>0]
462     eor     r2,r5,r2,lsl#24
463     ldrb    r8,[r12,r8,lsl#2] @ Te4[s3>>8]
464     mov     r3,r3,lsr#24
465
466     ldrb    r9,[r12,r9,lsl#2] @ Te4[s3>>16]
467     eor     r0,r7,r0,lsl#8
468     ldr     r7,[r11,#-16]
469     ldrb    r3,[r12,r3,lsl#2] @ Te4[s3>>24]
470     eor     r1,r1,r8,lsl#8
471     ldr     r4,[r11,#-12]
472     eor     r2,r2,r9,lsl#16
473     ldr     r5,[r11,#-8]
474     eor     r3,r6,r3,lsl#24
475     ldr     r6,[r11,#-4]
476
477     eor     r0,r0,r7
478     eor     r1,r1,r4
479     eor     r2,r2,r5
480     eor     r3,r3,r6
481     ldr     pc,[sp],#4      @ pop and return
482 .size   _armv4_AES_encrypt,.-_armv4_AES_encrypt
483
484 .global private_AES_set_encrypt_key
485 .type   private_AES_set_encrypt_key,%function
486 .align  5
487 private_AES_set_encrypt_key:
488 _armv4_AES_set_encrypt_key:
489     sub     r3,pc,#8        @ AES_set_encrypt_key
490     teq     r0,#0
491     moveq   r0,#-1
492     beq     .Labrt
493     teq     r2,#0
494     moveq   r0,#-1
495     beq     .Labrt
496
497     teq     r1,#128
498     beq     .Lok
499     teq     r1,#192
500     beq     .Lok
501     teq     r1,#256
502     movne   r0,#-1
503     bne     .Labrt
504
505 .Lok:   stmdb   sp!,{r4-r12,lr}
506     sub     r10,r3,#_armv4_AES_set_encrypt_key-AES_Te-1024  @ Te4
```

```
507
508     mov r12,r0      @ inp
509     mov lr,r1       @ bits
510     mov r11,r2        @ key
511
512 #if __ARM_ARCH__<7
513     ldrb    r0,[r12,#3] @ load input data in endian-neutral
514     ldrb    r4,[r12,#2] @ manner...
515     ldrb    r5,[r12,#1]
516     ldrb    r6,[r12,#0]
517     orr r0,r0,r4,lsl#8
518     ldrb    r1,[r12,#7]
519     orr r0,r0,r5,lsl#16
520     ldrb    r4,[r12,#6]
521     orr r0,r0,r6,lsl#24
522     ldrb    r5,[r12,#5]
523     ldrb    r6,[r12,#4]
524     orr r1,r1,r4,lsl#8
525     ldrb    r2,[r12,#11]
526     orr r1,r1,r5,lsl#16
527     ldrb    r4,[r12,#10]
528     orr r1,r1,r6,lsl#24
529     ldrb    r5,[r12,#9]
530     ldrb    r6,[r12,#8]
531     orr r2,r2,r4,lsl#8
532     ldrb    r3,[r12,#15]
533     orr r2,r2,r5,lsl#16
534     ldrb    r4,[r12,#14]
535     orr r2,r2,r6,lsl#24
536     ldrb    r5,[r12,#13]
537     ldrb    r6,[r12,#12]
538     orr r3,r3,r4,lsl#8
539     str r0,[r11],#16
540     orr r3,r3,r5,lsl#16
541     str r1,[r11,#-12]
542     orr r3,r3,r6,lsl#24
543     str r2,[r11,#-8]
544     str r3,[r11,#-4]
545 #else
546     ldr r0,[r12,#0]
547     ldr r1,[r12,#4]
548     ldr r2,[r12,#8]
549     ldr r3,[r12,#12]
550 #ifdef __ARMEL__
551     rev r0,r0
552     rev r1,r1
553     rev r2,r2
554     rev r3,r3
555 #endif
556     str r0,[r11],#16
557     str r1,[r11,#-12]
558     str r2,[r11,#-8]
```

```
559     str r3,[r11,#-4]
560 #endif
561
562     teq lr,#128
563     bne .Lnot128
564     mov r12,#10
565     str r12,[r11,#240-16]
566     add r6,r10,#256     @ rcon
567     mov lr,#255
568
569 .L128_loop:
570     and r5,lr,r3,lsr#24
571     and r7,lr,r3,lsr#16
572     ldrb    r5,[r10,r5]
573     and r8,lr,r3,lsr#8
574     ldrb    r7,[r10,r7]
575     and r9,lr,r3
576     ldrb    r8,[r10,r8]
577     orr r5,r5,r7,lsl#24
578     ldrb    r9,[r10,r9]
579     orr r5,r5,r8,lsl#16
580     ldr r4,[r6],#4      @ rcon[i++]
581     orr r5,r5,r9,lsl#8
582     eor r5,r5,r4
583     eor r0,r0,r5        @ rk[4]=rk[0]^...
584     eor r1,r1,r0        @ rk[5]=rk[1]^rk[4]
585     str r0,[r11],#16
586     eor r2,r2,r1        @ rk[6]=rk[2]^rk[5]
587     str r1,[r11,#-12]
588     eor r3,r3,r2        @ rk[7]=rk[3]^rk[6]
589     str r2,[r11,#-8]
590     subs    r12,r12,#1
591     str r3,[r11,#-4]
592     bne .L128_loop
593     sub r2,r11,#176
594     b .Ldone
595
596 .Lnot128:
597 #if __ARM_ARCH__<7
598     ldrb    r8,[r12,#19]
599     ldrb    r4,[r12,#18]
600     ldrb    r5,[r12,#17]
601     ldrb    r6,[r12,#16]
602     orr r8,r8,r4,lsl#8
603     ldrb    r9,[r12,#23]
604     orr r8,r8,r5,lsl#16
605     ldrb    r4,[r12,#22]
606     orr r8,r8,r6,lsl#24
607     ldrb    r5,[r12,#21]
608     ldrb    r6,[r12,#20]
609     orr r9,r9,r4,lsl#8
610     orr r9,r9,r5,lsl#16
```

```
611     str r8,[r11],#8
612     orr r9,r9,r6,lsl#24
613     str r9,[r11,#-4]
614 #else
615     ldr r8,[r12,#16]
616     ldr r9,[r12,#20]
617 #ifdef __ARMEL__
618     rev r8,r8
619     rev r9,r9
620 #endif
621     str r8,[r11],#8
622     str r9,[r11,#-4]
623 #endif
624
625     teq lr,#192
626     bne .Lnot192
627     mov r12,#12
628     str r12,[r11,#240-24]
629     add r6,r10,#256     @ rcon
630     mov lr,#255
631     mov r12,#8
632
633 .L192_loop:
634     and r5,lr,r9,lsr#24
635     and r7,lr,r9,lsr#16
636     ldrb  r5,[r10,r5]
637     and r8,lr,r9,lsr#8
638     ldrb  r7,[r10,r7]
639     and r9,lr,r9
640     ldrb  r8,[r10,r8]
641     orr r5,r5,r7,lsl#24
642     ldrb  r9,[r10,r9]
643     orr r5,r5,r8,lsl#16
644     ldr r4,[r6],#4      @ rcon[i++]
645     orr r5,r5,r9,lsl#8
646     eor r9,r5,r4
647     eor r0,r0,r9        @ rk[6]=rk[0]^...
648     eor r1,r1,r0        @ rk[7]=rk[1]^rk[6]
649     str r0,[r11],#24
650     eor r2,r2,r1        @ rk[8]=rk[2]^rk[7]
651     str r1,[r11,#-20]
652     eor r3,r3,r2        @ rk[9]=rk[3]^rk[8]
653     str r2,[r11,#-16]
654     subs  r12,r12,#1
655     str r3,[r11,#-12]
656     subeq r2,r11,#216
657     beq .Ldone
658
659     ldr r7,[r11,#-32]
660     ldr r8,[r11,#-28]
661     eor r7,r7,r3        @ rk[10]=rk[4]^rk[9]
662     eor r9,r8,r7        @ rk[11]=rk[5]^rk[10]
```

```
663     str r7,[r11,#-8]
664     str r9,[r11,#-4]
665     b .L192_loop
666
667 .Lnot192:
668 #if __ARM_ARCH__<7
669     ldrb    r8,[r12,#27]
670     ldrb    r4,[r12,#26]
671     ldrb    r5,[r12,#25]
672     ldrb    r6,[r12,#24]
673     orr r8,r8,r4,lsl#8
674     ldrb    r9,[r12,#31]
675     orr r8,r8,r5,lsl#16
676     ldrb    r4,[r12,#30]
677     orr r8,r8,r6,lsl#24
678     ldrb    r5,[r12,#29]
679     ldrb    r6,[r12,#28]
680     orr r9,r9,r4,lsl#8
681     orr r9,r9,r5,lsl#16
682     str r8,[r11],#8
683     orr r9,r9,r6,lsl#24
684     str r9,[r11,#-4]
685 #else
686     ldr r8,[r12,#24]
687     ldr r9,[r12,#28]
688 #ifdef __ARMEL__
689     rev r8,r8
690     rev r9,r9
691 #endif
692     str r8,[r11],#8
693     str r9,[r11,#-4]
694 #endif
695
696     mov r12,#14
697     str r12,[r11,#240-32]
698     add r6,r10,#256     @ rcon
699     mov lr,#255
700     mov r12,#7
701
702 .L256_loop:
703     and r5,lr,r9,lsr#24
704     and r7,lr,r9,lsr#16
705     ldrb    r5,[r10,r5]
706     and r8,lr,r9,lsr#8
707     ldrb    r7,[r10,r7]
708     and r9,lr,r9
709     ldrb    r8,[r10,r8]
710     orr r5,r5,r7,lsl#24
711     ldrb    r9,[r10,r9]
712     orr r5,r5,r8,lsl#16
713     ldr r4,[r6],#4      @ rcon[i++]
714     orr r5,r5,r9,lsl#8
```

```
715     eor r9,r5,r4
716     eor r0,r0,r9        @ rk[8]=rk[0]^...
717     eor r1,r1,r0        @ rk[9]=rk[1]^rk[8]
718     str r0,[r11],#32
719     eor r2,r2,r1        @ rk[10]=rk[2]^rk[9]
720     str r1,[r11,#-28]
721     eor r3,r3,r2        @ rk[11]=rk[3]^rk[10]
722     str r2,[r11,#-24]
723     subs  r12,r12,#1
724     str r3,[r11,#-20]
725     subeq r2,r11,#256
726     beq .Ldone
727
728     and r5,lr,r3
729     and r7,lr,r3,lsr#8
730     ldrb  r5,[r10,r5]
731     and r8,lr,r3,lsr#16
732     ldrb  r7,[r10,r7]
733     and r9,lr,r3,lsr#24
734     ldrb  r8,[r10,r8]
735     orr r5,r5,r7,lsl#8
736     ldrb  r9,[r10,r9]
737     orr r5,r5,r8,lsl#16
738     ldr r4,[r11,#-48]
739     orr r5,r5,r9,lsl#24
740
741     ldr r7,[r11,#-44]
742     ldr r8,[r11,#-40]
743     eor r4,r4,r5        @ rk[12]=rk[4]^...
744     ldr r9,[r11,#-36]
745     eor r7,r7,r4        @ rk[13]=rk[5]^rk[12]
746     str r4,[r11,#-16]
747     eor r8,r8,r7        @ rk[14]=rk[6]^rk[13]
748     str r7,[r11,#-12]
749     eor r9,r9,r8        @ rk[15]=rk[7]^rk[14]
750     str r8,[r11,#-8]
751     str r9,[r11,#-4]
752     b .L256_loop
753
754 .Ldone: mov r0,#0
755     ldmia   sp!,{r4-r12,lr}
756 .Labrt: tst lr,#1
757     moveq pc,lr       @ be binary compatible with V4, yet
758     .word 0xe12fff1e      @ interoperable with Thumb ISA:-)
759 .size private_AES_set_encrypt_key,.-private_AES_set_encrypt_key
760
761 .global private_AES_set_decrypt_key
762 .type   private_AES_set_decrypt_key,%function
763 .align  5
764 private_AES_set_decrypt_key:
765     str lr,[sp,#-4]!            @ push lr
766     bl  _armv4_AES_set_encrypt_key
```

```
767     teq r0,#0
768     ldrne lr,[sp],#4                @ pop lr
769     bne .Labrt
770
771     stmdb   sp!,{r4-r12}
772
773     ldr r12,[r2,#240] @ AES_set_encrypt_key preserves r2,
774     mov r11,r2      @ which is AES_KEY *key
775     mov r7,r2
776     add r8,r2,r12,lsl#4
777
778 .Linv:  ldr r0,[r7]
779     ldr r1,[r7,#4]
780     ldr r2,[r7,#8]
781     ldr r3,[r7,#12]
782     ldr r4,[r8]
783     ldr r5,[r8,#4]
784     ldr r6,[r8,#8]
785     ldr r9,[r8,#12]
786     str r0,[r8],#-16
787     str r1,[r8,#16+4]
788     str r2,[r8,#16+8]
789     str r3,[r8,#16+12]
790     str r4,[r7],#16
791     str r5,[r7,#-12]
792     str r6,[r7,#-8]
793     str r9,[r7,#-4]
794     teq r7,r8
795     bne .Linv
796     ldr r0,[r11,#16]!   @ prefetch tp1
797     mov r7,#0x80
798     mov r8,#0x1b
799     orr r7,r7,#0x8000
800     orr r8,r8,#0x1b00
801     orr r7,r7,r7,lsl#16
802     orr r8,r8,r8,lsl#16
803     sub r12,r12,#1
804     mvn r9,r7
805     mov r12,r12,lsl#2 @ (rounds-1)*4
806
807 .Lmix:  and r4,r0,r7
808     and r1,r0,r9
809     sub r4,r4,r4,lsr#7
810     and r4,r4,r8
811     eor r1,r4,r1,lsl#1  @ tp2
812
813     and r4,r1,r7
814     and r2,r1,r9
815     sub r4,r4,r4,lsr#7
816     and r4,r4,r8
817     eor r2,r4,r2,lsl#1  @ tp4
818
```

```
819     and r4,r2,r7
820     and r3,r2,r9
821     sub r4,r4,r4,lsr#7
822     and r4,r4,r8
823     eor r3,r4,r3,lsl#1  @ tp8
824
825     eor r4,r1,r2
826     eor r5,r0,r3        @ tp9
827     eor r4,r4,r3        @ tpe
828     eor r4,r4,r1,ror#24
829     eor r4,r4,r5,ror#24 @ ^= ROTATE(tpb=tp9^tp2,8)
830     eor r4,r4,r2,ror#16
831     eor r4,r4,r5,ror#16 @ ^= ROTATE(tpd=tp9^tp4,16)
832     eor r4,r4,r5,ror#8  @ ^= ROTATE(tp9,24)
833
834     ldr r0,[r11,#4]     @ prefetch tp1
835     str r4,[r11],#4
836     subs    r12,r12,#1
837     bne .Lmix
838
839   mov r0,#0
840 #if __ARM_ARCH__>=5
841     ldmia sp!,{r4-r12,pc}
842 #else
843     ldmia   sp!,{r4-r12,lr}
844     tst lr,#1
845     moveq pc,lr         @ be binary compatible with V4, yet
846     .word 0xe12fff1e        @ interoperable with Thumb ISA:-)
847 #endif
848 .size private_AES_set_decrypt_key,.-private_AES_set_decrypt_key
849
850 .type AES_Td,%object
851 .align  5
852 AES_Td:
853 .word 0x51f4a750, 0x7e416553, 0x1a17a4c3, 0x3a275e96
854 .word 0x3bab6bcb, 0x1f9d45f1, 0xacfa58ab, 0x4be30393
855 .word 0x2030fa55, 0xad766df6, 0x88cc7691, 0xf5024c25
856 .word 0x4fe5d7fc, 0xc52acbd7, 0x26354480, 0xb562a38f
857 .word 0xdeb15a49, 0x25ba1b67, 0x45ea0e98, 0x5dfec0e1
858 .word 0xc32f7502, 0x814cf012, 0x8d4697a3, 0x6bd3f9c6
859 .word 0x038f5fe7, 0x15929c95, 0xbf6d7aeb, 0x955259da
860 .word 0xd4be832d, 0x587421d3, 0x49e06929, 0x8ec9c844
861 .word 0x75c2896a, 0xf48e7978, 0x99583e6b, 0x27b971dd
862 .word 0xbee14fb6, 0xf088ad17, 0xc920ac66, 0x7dce3ab4
863 .word 0x63df4a18, 0xe51a3182, 0x97513360, 0x62537f45
864 .word 0xb16477e0, 0xbb6bae84, 0xfe81a01c, 0xf9082b94
865 .word 0x70486858, 0x8f45fd19, 0x94de6c87, 0x527bf8b7
866 .word 0xab73d323, 0x724b02e2, 0xe31f8f57, 0x6655ab2a
867 .word 0xb2eb2807, 0x2fb5c203, 0x86c57b9a, 0xd33708a5
868 .word 0x302887f2, 0x23bfa5b2, 0x02036aba, 0xed16825c
869 .word 0x8acf1c2b, 0xa779b492, 0xf307f2f0, 0x4e69e2a1
870 .word 0x65daf4cd, 0x0605bed5, 0xd134621f, 0xc4a6fe8a
```

```
.word 0x342e539d, 0xa2f355a0, 0x058ae132, 0xa4f6eb75
.word 0x0b83ec39, 0x4060efaa, 0x5e719f06, 0xbd6e1051
.word 0x3e218af9, 0x96dd063d, 0xdd3e05ae, 0x4de6bd46
.word 0x91548db5, 0x71c45d05, 0x0406d46f, 0x605015ff
.word 0x1998fb24, 0xd6bde997, 0x894043cc, 0x67d99e77
.word 0xb0e842bd, 0x07898b88, 0xe7195b38, 0x79c8eedb
.word 0xa17c0a47, 0x7c420fe9, 0xf8841ec9, 0x00000000
.word 0x09808683, 0x322bed48, 0x1e1170ac, 0x6c5a724e
.word 0xfd0efffb, 0x0f853856, 0x3daed51e, 0x362d3927
.word 0x0a0fd964, 0x685ca621, 0x9b5b54d1, 0x24362e3a
.word 0x0c0a67b1, 0x9357e70f, 0xb4ee96d2, 0x1b9b919e
.word 0x80c0c54f, 0x61dc20a2, 0x5a774b69, 0x1c121a16
.word 0xe293ba0a, 0xc0a02ae5, 0x3c22e043, 0x121b171d
.word 0x0e090d0b, 0xf28bc7ad, 0x2db6a8b9, 0x141ea9c8
.word 0x57f11985, 0xaf75074c, 0xee99ddbb, 0xa37f60fd
.word 0xf701269f, 0x5c72f5bc, 0x44663bc5, 0x5bfb7e34
.word 0x8b432976, 0xcb23c6dc, 0xb6edfc68, 0xb8e4f163
.word 0xd731dcca, 0x42638510, 0x13972240, 0x84c61120
.word 0x854a247d, 0xd2bb3df8, 0xaef93211, 0xc729a16d
.word 0x1d9e2f4b, 0xdcb230f3, 0x0d8652ec, 0x77c1e3d0
.word 0x2bb3166c, 0xa970b999, 0x119448fa, 0x47e96422
.word 0xa8fc8cc4, 0xa0f03f1a, 0x567d2cd8, 0x223390ef
.word 0x87494ec7, 0xd938d1c1, 0x8ccaa2fe, 0x98d40b36
.word 0xa6f581cf, 0xa57ade28, 0xdab78e26, 0x3fadbfa4
.word 0x2c3a9de4, 0x5078920d, 0x6a5fcc9b, 0x547e4662
.word 0xf68d13c2, 0x90d8b8e8, 0x2e39f75e, 0x82c3aff5
.word 0x9f5d80be, 0x69d0937c, 0x6fd52da9, 0xcf2512b3
.word 0xc8ac993b, 0x10187da7, 0xe89c636e, 0xdb3bbb7b
.word 0xcd267809, 0x6e5918f4, 0xec9ab701, 0x834f9aa8
.word 0xe6956e65, 0xaaffe67e, 0x21bccf08, 0xef15e8e6
.word 0xbae79bd9, 0x4a6f36ce, 0xea9f09d4, 0x29b07cd6
.word 0x31a4b2af, 0x2a3f2331, 0xc6a59430, 0x35a266c0
.word 0x744ebc37, 0xfc82caa6, 0xe090d0b0, 0x33a7d815
.word 0xf104984a, 0x41ecdaf7, 0x7fcd500e, 0x1791f62f
.word 0x764dd68d, 0x43efb04d, 0xccaa4d54, 0xe49604df
.word 0x9ed1b5e3, 0x4c6a881b, 0xc12c1fb8, 0x4665517f
.word 0x9d5eea04, 0x018c355d, 0xfa877473, 0xfb0b412e
.word 0xb3671d5a, 0x92dbd252, 0xe9105633, 0x6dd64713
.word 0x9ad7618c, 0x37a10c7a, 0x59f8148e, 0xeb133c89
.word 0xcea927ee, 0xb761c935, 0xe11ce5ed, 0x7a47b13c
.word 0x9cd2df59, 0x55f2733f, 0x1814ce79, 0x73c737bf
.word 0x53f7cdea, 0x5ffdaa5b, 0xdf3d6f14, 0x7844db86
.word 0xcaaff381, 0xb968c43e, 0x3824342c, 0xc2a3405f
.word 0x161dc372, 0xbce2250c, 0x283c498b, 0xff0d9541
.word 0x39a80171, 0x080cb3de, 0xd8b4e49c, 0x6456c190
.word 0x7bcb8461, 0xd532b670, 0x486c5c74, 0xd0b85742
@ Td4[256]
.byte 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38
.byte 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
.byte 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87
.byte 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
.byte 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d
```

```
923 .byte 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
924 .byte 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2
925 .byte 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
926 .byte 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16
927 .byte 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
928 .byte 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda
929 .byte 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
930 .byte 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a
931 .byte 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
932 .byte 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02
933 .byte 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
934 .byte 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea
935 .byte 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
936 .byte 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85
937 .byte 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
938 .byte 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89
939 .byte 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
940 .byte 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20
941 .byte 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
942 .byte 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31
943 .byte 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
944 .byte 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d
945 .byte 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
946 .byte 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0
947 .byte 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
948 .byte 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26
949 .byte 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
950 .size AES_Td,.-AES_Td
951
952 @ void AES_decrypt(const unsigned char *in, unsigned char *out,
953 @       const AES_KEY *key) {
954 .global AES_decrypt
955 .type   AES_decrypt,%function
956 .align  5
957 AES_decrypt:
958   sub r3,pc,#8     @ AES_decrypt
959   stmdb   sp!,{r1,r4-r12,lr}
960   mov r12,r0     @ inp
961   mov r11,r2
962   sub r10,r3,#AES_decrypt-AES_Td     @ Td
963 #if __ARM_ARCH__<7
964   ldrb   r0,[r12,#3] @ load input data in endian-neutral
965   ldrb   r4,[r12,#2] @ manner...
966   ldrb   r5,[r12,#1]
967   ldrb   r6,[r12,#0]
968   orr r0,r0,r4,lsl#8
969   ldrb   r1,[r12,#7]
970   orr r0,r0,r5,lsl#16
971   ldrb   r4,[r12,#6]
972   orr r0,r0,r6,lsl#24
973   ldrb   r5,[r12,#5]
974   ldrb   r6,[r12,#4]
```

```
975     orr r1,r1,r4,lsl#8
976     ldrb    r2,[r12,#11]
977     orr r1,r1,r5,lsl#16
978     ldrb    r4,[r12,#10]
979     orr r1,r1,r6,lsl#24
980     ldrb    r5,[r12,#9]
981     ldrb    r6,[r12,#8]
982     orr r2,r2,r4,lsl#8
983     ldrb    r3,[r12,#15]
984     orr r2,r2,r5,lsl#16
985     ldrb    r4,[r12,#14]
986     orr r2,r2,r6,lsl#24
987     ldrb    r5,[r12,#13]
988     ldrb    r6,[r12,#12]
989     orr r3,r3,r4,lsl#8
990     orr r3,r3,r5,lsl#16
991     orr r3,r3,r6,lsl#24
992 #else
993     ldr r0,[r12,#0]
994     ldr r1,[r12,#4]
995     ldr r2,[r12,#8]
996     ldr r3,[r12,#12]
997 #ifdef __ARMEL__
998     rev r0,r0
999     rev r1,r1
1000    rev r2,r2
1001    rev r3,r3
1002 #endif
1003 #endif
1004    bl  _armv4_AES_decrypt
1005
1006    ldr r12,[sp],#4    @ pop out
1007 #if __ARM_ARCH__>=7
1008 #ifdef __ARMEL__
1009    rev r0,r0
1010    rev r1,r1
1011    rev r2,r2
1012    rev r3,r3
1013 #endif
1014    str r0,[r12,#0]
1015    str r1,[r12,#4]
1016    str r2,[r12,#8]
1017    str r3,[r12,#12]
1018 #else
1019    mov r4,r0,lsr#24     @ write output in endian-neutral
1020    mov r5,r0,lsr#16     @ manner...
1021    mov r6,r0,lsr#8
1022    strb    r4,[r12,#0]
1023    strb    r5,[r12,#1]
1024    mov r4,r1,lsr#24
1025    strb    r6,[r12,#2]
1026    mov r5,r1,lsr#16
```

```
1027    strb    r0,[r12,#3]
1028    mov r6,r1,lsr#8
1029    strb    r4,[r12,#4]
1030    strb    r5,[r12,#5]
1031    mov r4,r2,lsr#24
1032    strb    r6,[r12,#6]
1033    mov r5,r2,lsr#16
1034    strb    r1,[r12,#7]
1035    mov r6,r2,lsr#8
1036    strb    r4,[r12,#8]
1037    strb    r5,[r12,#9]
1038    mov r4,r3,lsr#24
1039    strb    r6,[r12,#10]
1040    mov r5,r3,lsr#16
1041    strb    r2,[r12,#11]
1042    mov r6,r3,lsr#8
1043    strb    r4,[r12,#12]
1044    strb    r5,[r12,#13]
1045    strb    r6,[r12,#14]
1046    strb    r3,[r12,#15]
1047 #endif
1048 #if __ARM_ARCH__>=5
1049    ldmia sp!,{r4-r12,pc}
1050 #else
1051    ldmia   sp!,{r4-r12,lr}
1052    tst lr,#1
1053    moveq pc,lr     @ be binary compatible with V4, yet
1054    .word 0xe12fff1e        @ interoperable with Thumb ISA:-)
1055 #endif
1056 .size AES_decrypt,.-AES_decrypt
1057
1058 .type   _armv4_AES_decrypt,%function
1059 .align  2
1060 _armv4_AES_decrypt:
1061    str lr,[sp,#-4]!     @ push lr
1062    ldmia r11!,{r4-r7}
1063    eor r0,r0,r4
1064    ldr r12,[r11,#240-16]
1065    eor r1,r1,r5
1066    eor r2,r2,r6
1067    eor r3,r3,r7
1068    sub r12,r12,#1
1069    mov lr,#255
1070
1071    and r7,lr,r0,lsr#16
1072    and r8,lr,r0,lsr#8
1073    and r9,lr,r0
1074    mov r0,r0,lsr#24
1075 .Ldec_loop:
1076    ldr r4,[r10,r7,lsl#2] @ Td1[s0>>16]
1077    and r7,lr,r1     @ i0
1078    ldr r5,[r10,r8,lsl#2] @ Td2[s0>>8]
```

```
1079    and  r8,lr,r1,lsr#16
1080    ldr  r6,[r10,r9,lsl#2] @ Td3[s0>>0]
1081    and  r9,lr,r1,lsr#8
1082    ldr  r0,[r10,r0,lsl#2] @ Td0[s0>>24]
1083    mov  r1,r1,lsr#24
1084
1085    ldr  r7,[r10,r7,lsl#2] @ Td3[s1>>0]
1086    ldr  r8,[r10,r8,lsl#2] @ Td1[s1>>16]
1087    ldr  r9,[r10,r9,lsl#2] @ Td2[s1>>8]
1088    eor  r0,r0,r7,ror#24
1089    ldr  r1,[r10,r1,lsl#2] @ Td0[s1>>24]
1090    and  r7,lr,r2,lsr#8  @ i0
1091    eor  r5,r8,r5,ror#8
1092    and  r8,lr,r2      @ i1
1093    eor  r6,r9,r6,ror#8
1094    and  r9,lr,r2,lsr#16
1095    ldr  r7,[r10,r7,lsl#2] @ Td2[s2>>8]
1096    eor  r1,r1,r4,ror#8
1097    ldr  r8,[r10,r8,lsl#2] @ Td3[s2>>0]
1098    mov  r2,r2,lsr#24
1099
1100    ldr  r9,[r10,r9,lsl#2] @ Td1[s2>>16]
1101    eor  r0,r0,r7,ror#16
1102    ldr  r2,[r10,r2,lsl#2] @ Td0[s2>>24]
1103    and  r7,lr,r3,lsr#16 @ i0
1104    eor  r1,r1,r8,ror#24
1105    and  r8,lr,r3,lsr#8  @ i1
1106    eor  r6,r9,r6,ror#8
1107    and  r9,lr,r3      @ i2
1108    ldr  r7,[r10,r7,lsl#2] @ Td1[s3>>16]
1109    eor  r2,r2,r5,ror#8
1110    ldr  r8,[r10,r8,lsl#2] @ Td2[s3>>8]
1111    mov  r3,r3,lsr#24
1112
1113    ldr  r9,[r10,r9,lsl#2] @ Td3[s3>>0]
1114    eor  r0,r0,r7,ror#8
1115    ldr  r7,[r11],#16
1116    eor  r1,r1,r8,ror#16
1117    ldr  r3,[r10,r3,lsl#2] @ Td0[s3>>24]
1118    eor  r2,r2,r9,ror#24
1119
1120    ldr  r4,[r11,#-12]
1121    eor  r0,r0,r7
1122    ldr  r5,[r11,#-8]
1123    eor  r3,r3,r6,ror#8
1124    ldr  r6,[r11,#-4]
1125    and  r7,lr,r0,lsr#16
1126    eor  r1,r1,r4
1127    and  r8,lr,r0,lsr#8
1128    eor  r2,r2,r5
1129    and  r9,lr,r0
1130    eor  r3,r3,r6
```

```
1131    mov r0,r0,lsr#24
1132
1133    subs   r12,r12,#1
1134    bne .Ldec_loop
1135
1136    add r10,r10,#1024
1137
1138    ldr r5,[r10,#0]    @ prefetch Td4
1139    ldr r6,[r10,#32]
1140    ldr r4,[r10,#64]
1141    ldr r5,[r10,#96]
1142    ldr r6,[r10,#128]
1143    ldr r4,[r10,#160]
1144    ldr r5,[r10,#192]
1145    ldr r6,[r10,#224]
1146
1147    ldrb   r0,[r10,r0]    @ Td4[s0>>24]
1148    ldrb   r4,[r10,r7]    @ Td4[s0>>16]
1149    and r7,lr,r1      @ i0
1150    ldrb   r5,[r10,r8]    @ Td4[s0>>8]
1151    and r8,lr,r1,lsr#16
1152    ldrb   r6,[r10,r9]    @ Td4[s0>>0]
1153    and r9,lr,r1,lsr#8
1154
1155    ldrb   r7,[r10,r7]    @ Td4[s1>>0]
1156    ldrb   r1,[r10,r1,lsr#24]  @ Td4[s1>>24]
1157    ldrb   r8,[r10,r8]    @ Td4[s1>>16]
1158    eor r0,r7,r0,lsl#24
1159    ldrb   r9,[r10,r9]    @ Td4[s1>>8]
1160    eor r1,r4,r1,lsl#8
1161    and r7,lr,r2,lsr#8  @ i0
1162    eor r5,r5,r8,lsl#8
1163    and r8,lr,r2      @ i1
1164    ldrb   r7,[r10,r7]    @ Td4[s2>>8]
1165    eor r6,r6,r9,lsl#8
1166    ldrb   r8,[r10,r8]    @ Td4[s2>>0]
1167    and r9,lr,r2,lsr#16
1168
1169    ldrb   r2,[r10,r2,lsr#24]  @ Td4[s2>>24]
1170    eor r0,r0,r7,lsl#8
1171    ldrb   r9,[r10,r9]    @ Td4[s2>>16]
1172    eor r1,r8,r1,lsl#16
1173    and r7,lr,r3,lsr#16 @ i0
1174    eor r2,r5,r2,lsl#16
1175    and r8,lr,r3,lsr#8  @ i1
1176    ldrb   r7,[r10,r7]    @ Td4[s3>>16]
1177    eor r6,r6,r9,lsl#16
1178    ldrb   r8,[r10,r8]    @ Td4[s3>>8]
1179    and r9,lr,r3      @ i2
1180
1181    ldrb   r9,[r10,r9]    @ Td4[s3>>0]
1182    ldrb   r3,[r10,r3,lsr#24]  @ Td4[s3>>24]
```

```
1183    eor  r0,r0,r7,lsl#16
1184    ldr  r7,[r11,#0]
1185    eor  r1,r1,r8,lsl#8
1186    ldr  r4,[r11,#4]
1187    eor  r2,r9,r2,lsl#8
1188    ldr  r5,[r11,#8]
1189    eor  r3,r6,r3,lsl#24
1190    ldr  r6,[r11,#12]
1191
1192    eor  r0,r0,r7
1193    eor  r1,r1,r4
1194    eor  r2,r2,r5
1195    eor  r3,r3,r6
1196
1197    sub  r10,r10,#1024
1198    ldr  pc,[sp],#4     @ pop and return
1199  .size  _armv4_AES_decrypt,.-_armv4_AES_decrypt
1200  .asciz  "AES for ARMv4, CRYPTOGAMS by <appro@openssl.org>"
1201  .align  2
```

Listing B.1 – aes_armv4_pipeline.S