Capítulo 3

Implementação Eficiente e Segura de Algoritmos Criptográficos

Armando Faz Hernández, Roberto Cabral, Diego F. Aranha, Julio López

Abstract

Software implementation of a cryptographic algorithm is not an easy task even for advanced programmers, because it requires a careful knowledge not only about algorithms but also of the target architecture. In this tutorial, we will describe some techniques to produce an efficient and secure software implementation. For the sake of efficiency, we will detail how advanced vector instruction sets accelerate the execution of the following cryptographic algorithms: the AES encryption algorithm, the SHA-3 cryptographic hash function and the key agreement protocol based on the elliptic curve Curve25519. Focusing on the secure software development, we will illustrate some implementations that are vulnerable against side-channel attacks; also we will present some countermeasures that mitigate such attacks thereby preventing leakage of secret information.

Resumo

A implementação segura de um algoritmo criptográfico não é uma tarefa trivial nem mesmo para os programadores mais experientes, pois requer um conhecimento cuidadoso não apenas do próprio algoritmo, mas também da arquitetura alvo. Neste minicurso, vamos nos concentrar em descrever os aspectos que ajudam a tornar uma implementação de criptografia em software eficiente e segura. Do lado da eficiência, detalharemos como os conjuntos avançados de instruções aceleram a execução dos algoritmos criptográficos a seguir: o algoritmo de encriptação AES, a função de resumo SHA-3 e o protocolo de acordo de chaves baseado na curva elíptica Curve25519. Pensando no desenvolvimento seguro, mostraremos como algumas implementações são vulneráveis contra ataques de canais laterais, adicionalmente apresentaremos técnicas que mitigam ditos ataques evitando assim o vazamento de informação secreta.

 ©2015 SBC — Soc. Bras. de Computação

3.1. Introdução à criptografia

Nesta seção, serão apresentados um conjunto de conceitos básicos sobre criptografia que serão usados ao longo deste capítulo. Espera-se que leitores familiarizados com o tema possam relembrar os conceitos básicos e propriedades dos algoritmos criptográficos, e leitores que estão tendo contato com o tema pela primeira consigam uma base teórica mínima necessária para entender o capítulo.

3.1.1. Conceitos básicos

O modelo de comunicação mais simples consiste em duas entidades, A e B, querendo se comunicar. Neste modelo a entidade A (emissor) quer enviar uma mensagem para a entidade B (receptor) através de um meio de comunicação; esse meio pode ser acessado por uma terceira entidade E (também chamada de intruso ou adversário). Visando facilitar a descrição de protocolos de comunicação, denotamos as entidades A, B e E por *Alice*, *Bob* e *Eva*, respectivamente.

Um dos problemas presente neste modelo de comunicação é assegurar que uma mensagem enviada por Alice seja lida unicamente por Bob, que a mensagem recebida por Bob tenha sido gerada por Alice e que a mensagem não tenha sofrido qualquer alteração no caminho.

A criptografia moderna pode ser vista como um conjunto de técnicas matemáticas que permitem estabelecer uma comunicação segura na presença de adversários maliciosos. Ela engloba mecanismos para assegurar a integridade de dados, técnicas para troca de chave secreta, protocolos para autenticar usuários, votação eletrônica, moeda eletrônica, entre outros.

Uma comunicação segura deve possuir os seguintes serviços segurança:

- **Sigilo.** A informação é mantida em segredo de todos, exceto das partes autorizadas.
- **Autenticação.** Deve ser possível garantir a autenticidade do remetente de uma mensagem; um atacante não deve ser capaz de se passar por outra pessoa.
- **Integridade.** O receptor deve conseguir verificar se a mensagem recebida não foi modificada durante a transmissão; um atacante não deve ser capaz de substituir uma mensagem falsa pela legítima.
- **Irretratabilidade.** Garante que uma entidade não pode negar ter participado de uma comunicação.

De acordo com o tipo de chave usado, os métodos criptográficos podem ser subdivididos em duas grandes categorias: criptografia simétrica e criptografia assimétrica:

- **Criptografia Simétrica**. Consiste nas funções de encriptação que usam a mesma chave para encriptar e decriptar. Esses algoritmos requerem que o

emissor e o receptor estabeleçam uma chave para realizar a comunicação encriptada. Esta parte da criptografia é também conhecida como criptografia de chave secreta no sentido que a chave só pode ser conhecida pelos envolvidos na comunicação, pois toda a segurança do esquema baseia-se na chave criptográfica, uma vez que qualquer pessoa de posse da chave consegue encriptar ou decriptar mensagens.

- **Criptografia Assimétrica**. Também conhecida como criptografia de chave pública, considera o uso de duas chaves distintas, uma pública e uma privada, para cada participante da comunicação; sendo que a chave pública pode ser divulgada e a chave privada deve ser mantida em segredo. Assim, uma mensagem que é encriptada com a chave pública, será decriptada somente com o uso da correspondente chave privada. A geração do par de chaves deve garantir que a chave privada não possa ser obtida a partir da chave pública. Na criptografia assimétrica as operações de encriptação podem ser realizadas por qualquer entidade, contudo as operações de decriptação são restritas à entidade que possui a chave privada. Dentre os usos da criptografia assimétrica encontram-se a encriptação de dados, a geração de assinaturas digitais, o estabelecimento seguro de chaves, entre outros.

Os protocolos criptográficos podem ser construídos a partir de primitivas criptográficas. Um protocolo criptográfico é composto por uma série de computações e comunicações entre duas ou mais entidades com o fim de realizar uma tarefa especifica. Por exemplo: transmitir dados de forma segura, verificar a integridade de uma mensagem ou estabelecer chaves privadas mediante um canal inseguro.

3.1.2. Encriptação de dados

A encriptação de dados tem como objetivo prover sigilos das mensagens quando se usa um meio de comunicação inseguro. Portanto, o processo de *encriptação* (as vezes também denominado como cifração) consiste em converter um texto claro em um texto encriptado usando uma chave secreta. O processo inverso, conhecido por *decriptação*, transforma um texto encriptado em um texto claro usando a mesma chave usada na encriptação.

Um dos princípios fundamentas da criptografa foi postulado em 1883 por Kerckhoff e diz que a segurança do processo de encriptação não deve estar baseada apenas em manter em segredo o funcionamento interno do algoritmo. Na criptografia moderna, os algoritmos devem ser conhecidos por toda a comunidade e a segurança dos mesmos deve estar atrelada a uma *chave criptográfica*.

Em termos matemáticos o texto claro é representado por M e o texto encriptado por C. Portanto, a função de encriptação E usa uma chave criptográfica k e opera sobre M para produzir $C = E_k(M)$. No processo reverso, tem-se a função D que usa a mesma chave k que foi usada para encriptar e opera sobre C para produzir $M = D_k(C)$. A encriptação e decriptação são operações inversas, por conta disso, a identidade $M = D_k(E_k(M))$ deve ser verdadeira para quaisquer valores de M e k.

Se uma função de encriptação possui o mesmo tamanho de entrada e saída,

esta função também é conhecida como *cifrador de bloco*; alguns exemplos de cifradores de bloco são: o *Digital Encryption Standard* (DES) e o *Advanced Encryption Standard* (AES) com blocos de tamanho 64 e 128 bits, respectivamente. Para encriptar mensagens de tamanho arbitrário é possível usar *modos de operação*, os quais usam os cifradores de bloco como primitiva básica. Esses modos serão examinados com mais detalhe na Seção 3.4.1.2.

Um dos ataques que pode ser aplicado à grande maioria dos cifradores é o ataque de força bruta, que consiste em achar a chave usada na encriptação, dado um texto claro e um texto encriptado, testando cada uma das chaves possíveis. A complexidade desse ataque é 2^n, onde n é o tamanho da chave. Um cifrador possui um nível de segurança de n bits se o melhor algoritmo capaz de encontrar a chave criptográfica possui uma complexidade de $O(2^n)$.

3.1.3. Funções de resumo criptográfico

As funções de resumo criptográfico agem como uma função de compressão que mapeia uma cadeia de bits de tamanho arbitrário em uma cadeia de bits de tamanho fixo. Elas são muito usadas na criptografia moderna e seu uso é essencial em várias aplicações de segurança, tais como: esquemas de assinatura digital, verificação da integridade dos dados, geração de números pseudo aleatórios, geração de chaves, dentre outras.

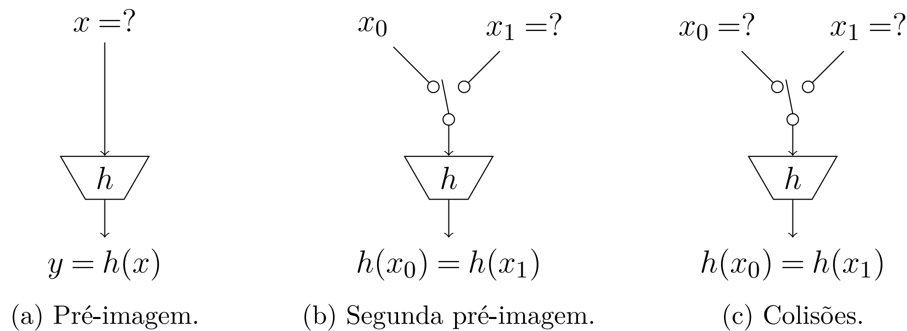
A cadeia de bits gerada por uma função de resumo é chamada de valor de resumo ou *digest*; e teoricamente, deve identificar à mensagem de forma única. Uma função de resumo $h : M \rightarrow R$ mapeia uma cadeia de bits $x \in M$ de tamanho finito e arbitrário em uma cadeia de bits $y \in R$ de tamanho fixo n. Tendo, $y = h(x)$.

Pode-se notar que mais de uma mensagem poderá ser mapeada em um mesmo valor de resumo, visto que o domínio M de h é maior que sua imagem R. Porém, algumas aplicações, como as assinaturas digitais, por exemplo, requerem que seja computacionalmente inviável encontrar duas mensagens diferentes que gerem o mesmo valor de resumo; outras aplicações apenas necessitam que seja inviável encontrar uma mensagem dado o valor de resumo.

As primeiras definições, analises e construções de funções de resumo criptográficos podem ser encontradas nos trabalhos de Rabin [Rab78], Yuval [Yuv78] e Merkle [Mer79]. Rabin propôs um modelo baseado no algoritmo de encriptação DES; Yuval usou o paradoxo de aniversário para mostrar como encontrar colisões para uma função de resumo de n bits com $2^{n/2}$ operações; e Merkle introduziu que esse tipo de função deveria ser resistente à colisões, primeira pré-imagem e segunda pré-imagem.

As funções de resumo criptográfico, diferentemente dos algoritmos de encriptação, não possuem chaves secretas. Assim, para termos uma função de resumo segura é necessário que satisfaça as seguintes propriedades de segurança, ilustradas na Figura 3.1:

- **Resistência à pré-imagem**: Dada uma função $h : M \rightarrow R$ e um valor de resumo $y \in R$, é computacionalmente inviável encontrar $x \in M$ tal que $h(x) = y$.



(a) Pré-imagem. (b) Segunda pré-imagem. (c) Colisões.

Figura 3.1: As três propriedades de segurança de uma função de resumo.

- **Resistência à segunda pré-imagem**: Dada uma função $h : M \to R$ e uma mensagem $x_0 \in M$, é computacionalmente inviável encontrar uma mensagem $x_1 \in M$ tal que $x_0 \neq x_1$ e $h(x_0) = h(x_1)$.
- **Resistência à colisão**: Dada uma função $h : M \to R$, é computacionalmente inviável encontrar duas mensagens $x_0, x_1 \in M$ tal que $x_0 \neq x_1$ e $h(x_0) = h(x_1)$.

Em 1993, a função de resumo chamada *Secure Hash Algorithm* (SHA) foi desenvolvida pelo *National Institute of Standards and Technology* (NIST) e publicada como padrão nacional americano para o processamento seguro de informações [Nat93]. Posteriormente esta versão foi revisada e ficou conhecida por SHA-1 [Nat95]. Já em 2002, o NIST definiu três novas funções de resumo, conhecidas por SHA-2, com tamanhos de valor de resumo de 256, 384 e 512 bits [Nat02], e em 2008 mais uma função de resumo com tamanho de saída de 224 bits [Nat08a]. Em 2015, uma terceira versão do padrão, chamada SHA-3, foi publicada [Nat08b, Nat15].

3.1.4. Protocolo de acordo de chaves

Suponha que Alice e Bob querem compartilhar uma chave secreta para ser usada em uma função de encriptação simétrica, sendo que o único meio de comunicação é inseguro, ou seja qualquer informação transmitida pode ser observada por Eva. Esse problema é comumente conhecido como acordo de chaves e foi engenhosamente resolvido por Diffie e Hellman em 1976 [DH76].

Eles propuseram uma solução para esse problema observando que a computação de algumas funções possuem uma certa assimetria; isto é, dado uma entrada x é fácil calcular $y = f(x)$, entretanto dado y é computacionalmente inviável obter x tal que $x = f^{-1}(y)$. Por exemplo, obter o produto de dois números primos grandes é simples, mas é difícil recuperar os números primos que compõem o produto. Diffie e Hellman observaram que este fenômeno poderia ser usado para estabelecer um protocolo de troca de chaves seguro. O *protocolo Diffie-Hellman* está descrito a seguir:

1. Alice e Bob negociam os seguintes parâmetros de domínio: um número primo p grande e um gerador g do grupo cíclico $\mathbb{Z}_p^*$.

2. Alice escolhe aleatoriamente um valor $x \in \mathbb{Z}_p$, computa $k_A = g^x$ e envia o resultado para Bob.

3. Bob atua de forma similar, escolhendo aleatoriamente um valor $y \in \mathbb{Z}_p$, computando $k_B = g^y$ e enviando o resultado para a Alice.

4. Bob recebe o valor k_A da Alice e computa k_A^y. De forma análoga, Alice computa k_B^x com o valor recebido pelo Bob.

5. Neste ponto, o seguinte valor $k_B^x = k_A^y = g^{xy}$ é o segredo compartilhado que pode ser usado para gerar uma chave privada.

A intuição atrás deste protocolo consiste na existência de uma função eficiente para calcular a exponenciação de elementos em $\mathbb{Z}_p^*$; no entanto, a computação de sua função inversa, conhecida como o logaritmo discreto, é computacionalmente inviável. Assim, qualquer atacante que interceptar as mensagens transmitidas deve calcular g^{xy} a partir de g, g^x e g^y.

Originalmente, Diffie e Hellman propuseram o uso do grupo multiplicativo dos inteiros modulo p para a implementação do protocolo. Embora, existam outros grupos onde a operação de exponenciação é mais eficiente; por exemplo, o grupo formado pelo conjunto de pontos de uma curva elíptica. Se a curva elíptica possui certas propriedades que tornam o logaritmo discreto inviável, então esse grupo é uma opção viável para a implementação do protocolo de acordo de chaves, o qual é comumente chamado de protocolo *Diffie-Hellman baseado em curvas elípticas* (ECDH).

3.1.5. Curvas elípticas

Uma curva elíptica E está definida sobre um corpo finito primo $\mathbb{F}_p$ pela seguinte equação:

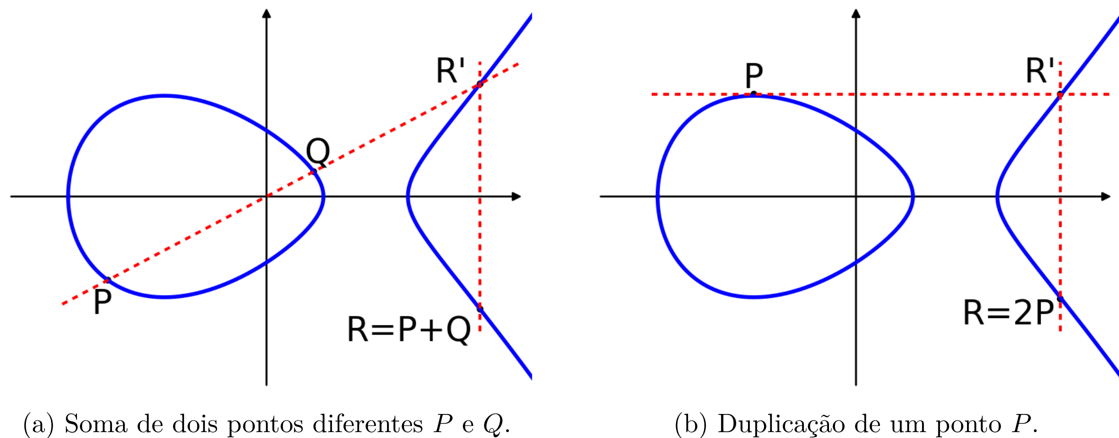
$$E\colon y^2 = x^3 + Ax + B \tag{1}$$

tal que $A, B \in \mathbb{F}_p$ e $4A^3 + 27B^2 \neq 0$. Os pontos pertencentes à curva elíptica E, junto como o ponto no infinito, formam um grupo cíclico:

$$E(\mathbb{F}_p) = \{(x,y)\colon x,y \in \mathbb{F}_p \text{ tal que } y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\}, \tag{2}$$

onde $\mathcal{O}$ é o elemento identidade do grupo. O número de elementos deste conjunto está delimitado pelo intervalo de Hasse: $\#E(\mathbb{F}_p) = p + 1 \pm 2\sqrt{p}$. Usualmente a operação de grupo é expresso de forma aditiva; assim a lei do grupo é a soma de dois pontos $P, Q \in E(\mathbb{F}_p)$ denotado como $P + Q$.

Define-se geometricamente a soma de dois pontos da seguinte forma: trace uma linha reta que passe por P e Q, esta linha irá intersectar E em um ponto R'; agora trace uma linha vertical sobre o ponto R'; a linha intersectará à curva E no ponto R. Então, o ponto R representará a soma $P + Q$. O caso especial da soma de um ponto consigo mesmo é conhecido como duplicação de pontos. Geometricamente o cálculo de $2P = P + P$ segue o mesmo conceito descrito; embora ao invés de traçar uma linha que passe por dois pontos, deve-se traçar a linha tangente à curva E no ponto P. Essa linha irá intersectar à curva em um ponto R'; do qual se trace uma



(a) Soma de dois pontos diferentes P e Q. (b) Duplicação de um ponto P.

Figura 3.2: Representação geométrica da lei do grupo em uma curva elíptica definida sobre $\mathbb{R}$.

linha vertical que cruza à curva num ponto R; este último representará a computação de $2P$; na Figura 3.2 é mostrada de forma gráfica a lei do grupo numa curva elíptica definida sobre $\mathbb{R}$.

A soma repetida de um ponto P, denominada por multiplicação de um ponto P por um inteiro k, está definida como:

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ vezes}}.$$

Um dos métodos mais eficientes para computar a multiplicação de pontos é o algoritmo *duplicação-e-soma*, chamado assim porque utiliza as funções de duplicação e soma, antes mencionadas. O algoritmo inicializa um ponto $Q = \mathcal{O}$ que serve como acumulador. Uma vez feito isso, o processamento lê os bits de k iterativamente, do mais ao menos significativo, de modo que a cada iteração i o ponto Q é duplicado e se o i-ésimo bit de k é igual a 1, o ponto P é somado ao acumulador. Uma vez que todos os bits foram lidos, o ponto Q conterá o ponto kP.

A obtenção de múltiplos de um ponto é uma operação que pode ser computada eficientemente, no entanto acredita-se que a operação inversa para certas curvas elípticas é computacionalmente inviável. Dado um ponto G que seja gerador do grupo elíptico $E(\mathbb{F}_p)$ e um ponto qualquer da curva $Q \in E(\mathbb{F}_p)$ determinar um $k \in \mathbb{Z}$ tal que $Q = kG$ é conhecido como o *Problema do Logaritmo Discreto em Curvas Elípticas* (ECDLP). Os algoritmos mais eficientes que resolvem ECDLP possuem uma complexidade de $O(\sqrt{\#E(\mathbb{F}_p)})$. Por exemplo, se $p \approx 2^{256}$ então a complexidade de resolvê-lo requer aproximadamente 2^{128} operações, tornando o calculo de k inviável.

3.2. Instruções avançadas dos processadores

Nesta seção serão descritos alguns conjuntos avançados de instruções que apareceram nos processadores atuais; por exemplo, os conjuntos de instruções AES-NI,

SSE, AVX e o mais recente, AVX2. O leitor será familiarizado a terminologia básica das instruções avançadas dos processadores e será apresentada a evolução das instruções vetoriais nas micro-arquiteturas; em particular a ênfase será dada à micro-arquitetura Haswell. Finalmente, será discutido um exemplo de como detectar a presença dos conjuntos avançados de instruções nos processadores.

3.2.1. Instruções vetoriais

No fim da década de 1990, os fabricantes de processadores focaram seus esforços em explorar o paralelismo de dados ao invés do paralelismo de instruções como era feito nas arquiteturas RISC. Para isso, eles incorporaram unidades funcionais que são capazes de executar uma única instrução sobre um conjunto de dados; esse processamento encaixa-se no paradigma conhecido como *Single Instruction Multiple Data* (SIMD), introduzido em [Fly72].

Um dos primeiros conjuntos de instruções a implementar o paradigma SIMD foi lançado em 1997, conhecido como *Multimedia eXtensions* (MMX) [Cor]. O MMX adicionou registradores de 64 bits e instruções vetoriais que habilitavam o processamento de duas operações de 32 bits, nessa época as arquiteturas possuíam registradores nativos de 32 bits. Essas instruções definem a semântica do conteúdo do registrador, isto é, um registrador de 64 bits poderia ser operado como um vetor de oito palavras de 8 bits, um vetor de quatro palavras de 16 bits, um vetor de duas palavras de 32 bits ou como um vetor de 64 bits. No começo as instruções MMX foram voltadas para auxiliar o processamento de aplicações gráficas e multimídia.

Nos últimos 18 anos, tanto a Intel como a AMD lançaram novos conjuntos de instruções vetoriais para explorar ainda mais o paralelismo a nível de dados. A seguir serão apresentados alguns dos conjuntos mais relevantes lançados por essas duas companhias:

- Em 1999, a Intel lançou *Streaming SIMD Extensions* (SSE) que adicionou oito registradores de 128 bits (denotados `XMM0-XMM7`) e incluiu instruções para dar suporte à computação de aritmética de ponto flutuante.
- No ano 2000, a AMD desenvolveu a arquitetura de 64 bits, com isso o tamanho dos registradores nativos aumentou de 32 bits para 64 bits e o número de registradores vetoriais foi duplicado (denotados `XMM0-XMM15`).
- O conjunto de instruções SSE foi evoluindo com o lançamento dos novos conjuntos SSE2, SSE3, SSSE3 e SSE4. O SSE2 foi lançado para dar suporte às operações de aritmética inteira. Os outros conjuntos, entretanto, foram incorporando outros tipos de instruções vetoriais; desse modo, começou a surgir instruções para manipulação de cadeias de caracteres, permutação de palavras dentro dos registradores e códigos de correção de erros.
- A diversidade das instruções aumentou ainda mais em 2010, quando foi incluído o *Advanced Encryption Standard New Instructions* (AES-NI) que acelera a computação do algoritmo de encriptação AES. Este conjunto será amplamente explicado na Seção 3.4.1.

- Já em 2011, o conjunto *Advanced Vector Extensions* (AVX) fez contribuições relevantes na arquitetura. Pois incluiu registradores de 256 bits, chamados `YMM`, que encontram-se sobrepostos sobre os registradores `XMM`. Além disso, AVX introduziu um novo formato de codificação que permite utilizar código de montagem de três operandos, o que torna a atribuição de registradores mais flexível.

- O mais recente conjunto de instruções é a segunda versão do AVX, chamada AVX2. Esse conjunto será detalhado na seção seguinte.

Como foi apresentado, as primeiras instruções vetoriais foram direcionadas ao processamento gráfico, esse cenário vem mudando nos últimos anos e agora está disponível uma vasta diversidade de instruções. Neste documento, serão apresentadas técnicas para se tirar proveito destes conjuntos na implementação eficiente de algoritmos criptográficos.

3.2.2. O conjunto de instruções AVX2

A microarquitetura Haswell foi lançada no início de 2013 e trouxe consigo uma série de inovações para acelerar a execução dos programas. Dentre as quais destacam-se: a inclusão de mais duas portas de execução, um novo multiplicador inteiro, as instruções de manipulação de bits, entre outras. No entanto, a característica mais relevante do Haswell é o suporte ao conjunto de instruções vetoriais AVX2.

O AVX2 contém novas instruções que expandem a computação de aritmética inteira nos registradores de 256 bits; pois o conjunto AVX continha instruções apenas para a aritmética de ponto flutuante. Além disso, AVX2 conta com instruções de permutação e combinação, que permitem movimentar as palavras contidas nos registradores vetoriais, entre outras características. A seguir, são destacadas algumas instruções que fazem parte do conjunto AVX2 e que são relevantes para o contexto de implementação eficiente de algoritmo criptográficos:

1. Acesso a memória.

   (a) `LOAD/STORE`. Essas funções carregam/armazenam um conjunto de dados de/para um endereço de memória de/para um registrador vetorial. Vale a pena mencionar que o endereço de memória deve estar alinhado para 32 bits, ou seja, o valor de endereço deve ser um múltiplo de 32, caso contrário o desempenho da aplicação é afetado.

   (b) `BRCAST`. Essa instrução replica um valor de 64 bits de um registrador nativo (ou endereço de memória) em um registrador vetorial.

2. Aritmética inteira.

   (a) `ADD/SUB`. Essas funções calculam a adição e subtração de números inteiros armazenados nas palavras de um registrador vetorial. Por exemplo, a instrução pode computar oito operações de 32 bits ou quatro operações de 64 bits.

   (b) `MUL`. Usando essa instrução é possível calcular quatro multiplicações de palavras de 32 bits; os quatro produtos de 64 bits gerados são armazenados em um registrador de 256 bits.

3. Funções lógicas.
   (a) `XOR/AND/OR/ANDNOT`. Essas funções calculam operações lógicas de 256 bits.
   (b) `SHL/SHR`. Para cada palavra armazenada em um registrador vetorial, essas instruções deslocam para esquerda ou direta, respectivamente, uma quantidade fixa de bits.
   (c) `SHLV/SHRV`. Essas instruções potencializam o processamento paralelo das instruções de deslocamento. Porque ao invés de fazer um deslocamento fixo, a instrução recebe, além do registrador alvo, um segundo registrador contendo a quantidade de bits a serem deslocados; desta forma, cada palavra do vetor alvo será deslocada de acordo com às quantidades especificadas no segundo registrador.

4. Permutações internas no registrador.
   (a) `PERM`. Instruções para embaralhar a posição das palavras dentro do registrador são chamadas de permutações. No entanto, nas primeiras versões do SSE e AVX, foram também conhecidos como *shuffle* ou *shuffling*. Durante o texto o termo permutação será usado para se referir a esse tipo de instruções.

5. Combinação de registradores.
   (a) `UPCK`. Essas instruções preenchem um registrador intercalando as palavras da parte alta/baixa de outros dois registradores de origem.
   (b) `BLEND`. Esse tipo de instrução preenche um registrador a partir de palavras contidas em dois registradores diferentes; a escolha está baseada no valor de uma máscara binária.
   (c) `PRBLEND`. Essa instrução combina partes de 128 bits de dois registradores de 256 bits em um novo registrador de 256 bits; a escolha está baseada no valor de uma máscara binária.
   (d) `ALIGNR`. Essa instrução concatena dois registradores de 128 bits e desloca n bytes, os 128 bits menos significativos do valor intermediário gerado são armazenado no registrador destino.

6. Conversão de elementos.
   (a) `CAST`. São pseudo-instruções que mudam a semântica do conteúdo dos registradores. A maioria dessas instruções são tratadas diretamente pelos compiladores e não geram nenhuma instrução em código de montagem.

Instrução vetorial	Latência	Vazão	Porta de execução							
			0	1	2	3	4	5	6	7
`LOAD`	3	2			$\times$	$\times$				
`STORE`	3	1			$\times$	$\times$	$\times$			$\times$
`BRCAST`	5	2			$\times$	$\times$				
`ADD/SUB`	1	2		$\times$				$\times$		
`MUL`	5	1	$\times$							
`XOR`	1	3	$\times$	$\times$				$\times$		
`SHL/SHR`	1	1						$\times$		
`SHLV SHRV`	2	0.5	$\times$					$\times$		
`PERM`	3	1						$\times$		
`BLEND`	1	3	$\times$	$\times$				$\times$		
`PRBLEND`	3	1						$\times$		
`ALIGNR`	1	1						$\times$		
`UPCK`	1	1						$\times$		
`CAST`	0	-								
`EXTRACT`	3	1						$\times$		
`INSERT`	3	1						$\times$		

Tabela 3.1: Latência, vazão (instruções executadas por ciclo quando não há dependências) e portas de execução de algumas instruções AVX2.

(b) `EXTRACT`. Essa instrução retorna um registrador de 128 bits composto pela parte alta (ou baixa) de um registrador de 256 bits. Diferente das instruções de `CAST`, esta é uma instrução explicita que é codificada em linguagem de montagem.

(c) `INSERT`. Essa instrução tem a funcionalidade inversa à instrução `EXTRACT`, pois ela permite adicionar o conteúdo de um registrador de 128 bits na parte alta (ou baixa) de um registrador de 256 bits.

Essa lista não inclui todas as instruções pertencentes ao conjunto de instruções AVX2; é recomendado consultar [Cor11] para informação adicional sobre o conjunto AVX2. Na Tabela 3.1 são mostradas as latências, a vazão e as portas de execução das instruções AVX2 acima citadas; essas informações foram extraídas do relatório técnico produzido por Agner Fog [Fog14].

3.2.3. Detecção dos conjuntos de instruções

Antes de usar um conjunto avançado de instruções é preciso verificar se tal conjunto é suportado pelo processador alvo. Existem várias formas de fazer essa verificação, caso o sistema operacional usado seja o Linux a maneira mais fácil é rodar o seguinte comando no terminal:

```
1 $ cat /proc/cpuinfo | grep flags
```

```
flags:  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
        pge mca cmov pat pse36 clflush dts acpi mmx fxsr
        sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
        pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
        tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2
        x2apic movbe popcnt tsc_deadline_timer aes xsave
        fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms
        invpcid rtm avx
```

Esse comando mostra as características do processador. Dentre as saídas temos o campo *flags*, onde é apresentado os conjuntos de instruções suportados. O processador onde esse comando foi executado é apresentado no Apêndice A e, como se pode perceber, o mesmo suporta todos os conjuntos de instruções aqui apresentados.

Outra forma de verificar se os conjuntos de instruções são suportados, independente de sistema operacional, é usando a instrução `CPUID` para verificar se o bit que identifica a presença de um conjunto de instruções está habilitado ou não. No seguinte trecho de código pode-se ver como usar `CPUID` para identificar se o conjunto de instrução AES-NI é suportado. Adicionalmente, recomenda-se a execução do programa *FeatureDetector*, que pode ser encontrado em [Yee15], para verificar se os outros conjuntos de instruções são suportados.

```c
#include <stdio.h>
#define cpuid( func , ax , bx , cx , dx ) \
  __asm__ __volatile__ (" cpuid " : \
    "=a" ( ax ) , "=b" ( bx ) , \
    "=c" ( cx ) , "=d" ( dx ) : \
    "a" ( func ));
int Check_CPU_support_AES() {
  unsigned int a ,b ,c , d ;
  cpuid (1 ,a ,b ,c , d );
  return ( c & 0x2000000 );
}
void main (){
  printf("O processador suporta AES-NI: %s\n",
  Check_CPU_support_AES() ? "Sim" : "Nao" );
}
```

3.3. Técnicas para implementação segura

Utilizar criptografia de maneira segura na prática é não-trivial. Algoritmos criptográficos fornecem suas propriedades de segurança apenas em condições muito precisas, quanto todas as premissas são devidamente satisfeitas. Alguns erros comuns observados em sistemas em produção quando da utilização de algoritmos criptográficos são a geração, distribuição e armazenamento inseguros de chaves criptográficas, a escolha inadequada de primitivas, a utilização de algoritmos obsoletos, o desenvolvimento de algoritmos e implementações próprias sem revisão externa, a validação descuidada da titularidade de chaves públicas e certificados digitais, e o vazamento

de informação sensível por canais laterais [Sta10] ou recuperação de dados latentes da memória principal [HSH+09]. Como este capítulo se restringe à implementação segura e eficiente dos algoritmos criptográficos propriamente ditos, sem tratar em detalhes de sua utilização correta, a discussão será limitada ao escopo de implementação e mitigação de ataques de canal lateral.

3.3.1. Ataques de canal lateral

Eficiência não é o único requisito para aplicações práticas de criptografia, especialmente em sistemas embarcados que expõem superfícies de ataque consideráveis e alto potencial para manipulação direta por agentes maliciosos. Intervenções desse tipo podem incluir a medição em tempo real de características como tempo de execução, consumo de energia, emanações acústicas e eletro-magnéticas; e injeção de falhas para interromper a operação normal de uma função criptográfica, sempre na esperança de extrair informação secreta e material de chave diretamente do estado interno do algoritmo [Sta10]. É possível mitigar essas ameaças com o projeto cuidadoso das implementações de algoritmos criptográficos, aplicando técnicas para execução regular e invariante no tempo, de forma a desassociar padrões de computação com informação sensível.

Algumas vezes os recursos necessários para se implementar um algoritmo criptográfico de forma segura já estão disponíveis na plataforma, na forma de instruções dedicadas ou organização da arquitetura. Na maioria dos casos, o implementador precisa tomar cuidado adicional para verificar se suas contramedidas são corretamente traduzidas em linguagem de máquina pelo compilador ou utilizar linguagem de montagem diretamente. Em software, o escopo do que pode ser feito é bastante limitado, pois é difícil ter controle de granularidade fina sobre todos os aspectos da execução. Por essa razão, a ênfase será dada a contramedidas simples que protegem implementações contra ataques de canal lateral baseados no tempo.

3.3.2. Contramedidas algorítmicas

Implementações seguras de algoritmos criptográficos devem ser realizadas em tempo invariante com a informação sensível processada (dados a serem cifrados ou chave criptográfica), caso contrário um adversário capaz de monitorar o tempo de execução do algoritmo pode correlacionar suas observações com informação secreta. A monitoração não precisa ser necessariamente local e pode acontecer via iteração remota, pois já foi demonstrado que a latência de rede não insere ruído o suficiente para esconder pequenas variações do tempo de execução, mesmo que na granularidade de ciclos [BT11, BB05]. As variações ocorrem especialmente em trechos críticos que manipulam a chave privada, validam o formato de preenchimento (*padding*) ou conferem um valor de resumo ou autenticador.

Pequenas variações de tempo de execução podem se apresentar durante a comparação de cadeias de caracteres, desvios condicionais dependentes de bits da chave e limitantes do número de iterações de laços. No primeiro caso, se a interrupção do laço acontecer exatamente na primeira posição diferente entre as cadeias sendo comparadas, o adversário é capaz de realizar um ataque de busca exaustiva

em cada *byte* individualmente, baseado no tempo de resposta da comparação. Desta forma, é possível determinar o autenticador de uma mensagem, por exemplo, sem conhecimento da chave. O trecho de código abaixo ilustra uma forma segura de comparação:

```
1 int cmp_const(const void * a, const void * b,
2               const size_t size) {
3     const unsigned char *_a = a, *_b = b;
4     unsigned char result = 0;
5     size_t i;
6     for (i = 0; i < size; i++) {
7         result |= _a[i] ^ _b[i];
8     }
9     return result;
10 }
```

Desvios condicionais podem ser outra fonte de problemas [Koc96], especialmente quando se considera execução especulativa em processadores modernos ou o efeito da predição de desvios [AcKKS07]. A remoção de desvios condicionais envolve a aplicação de técnicas de programação sem desvios condicionais. A aplicação correta dessas técnicas é altamente dependente do algoritmo sendo estudado, mas uma generalização útil é calcular os dois ramos do desvio condicional simultaneamente e utilizar operações mascaradas para selecionar o valor correto apenas ao final da execução. A ideia é ilustrada pelo trecho de código abaixo para seleção entre dois valores em tempo constante, dependendo de um bit:

```
1 unsigned select(unsigned a, unsigned b,
2                 unsigned bit) {
3   /* -0 = 0, -1 = 0xFF....FF */
4   unsigned mask = - bit;
5   unsigned ret = mask & (a^b);
6   ret = ret ^ a;
7   return ret;
8 }
```

Outra possibilidade é utilizar uma variante da função de seleção para alterar em tempo constante os valores de entrada do trecho de código protegido ser executado. Por fim, cuidado adicional deve ser tomado também para não permitir que o número de iterações de laços sejam variáveis com informação sensível, por exemplo o comprimento de uma chave criptográfica em um sistema criptográfico assimétrico.

3.3.3. Contramedidas de acesso à memória

Acessos à memória devem também ser cuidadosamente protegidos. No contexto de cifras de bloco, a preocupação se concentra na representação de caixas de substituição como vetores ou introdução de qualquer tabela pré-calculada para acelerar operações sobre bits realizadas pela cifra. O algoritmo AES ilustra muito bem o problema, pois seu desempenho depende enormemente da eficiência das caixas de substituição, motivando o implementador a utilizar tabelas simples. Entretanto, um

adversário capaz de monitorar o comportamento da memória *cache* pode determinar que porções da caixa de substituição são usadas na etapa de cifração e recuperar informação sensível [Ber04, Per05, BM06, TOS10].

Existem contramedidas de diversos tipos para mitigar o problema. Uma opção simples é adotar arquiteturas com latência de acesso uniforme à memória, como alguns microcontroladores simples. Outra possibilidade é utilizar uma implementação em hardware do algoritmo, quando disponível, que deve oferecer uma superfície de ataque menor. Alternativas mais sofisticadas para implementação em software são *bitslicing*, onde as operações sobre bits são realizadas explicitamente, sem ajuda de tabelas pré-calculadas, e o impacto em desempenho é reduzido pela aplicação do mesmo circuito lógico a bits de diferentes variáveis simultaneamente [Bih97, KS09]. Para tabelas pequenas, também é possível utilizar uma instrução para acesso a tabela armazenada em um registrador [Ham09] ou percorrer a tabela inteira a cada leitura, utilizando a função de seleção apresentada anteriormente para realizar uma cópia condicional entre o valor lido e um valor atual da variável de interesse.

A implementação segura de criptografia assimétrica, é comum a necessidade de manipular vetores em memória em tempo constante. Por exemplo, algoritmos de exponenciação costumam calcular pequenos múltiplos da base para acelerar o cálculo de uma potência da chave privada. Se o adversário puder recuperar as posições de memória sendo acessadas ao longo das iterações do algoritmo, o expoente privado termina relevado. A mitigação mais comum para esse problema é percorrer toda a tabela utilizando uma variação da função de seleção para vetores, na forma de uma cópia condicional sem desvios, que terminará visitando toda a tabela em ordem determinística mas copiando apenas as posições desejadas:

```
1 void copy_cond(int *c, const int *a,
2                int  d, dig_t b) {
3   dig_t mask, t;
4   mask = -b;
5   for (int i = 0; i < d; i++) {
6     t = (a[i] ^ c[i]) & mask;
7     c[i] ^= t;
8   }
9 }
```

Como será visto adiante, outros algoritmos de exponenciação possuem padrão regular de execução em cada uma de suas iterações, mas as variáveis de entrada e saída podem ser diferentes dependendo dos bits da chave [Mon87]. Nesse caso, monitorar a latência de acesso à memória pode revelar ao adversário informação sobre a chave a partir dos acertos ou erros na memória *cache*. Uma solução para o problema, é realizar trocas condicionais entre as variáveis a cada iteração a partir de uma variante da cópia condicional encontrada abaixo. Por mais que as versões das funções sejam apresentadas para vetores de palavras do processador, é possível aprimorar seu desempenho pela implementação de versões de maior granularidade utilizando instruções vetoriais.

```
void swap_cond(int *c, int *a, int d, dig_t b) {
  dig_t mask, t;

  mask = -b;
  for (int i = 0; i < d; i++) {
    t = (a[i] ^ c[i]) & mask;
    a[i] ^= t;
    c[i] ^= t;
  }
}
```

Outras técnicas para implementação segura de algoritmos criptográficos podem ser encontradas em `https://cryptocoding.net/`.

3.4. Técnicas para acelerar operações criptográficas

O objetivo desta seção é apresentar exemplos concretos sobre a implementação de algoritmos criptográficos, e mostrar conjuntos avançados de instruções que podem ser usados para acelerar a implementação de: o algoritmo de encriptação AES, a função de resumo SHA-3 e o protocolo de acordo de chaves baseado na curva elíptica *Curve25519*.

Para cada algoritmo, serão descritas as operações necessárias para seu funcionamento; os detalhes de como implementá-lo eficientemente em uma arquitetura de 64 bits, que preferencialmente suporte os conjuntos avançados de instruções; e finalmente os resultados de desempenho serão reportados.

3.4.1. Aceleração do algoritmo de encriptação AES

Esta seção descreve como maximizar o uso do suporte nativo em *hardware* para acelerar a computação do algoritmo de encriptação AES. Inicialmente, as operações realizadas para encriptar um bloco de dados serão descritas e posteriormente como acelerar a execução destas operações através de instruções especiais presentes nos processadores atuais.

3.4.1.1. Descrição do algoritmo AES

O algoritmo *Advanced Encryption Standard* (AES) foi publicado pelo NIST em 2001 [Nat01a]. AES é um cifrador de bloco que encripta uma mensagem M de 128 bits usando uma chave k e produz um texto cifrado C de 128 bits. O tamanho da chave pode ser de 128, 192 ou 256 bits; o cifrador AES é denotado por AES-128, AES-192 ou AES-256 dependendo do tamanho da chave usada.

O Algoritmo 1 descreve a sequência de operações do AES, cuja entrada é uma chave k de 128, 192 ou 256 bits e uma mensagem de 128 bits, que pode ser visto como uma matriz S de 4×4 bytes, denotada por *estado*; o AES modifica o estado iterativamente usando um conjunto de operações, onde o número de iterações N depende do tamanho da chave. O estado é modificado a cada rodada pelas seguintes transformações:

- `SubBytes`. O estado é atualizado pela aplicação da caixa de substituição da seguinte maneira: seja s algum byte do estado a ser substituído; primeiramente se calcula t' que é o inverso multiplicativo de s no corpo $\mathbb{F}_{2^8}$, quando $s = 0$ então $t' = 0$. Depois, seja $(t'_7, \ldots, t'_0)_2$ a representação na base 2 de t'. Então é possível substituir s pelo byte $t = At' + b$, onde A e b estão definidas no padrão [Nat01a]. As operações de multiplicação e soma devem ser feitas no corpo binário $\mathbb{F}_2$, isto é, com a aritmética modulo 2. A caixa de substituição foi projetada para que possua uma baixa correlação entre os bits de entrada e os bits de saída [DR01].

- `ShiftRows`. Realiza uma rotação nas linhas da matriz, isso assegura que os bytes de uma coluna sejam dispersos nas quatro colunas.

$$\begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_5 & t_6 & t_7 & t_4 \\ t_{10} & t_{11} & t_8 & t_9 \\ t_{15} & t_{12} & t_{13} & t_{14} \end{bmatrix} = \texttt{ShiftRows}\left(\begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 & t_7 \\ t_8 & t_9 & t_{10} & t_{11} \\ t_{12} & t_{13} & t_{14} & t_{15} \end{bmatrix}\right) \tag{3}$$

- `MixColumns`. Calcula a multiplicação do estado, visto como matriz, com uma matriz predefinida. Diferente da operação `SubBytes`, aqui as operações aritméticas são realizadas no corpo binário $\mathbb{F}_{2^8}$.

$$\begin{bmatrix} u_0 & u_1 & u_2 & u_3 \\ u_4 & u_5 & u_6 & u_7 \\ u_8 & u_9 & u_{10} & u_{11} \\ u_{12} & u_{13} & u_{14} & u_{15} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} t_0 & t_1 & t_2 & t_3 \\ t_5 & t_6 & t_7 & t_4 \\ t_{10} & t_{11} & t_8 & t_9 \\ t_{15} & t_{12} & t_{13} & t_{14} \end{bmatrix} \tag{4}$$

- `AddRoundKey`. Aplica uma operação XOR entre o estado e a chave de rodada correspondente, como pode ser visto nas linhas 3, 8 e 12 do Algoritmo 1.

A chave é usada para gerar um conjunto de *chaves de rodada* as quais são computadas mediante uma função de expansão de chaves, descrita no Algoritmo 2. Neste algoritmo são introduzidas duas operações que operam sobre palavras de 32 bits:

- `RotWord`. Seja $(t_3, t_2, t_1, t_0)_8$ a representação na base 8 da palavra T, então a função de rotação está definida como segue:

$$(t_2, t_1, t_0, t_3)_8 \leftarrow \texttt{RotWord}(T). \tag{5}$$

- `SubWord`. Seja $(t_3, t_2, t_1, t_0)_8$ a representação na base 8 da palavra T, a operação consiste em aplicar a caixa de substituição descrita em `SubBytes` para cada byte:

$$(t'_3, t'_2, t'_1, t'_0)_8 \leftarrow \texttt{SubWord}(T),\ t'_i \leftarrow \texttt{SubBytes}(t_i) \text{ para } i \in \{0,1,2,3\}. \tag{6}$$

O conjunto de chaves de rodada pode ser gerado antes da execução do algoritmo de encriptação. Essas chaves podem ser reutilizadas para gerar as chaves de rodada do processo de decriptação.

Algoritmo 1 Encriptação de um bloco de 128 bits usando AES.

Entrada: M, uma cadeia binária de 128 bits.
k, uma chave de tamanho $l \in \{128, 192, 256\}$ bits.

Saída: C, uma cadeia binária de 128 bits que representa o texto encriptado de M.

1: $\{R_0, \ldots, R_N\} \leftarrow \texttt{KeyExpansion}(k)$

2: $N \leftarrow \begin{cases} 10 & \textbf{if } l = 128 \\ 12 & \textbf{if } l = 192 \\ 14 & \textbf{if } l = 256 \end{cases}$

3: $S \leftarrow M \oplus R_0$

4: **for** $i \leftarrow 1$ **to** $N-1$ **do**

5: $\quad T \leftarrow \texttt{SubBytes}(S)$

6: $\quad T' \leftarrow \texttt{ShiftRows}(T)$

7: $\quad U \leftarrow \texttt{MixColumns}(T')$

8: $\quad S \leftarrow U \oplus R_i$

9: **end for**

10: $S \leftarrow \texttt{SubBytes}(S)$

11: $S \leftarrow \texttt{ShiftRows}(S)$

12: $C \leftarrow S \oplus R_N$

13: **return** C

Algoritmo 2 Expansão de chaves do algoritmo AES (`KeyExpansion`).

Entrada: k, uma chave de tamanho $l \in \{128, 192, 256\}$ bits.

Saída: $\{R_0, \ldots, R_N\}$, o conjunto de chaves de rodada.

1: $N_k \leftarrow \frac{l}{32}$

2: $N \leftarrow \begin{cases} 10 & \textbf{if } l = 128 \\ 12 & \textbf{if } l = 192 \\ 14 & \textbf{if } l = 256 \end{cases}$

3: $\texttt{RCon}_i \leftarrow 2^{i-1} \in \mathbb{F}_{2^8}$ para $i \in \{1, \ldots, 10\}$.

4: Seja $(w_{N_k-1}, \ldots, w_0)_{32}$ a representação na base 32 da chave k.

5: **for** $i \leftarrow N_k$ **to** $4N+3$ **do**

6: $\quad T \leftarrow w_{i-1}$

7: $\quad$ **if** $i \equiv 0 \mod N_k$ **then**

8: $\quad\quad T \leftarrow \texttt{SubWord}(\texttt{RotWord}(T)) \oplus \texttt{RCon}_{i/N_k}$

9: $\quad$ **else if** $N_K > 6$ **and** $i \equiv 4 \mod N_k$ **then**

10: $\quad\quad T \leftarrow \texttt{SubWord}(T)$

11: $\quad$ **end if**

12: $\quad w_i \leftarrow w_{i-N_k} \oplus T$

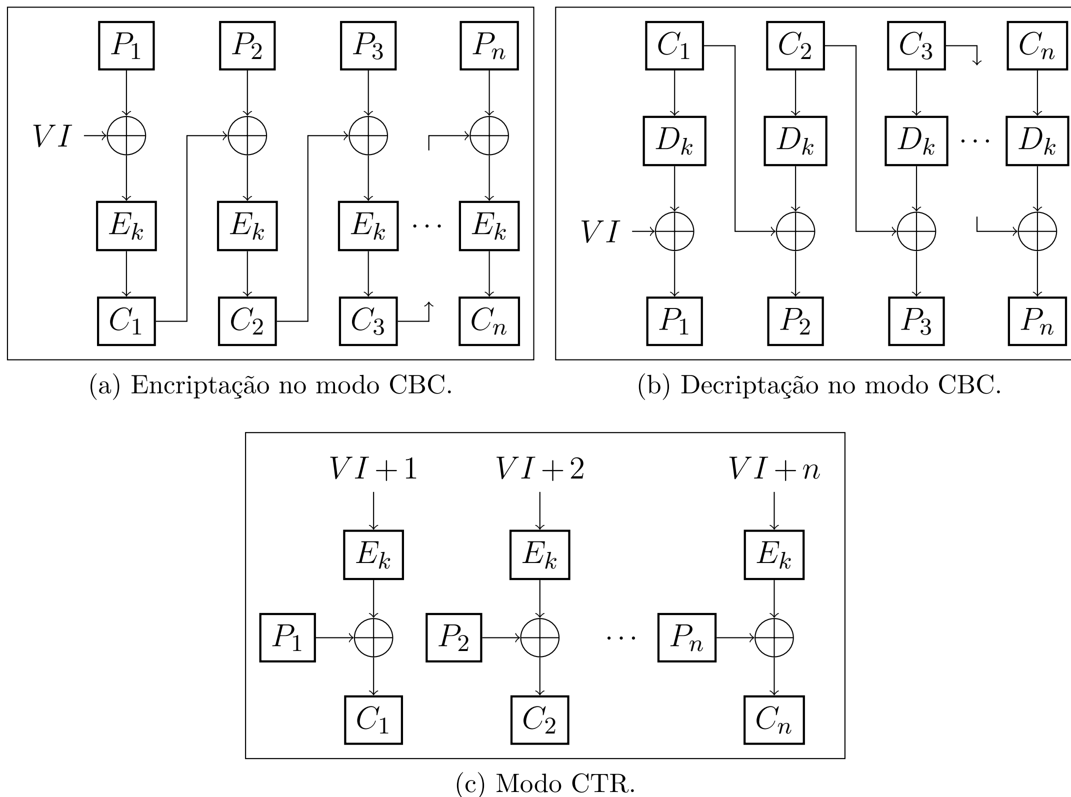
13: **end for**

14: **for** $i \leftarrow 0$ **to** N **do**

15: $\quad R_i \leftarrow (w_{4i} \parallel w_{4i+1} \parallel w_{4i+2} \parallel w_{4i+3})$

16: **end for**

17: **return** $\{R_0, \ldots, R_N\}$



(a) Encriptação no modo CBC.

(b) Decriptação no modo CBC.

(c) Modo CTR.

Figura 3.3: Modos de operação definidos pelo NIST no padrão SP 800-38A [Nat01b].

3.4.1.2. Modos de operação

O algoritmo AES recebe como entrada um bloco de 128 bits e uma chave para produzir um texto encriptado do mesmo tamanho. Se o texto claro for maior que 128 bits ainda é possível usar o AES, dividindo-o em blocos de 128 bits. No entanto, quando múltiplos blocos são encriptados usando a mesma chave alguns problemas de segurança podem surgir.

Para resolver esse problema, existem algoritmos chamados *modos de operação* que permitem que os cifradores de bloco sejam usados para mensagens de qualquer tamanho. Esses modos dividem a mensagem em n blocos, $P = \{P_1, \ldots, P_n\}$, e utilizam o cifrador de bloco como primitiva básica para obter o texto cifrado, $C = \{C_1, \ldots, C_n\}$. O NIST definiu no padrão SP 800-38A cinco modos de operação: ECB, CBC, CFB, OFB e CTR [Nat01b]. Aqui serão detalhados dois dos mais utilizados.

No modo de operação *Cipher Block Chaining* (CBC) a computação do texto cifrado é feita como segue: o primeiro bloco, C_1, é computado como pela encriptação de $P_1 \oplus VI$, onde VI é um vetor de inicialização; os blocos subsequentes são computados por $C_i = E_k(P_i \oplus C_{i-1})$, para $2 \leq i \leq n$. Como pode ser visto na Figura 3.3a, a geração do texto encriptado é inerentemente sequêncial. Entretanto, a decriptação pode ser computada mais rapidamente, pois depende apenas dos blocos de texto

encriptado, veja a Figura 3.3b.

O modo de operação *Counter* (CTR) é um modo mais flexível e eficiente do que o CBC. O CTR calcula o texto cifrado da seguinte maneira: cada bloco do texto cifrado C_i é calculado pela operação XOR entre P_i e o resultado da encriptação de $VI+i$, para $1 \leq i \leq n$. A encriptação de um texto claro pode ser feita em paralelo, sendo mais eficiente do que o modo CBC. Além disso, o processo de decriptação é análogo ao de encriptação e pode ser computado por $P_i = C_i \oplus E_k(VI+i)$, para $1 \leq i \leq n$. Na Figura 3.3c está ilustrado o modo de operação CTR.

O vetor de inicialização (VI) tem o efeito de aleatorizar o processo de encriptação, portanto deve ser escolhido aleatoriamente sempre que uma encriptação é realizada. Por conta disso, é recomendado que o mesmo vetor de inicialização não seja utilizado mais de uma vez, caso contrário o modo de operação se torna inseguro.

3.4.1.3. Implementações do algoritmo AES

A maioria das operações usadas pelo AES podem ser calculadas através de operações binárias que estão presentes na maioria dos processadores atuais. Entretanto, uma implementação eficiente e resistente a ataques de canal lateral do AES não é uma tarefa trivial. Um conjunto de técnicas que podem ser usadas para conseguir uma implementação com tais características é apresentado a seguir:

- **Tabelas de consulta.** O AES pode ser implementado eficientemente usando tabelas de consulta; nessa técnica uma parte do processamento do algoritmo pode ser substituída diretamente por um acesso a uma tabela previamente computada. Essa técnica, entretanto, requer que os acessos à tabela sejam feitos de forma segura, evitando assim os ataques de canal lateral.

- **Bitslice**. A técnica de implementação *bitslicing* foi proposta por Biham para o algoritmo DES [Bih97]. Posteriormente essa técnica começou a ser usada também no AES [RSD06, MN07, KS09], visando aumentar o desempenho, pois com o *bitslice* as transformações do AES podem ser descritas na forma de circuitos lógicos que operam bit a bit.

- **AES-NI**. Em 2010 a Intel agregou a seus processadores um novo conjunto de instruções, *Intel Advanced Encryption Standard New Instructions* (AES-NI), que implementa todas as etapas do algoritmo AES em *hardware*, resultando assim em uma implementação do AES mais eficiente que as implementações de *software* conhecidas até então. Além de permitir a implementação do AES com alto desempenho as novas instruções também fornecem benefícios de segurança, pois elas não utilizam tabelas de consulta e seu tempo de execução deve ser igual, independente dos dados de entrada, eliminando assim os principais ataques de tempo conhecidos [Gue10].

Na seção seguinte serão apresentados mais detalhes de como as novas instruções são aplicadas na implementação segura e eficiente do algoritmo AES.

3.4.1.4. Aceleração usando instruções AES-NI

Em 2010, a Intel lançou uma arquitetura chamada *Westmere*, que adicionou seis novas instruções para acelerar o processo de encriptação usando o algoritmo AES. Esse novo conjunto de instruções é conhecido como AES-NI e tais instruções podem ser usadas diretamente em linguagem de montagem ou também como funções em linguagem C (comumente conhecidas como *intrinsics* [Cor14]).

As instruções AES-NI usam registradores de 128 bits para armazenar a informação da chave e o estado. A seguir são apresentadas as instruções presentes no conjunto AES-NI:

- `AESENC`. Realiza uma rodada da função de encriptação do AES. Aplica as operações de `ShiftRows`, `SubBytes` e `MixColumns` sobre um registrador `XMM1`, que contém o estado; posteriormente é aplicado uma operação XOR entre o resultado e a chave de rodada, guardada no registrador `XMM2`.
- `AESENCLAST`. Computa a última rodada de função de encriptação do AES. Aplica as operações de `ShiftRows` e `SubBytes` sobre um registrador `XMM1`, que contém o estado; posteriormente é aplicado uma operação XOR entre o resultado e a chave de rodada, guardada no registrador `XMM2`.
- `AESDEC/AESDECLAST`. Essas instruções são análogas às instruções previamente explicadas, com a diferença que estas são usadas para computar a decriptação.
- `AESKEYGENASSIST`. Essa instrução é usada para gerar as chaves de rodada do AES; na Figura 3.4d pode ser visto o código que é usado para gerar as chaves de rodada do algoritmo AES-128.
- `AESIMC`. Essa função gera as chaves de rodada da decriptação a partir das chaves de rodada da encriptação, usando o trecho de código na Figura 3.4c.

Os trechos de código abaixo mostram como implementar em linguagem de montagem o algoritmo AES-128 para encriptar um bloco de 128 bits. Inicialmente, as chaves de rodada previamente computadas se encontram armazenadas nos registradores `XMM0` a `XMM10`, e o estado S é armazenado no registrador `XMM15`. A primeira instrução faz um XOR com a primeira chave de rodada e depois utiliza nove vezes a instrução `AESENC` para computar uma rodada. Finalmente, a computação da última rodada é executada pela instrução `AESENCLAST`. Nas Figuras 3.4a e 3.4b são mostradas as sequencias de instruções para realizar a encriptação e a decriptação de blocos de 128 bits, respectivamente.

3.4.1.5. Testes de desempenho

Alguns experimentos foram projetados para nos auxiliar a observar os ganhos obtidos com a utilização do conjunto AES-NI.

```
1 pxor    xmm15, xmm0       ; AddRoundKey
2 aesenc xmm15, xmm1        ; Round 1
3 aesenc xmm15, xmm2        ; Round 2
4 aesenc xmm15, xmm3        ; Round 3
5 aesenc xmm15, xmm4        ; Round 4
6 aesenc xmm15, xmm5        ; Round 5
7 aesenc xmm15, xmm6        ; Round 6
8 aesenc xmm15, xmm7        ; Round 7
9 aesenc xmm15, xmm8        ; Round 8
10 aesenc xmm15, xmm9       ; Round 9
11 aesenclast xmm15, xmm10 ; Round 10
```

(a) Encriptação.

```
1 pxor   xmm15, xmm10      ; AddRoundKey
2 aesdec xmm15, xmm9       ; Round 1
3 aesdec xmm15, xmm8       ; Round 2
4 aesdec xmm15, xmm7       ; Round 3
5 aesdec xmm15, xmm6       ; Round 4
6 aesdec xmm15, xmm5       ; Round 5
7 aesdec xmm15, xmm4       ; Round 6
8 aesdec xmm15, xmm3       ; Round 7
9 aesdec xmm15, xmm2       ; Round 8
10 aesdec xmm15, xmm1      ; Round 9
11 aesdeclast xmm15, xmm0 ; Round 10
```

(b) Decriptação.

```
1 mov rdx, OFFSET Key_Schedule
2 mov rax, OFFSET Key_Schedule_Decrypt
3 movdqu xmm1, XMMWORD PTR [rdx]
4 movdqu XMMWORD PTR [rax], xmm1
5 add rdx, 0x10
6 add rax, 0x10
7 mov ecx, 9   ;{9,11,13}
8
9 repeat:
10   movdqu xmm1, XMMWORD PTR [rdx]
11   aesimc xmm1, xmm1
12   movdqu XMMWORD PTR [rax], xmm1
13   add rdx, 0x10
14   add rax, 0x10
15   sub rcx, 0x01
16   jnz repeat
17
18 movdqu xmm1, XMMWORD PTR [rdx]
19 movdqu XMMWORD PTR [rax], xmm1
```

(c) Geração das chaves de rodada para decriptação.

```
1 keygen_128:
2   movdqu xmm1, XMMWORD PTR Keyindent
3   movdqu XMMWORD PTR KeySch, xmm1
4   mov rcx, OFFSET KeySch+16
5   aeskeygenassist xmm2, xmm1, 0x01
6   call key_expansion_128
7   aeskeygenassist xmm2, xmm1, 0x02
8   call key_expansion_128
9   aeskeygenassist xmm2, xmm1, 0x04
10  call key_expansion_128
11  aeskeygenassist xmm2, xmm1, 0x08
12  call key_expansion_128
13  aeskeygenassist xmm2, xmm1, 0x10
14  call key_expansion_128
15  aeskeygenassist xmm2, xmm1, 0x20
16  call key_expansion_128
17  aeskeygenassist xmm2, xmm1, 0x40
18  call key_expansion_128
19  aeskeygenassist xmm2, xmm1, 0x80
20  call key_expansion_128
21  aeskeygenassist xmm2, xmm1, 0x1b
22  call key_expansion_128
23  aeskeygenassist xmm2, xmm1, 0x36
24  call key_expansion_128
25  jmp END
26
27 key_expansion_128:
28  pshufd xmm2, xmm2, 0xFF
29  pslldq xmm3, xmm1, 0x04
30  pxor   xmm1, xmm3
31  pslldq xmm3, xmm1, 0x04
32  pxor   xmm1, xmm3
33  pslldq xmm3, xmm1, 0x04
34  pxor   xmm1, xmm3
35  pxor   xmm1, xmm2
36  movdqu XMMWORD PTR [rcx], xmm1
37  add rcx, 0x10
38  ret
39 END:
```

(d) Geração das chaves de rodada para encriptação.

Figura 3.4: Lista de códigos de montagem usados na implementação das funções de encriptação, decriptação e geração das chaves de rodada do algoritmo AES-128 usando as instruções AES-NI.

Teste 1. *Calcular os ciclos por byte e a vazão para encriptar mensagens de 50 MB usando o algoritmo AES com tamanho de chave de 128 bits e com os modos de operação CBC e CTR.*

O desempenho de dois códigos que implementam o algoritmo AES-128 serão mostrados, usando os modos de operação CBC e CTR. A primeira implementação

Implementação	Modo CBC		Modo CTR	
	Ciclos/byte	MB/s	Ciclos/byte	MB/s
Nativa de x64	11,08	306,08	11,67	290,51
AES-NI	4,50	752,36	0,83	4061,86

Tabela 3.2: Comparação do desempenho do AES-128-CBC e AES-128-CTR. Os resultados deste teste foram obtidos no computador HW-ULTRA especificado no Apêndice A.

não aproveita o suporte de encriptação por *hardware* e simplesmente utiliza instruções nativas de 64 bits. A segunda implementação, entretanto, utiliza as instruções AES-NI presentes nos processadores da Intel. Esses códigos fazem parte da biblioteca de exemplos da Intel (versão 1.2) que está disponível em [Rot11].

Os resultados do Teste 1 são apresentados na Tabela 3.2; pode-se perceber que no modo de operação CBC, a implementação que usa AES-NI é aproximadamente 2,5 vezes mais rápida que a implementação nativa de 64 bits. O impacto no desempenho causado pelas instruções AES-NI é ainda maior no CTR, que chega a ser 14 vezes mais rápido do que uma implementação sequencial nativa de 64 bits.

Teste 2. *Utilizar o modulo de medição de desempenho da biblioteca OpenSSL para determinar o tempo de execução do algoritmo AES-128-CBC. O teste será feito, com e sem suporte das instruções AES-NI.*

Neste experimento foi utilizado o OpenSSL, versão 1.0.2d [The03]. O desempenho do algoritmo AES pode ser medido pela seguinte linha de comando:

```
1 $ openssl speed -elapsed -evp aes-128-cbc | tail
```

Na hora de indicar o parâmetro `-evp` o OpenSSL escolherá a implementação adequada para a arquitetura alvo; caso seu computador não conte com o conjunto de instruções AES-NI, então será executado o teste com uma implementação suportada. Parte da saída deste comando, executada no processador HW-ULTRA, é apresentada a seguir:

```
1 The 'numbers' are in 1000s of bytes per second processed.
2 type              16 bytes     64 bytes     256 bytes
3 aes-128-cbc      285865.74k   306641.68k   311269.29k
4 type            1024 bytes   8192 bytes
5 aes-128-cbc      313865.59k   314833.08k
```

Agora, o mesmo experimento será executado indicando explicitamente, através de uma variável de ambiente, que execute uma implementação genérica, isto é, sem suporte ao AES-NI.

```
1 $ OPENSSL_ia32cap="~0x200000200000000" \
2   openssl speed -elapsed -evp aes-128-cbc | tail
```

```
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes
aes-128-cbc     129931.97k   144134.36k   146498.66k
type            1024 bytes   8192 bytes
aes-128-cbc     148852.61k   149275.23k
```

Os resultados dos experimentos, mostram que a implementação usando AES-NI é acima de 2 vezes mais rápida do que a implementação genérica. Maiores detalhes sobre o desempenho do OpenSSL usando instruções AES-NI podem ser consultados em [GGF+15].

Observações: esses dois experimentos são uma prova de que é possível atingir um nível de desempenho alto para encriptar dados desde que a implementação aproveite de maneira adequada as instruções especializadas contidas nos processadores.

3.4.2. Implementação da função de resumo SHA-3

Esta seção detalha como usar o conjunto de instruções vetoriais para acelerar a computação do mais novo padrão de funções de resumo SHA-3. Inicialmente será apresentada uma breve introdução da família de funções de resumo SHA-3 e, posteriormente, como a implementação em software do SHA-3 pode ser acelerada por meio de instruções vetoriais.

A família de funções de resumo *Standard Hash Algorithm* (SHA) [Nat08a] foi padronizada pelo Instituto Nacional de Padrões e Tecnologias americano (NIST) e atualmente é usada em muitas aplicações e protocolos. Nos últimos anos, a família SHA sofreu alguns ataques; em 2005, [BCJ+05] e [RO05] mostraram ataques à colisões em uma versão reduzida do SHA-1; no mesmo ano, [WYY05] mostrou um ataque que quebra, teoricamente, a resistência à colisão do SHA-1. A segunda versão da família, SHA-2, possui uma estrutura muito similar ao SHA-1 e já possui alguns ataques à sua versão reduzida, como é mostrado em [IMPR09].

Adicionalmente, foram feitas críticas aos algoritmos SHA-1 e SHA-2 por seus detalhes de projeto serem fechados ao público. Por este motivo, após um período de consulta pública e dois *workshops*, o NIST decidiu publicar, em 2007, uma chamada aberta para a seleção de um novo algoritmo, o SHA-3. Ao contrário dos algoritmos anteriores, o concurso para a escolha do SHA-3 foi feito nos mesmos moldes do AES, algoritmo padrão de cifra de bloco, selecionado em 2001 após cinco anos de concurso [Nat01a]. Por esse motivo, todos os algoritmos que foram submetidos tiveram que ter suas patentes abertas e o código disponível, afim de serem analisados e testados durante o concurso. Em 2012, após três rodadas de competição, o algoritmo Keccak [BDPA11] foi anunciado como vencedor [jCPB+12], este algoritmo baseia-se na construção esponja definida sobre uma permutação Keccak-p.

Em Agosto de 2015 foi publicado o documento oficial FIPS 202 [Nat15], que descreve os detalhes da família SHA-3 e padroniza quatro funções de resumo, SHA3-224, SHA3-256, SHA3-384 e SHA3-512 e duas funções de resumo com saída variável, *Extendable Output Functions* (XOF), chamadas SHAKE128 e SHAKE256.

Funções	Tamanho de saída	Nível de segurança em bits		
		Colisão	Pré-imagem	2ª Pré-imagem
SHA-1	160	<80	160	$160 - L(M)$
SHA-224	224	112	224	$\min(224, 256 - L(M))$
SHA-512/224	224	112	224	224
SHA-256	256	128	256	$256 - L(M)$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	$512 - L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	$\min(d/2, 128)$	$\geq \min(d, 128)$	$\min(d, 128)$
SHAKE256	d	$\min(d/2, 256)$	$\geq \min(d, 256)$	$\min(d, 256)$

Tabela 3.3: Nível de segurança das funções SHA-1, SHA-2 e SHA-3 [Nat15].

Os parâmetros de todas as funções da família SHA podem ser encontrados na Tabela 3.3. Nas seções seguintes serão apresentadas a família SHA-3 baseada no algoritmo Keccak e como implementar esse algoritmo eficientemente em arquiteturas de 64 bits.

3.4.2.1. Permutações Keccak-p

Uma permutação Keccak-p que possui n_r rodadas e trabalha sobre um estado S de b bits é denotada por Keccak-$p[b, n_r]$; a permutação é definida para quaisquer $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ e qualquer inteiro positivo n_r. Uma rodada da permutação Keccak-p consiste de uma sequência de cinco transformações, chamadas de *etapas de mapeamento*.

O estado é organizado em uma matriz $5 \times 5 \times w$, onde $w = b/25$ pode ser visto como o tamanho das palavras do estado em bits. As 25 palavras de w bits de um estado S são denotadas mediante s_i para $0 \leq i < 25$ como pode ser visto a seguir:

$$S = \begin{bmatrix} s_0 & s_1 & s_2 & s_3 & s_4 \\ s_5 & s_6 & s_7 & s_8 & s_9 \\ s_{10} & s_{11} & s_{12} & s_{13} & s_{14} \\ s_{15} & s_{16} & s_{17} & s_{18} & s_{19} \\ s_{20} & s_{21} & s_{22} & s_{23} & s_{24} \end{bmatrix}; \; S[x, y] = s_{5x+y} \text{ para } 0 \leq x, y < 5. \quad (7)$$

Etapas de mapeamento. As cinco etapas de mapeamento usadas durante uma rodada da função Keccak-$p[b, n_r]$ são denotadas por θ, ρ, π, χ e ι. O algoritmo para cada uma dessas etapas tem como entrada um estado S e como retorno esse estado

atualizado. A etapa de mapeamento ι é a única que possui uma entrada adicional, um inteiro i_r chamado índice da rodada.

O tamanho do estado é um parâmetro que está omitido da notação, pois b sempre é especificado quando as etapas de mapeamento são chamadas; visando simplificar a notação, todas as operações lógicas e de rotação usadas nas etapas de mapeamento são aplicadas sobre palavras de tamanho w.

As especificações das etapas de mapeamento são apresentadas a seguir:

- **Etapa de mapeamento** θ. O efeito da etapa θ é aplicar uma operação XOR entre cada palavra do estado com a paridade de duas colunas do estado.
- **Etapa de mapeamento** ρ. Nesta etapa cada palavra do estado é rotada uma quantidade fixa de r_i bits. A seguir pode-se ver a matriz R, onde cada elemento r_i representa a quantidade de bits que a palavra s_i será rotada.

$$R = \begin{bmatrix} 0 & 1 & 62 & 28 & 27 \\ 36 & 44 & 6 & 55 & 20 \\ 3 & 10 & 43 & 25 & 39 \\ 41 & 45 & 15 & 21 & 8 \\ 18 & 2 & 61 & 56 & 14 \end{bmatrix} ;\ R[x,y] = r_{5x+y} \text{ para } 0 \leq x,y < 5. \quad (8)$$

- **Etapa de mapeamento** π. O efeito da etapa de mapeamento π é embaralhar as palavras do estado. Ela promove uma difusão a longo prazo dentro das rodadas, a fim de evitar que padrões sejam explorados em determinados ataques.
- **Etapa de mapeamento** χ. O efeito da etapa de mapeamento χ é aplicar uma operação XOR em cada palavra do estado s_i com a saída de uma função não linear aplicada a duas palavras da mesma linha de s_i.
- **Etapa de mapeamento** ι. A etapa de mapeamento ι consiste na aplicação de uma operação XOR entre o elemento s_0 com um valor contante $rc(i_r)$, onde i_r é o índice da rodada; os valores de rc são definidos em [Nat15] e são gerados a partir de um *Linear Feedback Shift Register* (LFSR).

Dado um estado S e o índice de uma rodada i_r, uma função Rnd é a transformação que resulta na aplicação das etapas de mapeamento θ, ρ, π, χ e ι, como pode ser visto a seguir:

$$\text{Rnd}(S, i_r) = \iota(\chi(\pi(\rho(\theta(S)))), i_r). \quad (9)$$

A permutação Keccak-$p[b, n_r]$ consiste de n_r iterações de Rnd.

3.4.2.2. Construção esponja

A construção esponja [BDPA07] define uma função `ESPONJA`$[f,$ `pad`$, r]$ com domínio $\{0,1\}^*$ e codominio $\{0,1\}^\infty$ usando uma permutação de tamanho fixo f, uma regra

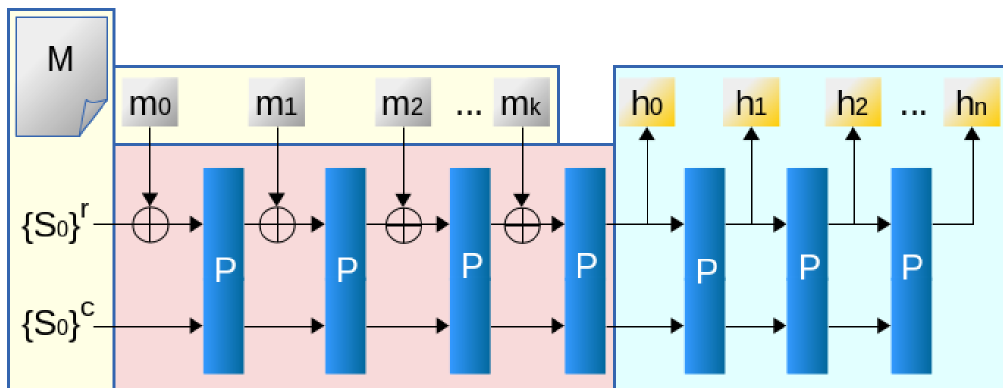


Figura 3.5: Construção esponja: $Z = \texttt{ESPONJA}[f,\, \texttt{pad},\, r](M, d)$.

de preenchimento pad e uma *taxa de bits* r. A partir desta função, uma saída de tamanho finito pode ser obtida pelo truncamento dos primeiros l bits. Uma instância da construção esponja é chamada de *função esponja*.

A função esponja opera sobre um estado de $b = r + c$ bits; onde r é a taxa de bits e c é a *capacidade*. Primeiramente, todos os bits do estado são inicializados com zeros. A regra de preenchimento pad é aplicada sobre a mensagem de entrada e posteriormente a mensagem é dividida em k blocos de r bits; uma vez feito isso o processamento é composto por duas fases: a fase de absorção e a fase de extração.

Na *fase de absorção* é aplicado uma operação XOR entre um bloco de r bits da mensagem de entrada com os primeiros r bits do estado atual. Então, o estado é atualizado por meio de uma permutação f; esse processo é aplicado para cada um dos k blocos da mensagem.

Na *fase de extração* os primeiros r bits do estado são usados como saída; se o tamanho de l for maior que o tamanho de r, o estado é processado novamente pela permutação f e os r bits retornados são concatenados com os r bits retornados previamente; esse processo é repetido até que os l bits requeridos sejam extraídos.

Os últimos c bits do estado nunca são afetados diretamente pelos blocos de entrada na fase de absorção e nem extraídos na fase de extração; o parâmetro c determina a segurança atingível pela construção [BDPA08].

A construção esponja é ilustrada na Figura 3.5 e seu pseudo-código é apresentado no Algoritmo 3.

3.4.2.3. O algoritmo Keccak

Keccak é uma família de funções de resumo, originalmente definida em [BDPA11]. A seguir, a regra de preenchimento usada pela família Keccak será descrita com os parâmetros e a permutação fundamental usada pelo Keccak:

- **Regra de preenchimento pad10*1**. Recebe como entrada o tamanho da mensagem m e a taxa de bits r, retornando uma cadeia de dados $Z = 1||0^j||1$,

Algoritmo 3 $\texttt{ESPONJA}[f, \texttt{pad}, r](M, d)$

Entrada: Cadeia de dados M e um inteiro não negativo d.
Saída: Cadeia de dados Z, tal que $|Z| = d$.

1: $P = M \parallel \texttt{pad}(r, |M|)$
2: $n = |P|/r$
3: $c = b - r$
4: Seja $P = P_0 \parallel \ldots \parallel P_{n-1}$ a divisão de P em blocos de tamanho r.
5: $S = 0^b$
6: **for all** i tal que $0 \le i < n$ **do**
7: $\quad S = f(S \oplus (P_i \parallel 0^c))$
8: **end for**
9: Seja Z uma cadeia de dados vazia.
10: **while** $d > |Z|$ **do**
11: $\quad Z = Z \parallel \texttt{Trunc}_r(S)$
12: $\quad$ Se $d > |Z|$ então $S = f(S)$
13: **end while**
14: **return** $\texttt{Trunc}_d(Z)$

onde $j = -(m+2) \bmod r$.

- **Especificação de Keccak**$[c]$. A família de funções esponjas Keccak com a função de permutação Keccak-$p[b, n_r]$ e a regra de preenchimento pad10*1 está definida para quais quer par de taxa de bits r e capacidade c tal que $r + c \in \{25, 50, 100, 200, 400, 800, 1600\}$.

Quando o tamanho do estado é restrito para $b = 1600$, a família Keccak é denotada por Keccak[c]; nesse caso, r é determinado pela escolha do parâmetro c. Em particular,

$$\text{Keccak}[c] = \texttt{ESPONJA}[\text{Keccak-}p[1600, 24], \text{pad}10^*1, 1600 - c]. \tag{10}$$

Assim, dado uma mensagem M e um tamanho de saída d,

$$\text{Keccak}[c](M, d) = \texttt{ESPONJA}[\text{Keccak-}p[1600, 24], \text{pad}10^*1, 1600 - c](M, d). \tag{11}$$

3.4.2.4. Especificações da função SHA-3

Nesta seção, serão descritas as quatro funções de resumo SHA-3 e as duas funções com saída variável SHAKE128 e SHAKE256.

As quatro funções de resumo SHA-3 são definidas a partir da aplicação da função Keccak$[c]$ sobre uma mensagem M concatenada com dois bits e a especificação do tamanho de saída, como pode ser visto a seguir:

$$\begin{aligned}
\text{SHA3-224(M)} &= \text{Keccak}[448](\text{M}||01, 224) \\
\text{SHA3-256(M)} &= \text{Keccak}[512](\text{M}||01, 256) \\
\text{SHA3-384(M)} &= \text{Keccak}[768](\text{M}||01, 384) \\
\text{SHA3-512(M)} &= \text{Keccak}[1024](\text{M}||01, 512)
\end{aligned}$$

Em cada caso, a capacidade sempre é o dobro do tamanho da saída; os dois bits adicionados à mensagem (01) servem para dar suporte à separação de domínio, por exemplo, distinguir as mensagens processadas pela função de resumo SHA-3 e pelas funções com saída variável.

As duas funções com saída variável, SHAKE128 e SHAKE256 são definidas a partir da função Keccak[c], uma mensagem M concatenada com quatro bits e pela especificação do tamanho de saída d, como pode ser visto a seguir:

$$\text{SHAKE128(M)} = \text{Keccak[256](M}||\text{1111,d)}$$
$$\text{SHAKE256(M)} = \text{Keccak[512](M}||\text{1111,d)}$$

Os primeiros dois bits servem para dar suporte a separação de domínio e os dois bits adicionais (11) são usados para prover compatibilidade ao esquema Sakura [BDPV14]. Esse esquema facilitará o desenvolvimento de uma extensão dessas funções, chamada resumos em árvore [Mer88], onde a mensagem pode ser processada em paralelo, conseguindo assim computar mensagens longas de forma mais eficiente.

3.4.2.5. Implementação nativa de 64 bits do SHA-3

Dado que a permutação Keccak-p[1600,24] trabalha com palavras de 64 bits, a implementação para uma arquitetura de 64 bits pode ser feita de forma direta. O primeiro passo para a implementação é definir como o estado do algoritmo será representado.

Organização do estado. As palavras do estado são mapeadas diretamente à variáveis de 64 bits; desse modo, para comportar as 25 palavras do estado $S = \{s_0, \ldots, s_{24}\}$, é preciso usar 25 variáveis de 64 bits, denominadas $M = \{M_0, \ldots, M_{24}\}$, onde $\{M_0 = s_0, M_1 = s_1, \ldots, M_{24} = s_{24}\}$.

Uma vez mapeado o estado a variáveis de 64 bits, pode-se começar a computação da permutação Keccak-p[1600,24]. Uma implementação direta de uma rodada, denotada Rnd(M, i_r), pode ser vista no Algoritmo 4.

A maioria das arquiteturas de 64 bits atuais possuem menos que 25 registradores de propósito geral e sabe-se que operações em registradores são substancialmente mais rápidas que operações que buscam dados da memória. Por conta disso, uma implementação eficiente tem que aproveitar os dados dentro dos registradores na maior parte do tempo.

Visando um melhor aproveitamento dos registradores, pode-se implementar as etapas de mapeamento ρ, π, χ e ι de forma modular, processando as etapas ρ e π para um bloco de cinco palavras que serão usadas conjuntamente na etapa χ. O trecho de código a seguir é necessário para computar as novas palavras s_5, s_6, s_7, s_8 e s_9 após a aplicação das etapas de mapeamento ρ, π e χ:

Algoritmo 4 Rnd(M, i_r)

Entrada: Estado M e i_r o índice da rodada.
Saída: Estado M atualizado.

Etapa de mapeamento θ
1: $C_y = M_{0+y} \oplus M_{5+y} \oplus M_{10+y} \oplus M_{15+y} \oplus M_{20+y}$ para $0 \leq y < 5$
2: $D_x = C_{(x-1) \texttt{ mod } 5} \oplus (C_{(x+1) \texttt{ mod } 5} \lll 1)$ para $0 \leq x < 5$
3: $M_{5x+y} = M_{5x+y} \oplus D_x$ para $0 \leq x, y < 5$

Etapas de mapeamento ρ e π
4: $B_{(16x+10y) \texttt{ mod } 25} = (M_{5x+y} \lll r_{5x+y})$ para $0 \leq x, y < 5$

Etapa de mapeamento χ
5: **for all** (x, y) tal que $0 \leq x, y < 5$ **do**
6: $T = (B_{(5x+y+2) \texttt{ mod } 5}) \wedge (\neg B_{(5x+y+1) \texttt{ mod } 5})$
7: $M_{5x+y} = B_{5x+y} \oplus T$
8: **end for**

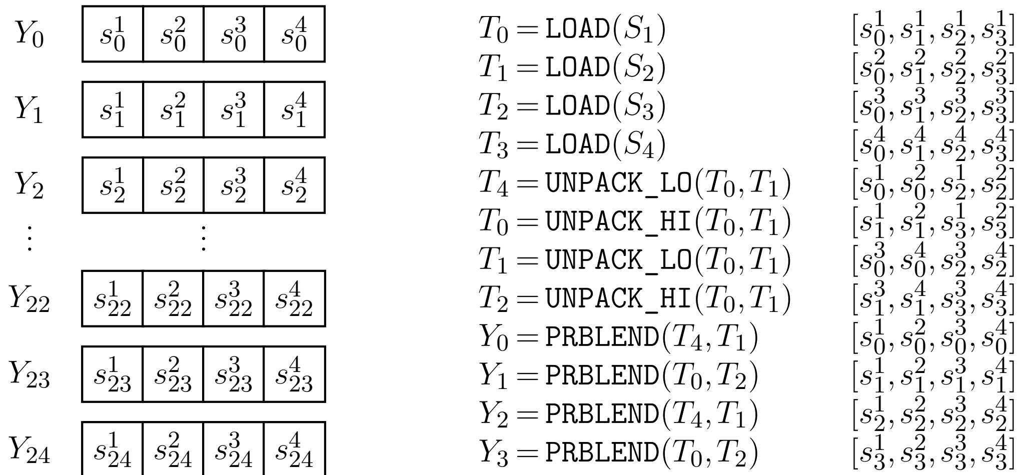
Etapa de mapeamento ι
9: $M_0 = M_0 \oplus rc(i_r)$
10: **return** M

$$
\begin{aligned}
T_0 &= M_3 \lll 28 \\
T_1 &= M_9 \lll 20 \\
T_2 &= M_{10} \lll 3 \\
T_3 &= M_{16} \lll 45 \\
T_4 &= M_{22} \lll 61 \\
M_{16} &= T_0 \oplus (T_2 \oplus (\neg T_1)) & M_{16} &= \chi(\pi(\rho(s_5))) \\
M_3 &= T_1 \oplus (T_3 \oplus (\neg T_2)) & M_3 &= \chi(\pi(\rho(s_6))) \\
M_{10} &= T_2 \oplus (T_4 \oplus (\neg T_3)) & M_{10} &= \chi(\pi(\rho(s_7))) \\
M_{22} &= T_3 \oplus (T_0 \oplus (\neg T_4)) & M_{22} &= \chi(\pi(\rho(s_8))) \\
M_9 &= T_4 \oplus (T_1 \oplus (\neg T_0)) & M_9 &= \chi(\pi(\rho(s_9)))
\end{aligned}
$$

Após a execução das etapas ρ, π e χ as palavras computadas não ficaram armazenadas nos registradores M_5, M_6, M_7 e M_8, isso acontece porque as palavras armazenadas nesses registradores ainda não foram processadas pelas etapas ρ, π e χ. A computação da etapa ι depende apenas do registrador M_0, desse modo pode-se computá-la a qualquer momento após calcular o novo valor de X_0 após a execução da etapa χ.

Pode-se perceber que, após a execução dessas etapas na iteração i, o estado tem uma organização diferente da inicial. Por conta disso, a implementação das etapas de mapeamento para a iteração seguinte $(i+1)$ precisa levar em conta a nova configuração do estado. O estado volta à sua configuração original após aplicação das etapas de mapeamento ρ, π e χ na iteração $i+1$.

Com essas otimizações, pode-se dispensar o estado temporário usado no Algoritmo 4 e também diminuir o acesso a memória, visto que os dados podem ser encontrados em registradores na maior parte do tempo.



(a) Organização do estado para a implementação vetorial usando registradores de 256 bits.

(b) Sequencia de instruções para ordenar as palavras dos registradores Y_0, Y_1, Y_2 e Y_3.

Figura 3.6: Organização do estado e sequencia de instruções para organizar as primeiras quatro palavras do estado.

3.4.2.6. Implementação vetorial usando registradores de 256 bits

Uma forma direta de usar as instruções vetoriais para implementar a função de permutação Keccak-p[1600,24] é processar mais de um estado por vez. Tirando proveito dos registradores de 256 bits pode-se processar quatro estados concorrentemente. A seguir são apresentados os detalhes internos dessa implementação; primeiramente, como criar um novo estado, com registradores de 256 bits, que contenha os quatro estados a serem processados e posteriormente como aplicar as etapas de mapeamento sobre esse novo estado.

Organização do estado. As instruções de 256 bits processam quatro palavras de 64 bits usando apenas uma instrução. O primeiro passo para a implementação foi mapear os quatro estados a serem processados em registradores de 256 bits; para comportar as 100 palavras dos estados $S^1 = \{s_0^1, \ldots, s_{24}^2\}$, $S^2 = \{s_0^2, \ldots, s_{24}^2\}$, $S^3 = \{s_0^3, \ldots, s_{24}^3\}$ e $S^4 = \{s_0^4, \ldots, s_{24}^4\}$ foi preciso usar 25 registradores de 256 bits, denominados $Y = \{Y_0, \ldots, Y_{24}\}$. O mapeamento de cada uma das palavras do estado em seus respectivos registradores pode ser visto na Figura 3.6a.

O mapeamento dos estados S^1, S^2, S^3 e S^4 em Y pode ser feito eficientemente com o uso das instruções `LOAD`, `UNPACK_LO`, `UNPACK_HI` e `PERM`; na Figura 3.6b é apresentado o trecho de código que mapeia as primeiras quatro palavras dos estados nos registradores Y_0, Y_1, Y_2 e Y_3; o processamento para as outras palavras é análogo ao apresentado.

Uma vez mapeado os estados a registradores de 256 bits, pode-se começar a computação da função de permutação Keccak-p[1600,24]. A implementação dessa versão que usa registradores de 256 bits para processar quatro estados independentes concorrentemente é muito similar ao Algoritmo 4; a diferença é que aqui todas as

operações são sobre registradores de 256 bits enquanto no Algoritmo 4 as operações são sobre palavras de 64 bits.

Também pode-se aplicar diretamente todas as otimizações usadas na implementação de 64 bits. Esta implementação usa praticamente a mesma quantidade de instruções que a implementação 64 bits.

3.4.2.7. Testes de desempenho

Para analisar o desempenho das implementações do algoritmo SHA-3 aqui apresentadas projetamos alguns experimentos que nos auxiliarão observar os ganhos obtidos com a utilização de instruções vetoriais.

Teste 3. *Calcular os ciclos por byte para produzir o valor de resumo de uma mensagem de 50 MB usando a implementação nativa de 64 bits e para produzir os valores de resumo de quatro mensagens de 50 MB concorrentemente usando a implementação que aproveita as instruções vetoriais.*

A implementação nativa de 64 bits, foi desenvolvida por Ronny Van Keeris, e é a implementação mais rápida disponível no *ECRYPT Benchmarking of Cryptographic Systems* (eBACS)[BL15a]. Essa implementação foi otimizada para a arquitetura de 64 bits usando as técnicas apresentadas nesta seção. A implementação vetorizada é uma implementação que usa registradores de 256 bits para calcular quatro valores de resumo concorrentemente e faz parte das implementações vetoriais do SHA-3 que foram apresentadas em [CL14].

Na Figura 3.7a é apresentado os ciclos por bytes necessários para computar o valor de resumo de mensagens de 50 MB para cada uma das funções da família SHA-3. Pode-se perceber o impacto das instruções vetoriais na aceleração desse algoritmo, uma vez que calcular o valor de resumo de quatro mensagens de mesmo tamanho usando as instruções vetoriais é em torno de 2,5 vezes mais rápido do que fazer a mesma computação usando a implementação de 64 bits.

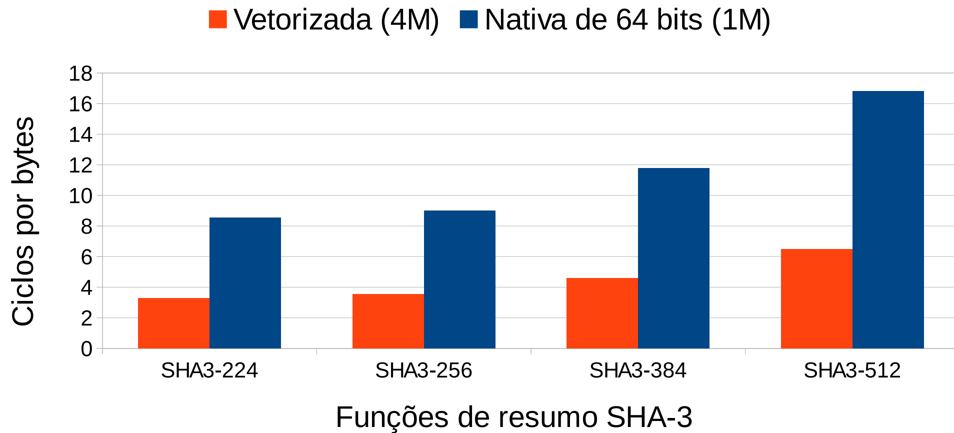
Teste 4. *Calcular os ciclos por byte para produzir o valor de resumo de mensagens com tamanho variando de 256 B até 512 MB.*

Na Figura 3.7b é apresentado os ciclos por byte necessários para calcular o valor de resumo de mensagens de tamanho 256 B a 512 MB usando a função SHA3-256.

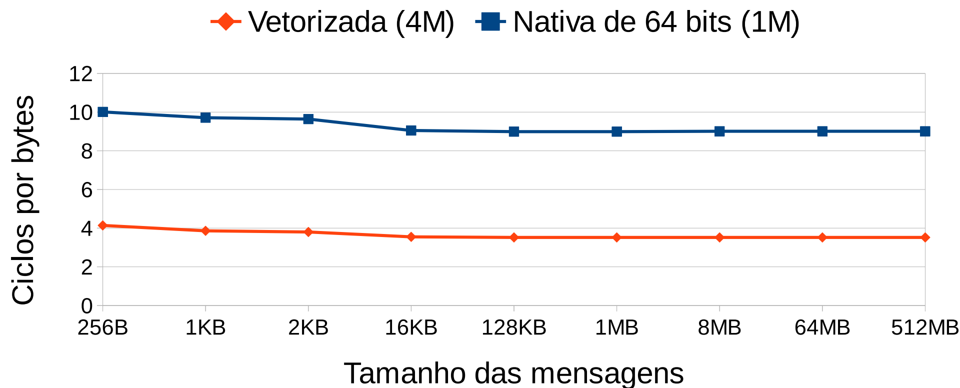
Observações: devido à importância do novo padrão de funções de resumo, é interessante dispor de técnicas de implementação que ajudem na aceleração da execução desses algoritmos, como as que foram apresentadas nesta seção.

3.4.3. Implementação do protocolo ECDH usando AVX2

A implementação do protocolo de acordo de chaves baseado em curvas elípticas envolve diversos aspectos a serem levados em conta: do ponto de vista de desempenho, como usar as instruções vetoriais na aceleração da execução do protocolo; do ponto



(a) Ciclos por byte para calcular o valor de resumo para as funções da família SHA-3.



(b) Ciclos por byte para calcular o valor de resumo de mensagens de tamanho 256 B a 512 MB.

Figura 3.7: Resultado dos testes de desempenho das implementações do algoritmo SHA-3.

de vista de segurança, quais contramedidas são usadas para evitar vazamento de informação sigilosa.

Atualmente o protocolo ECDH é implementado usando as diretrizes definidas pelo padrão SP 800-56A do NIST [BJS07] e utilizando as curvas elípticas recomendadas no padrão de assinaturas digitais definido em [KSD13]. Embora esse padrão ainda esteja em vigência; recentemente, surgiram inúmeras propostas de implementação do protocolo ECDH usando novos modelos de curvas elípticas. Essas curvas permitem acelerar a execução do protocolo sem afetar a segurança do esquema [BL15c].

Atualmente uma das propostas que tem ganhado muita atenção encontra-se disponível em um documento publicado pela comunidade *Internet Engineering Task Force* (IETF) [LHT15]; dita proposta consiste de duas curvas elípticas, conhecidas por *Curve25519* [Ber06] e *Goldilocks* [Ham15], que suportam os níveis de segurança

Algoritmo 5 Multiplicação de pontos mediante o algoritmo de Montgomery.

Entrada: k, um número inteiro de t bits.
$x(P)$, a coordenada x de um ponto $P \in E$.

Saída: $x(kP)$, a coordenada x do ponto kP.

1: Seja $(k_t = 0, k_{t-1}, \ldots, k_0)_2$ a representação binária de k.
2: $X_{Q_0-Q_1} \leftarrow x(P)$, $Z_{Q_0-Q_1} \leftarrow 1$
3: $X_{Q_0} \leftarrow x(P)$, $Z_{Q_0} \leftarrow 1$
4: $X_{Q_1} \leftarrow 1$, $Z_{Q_1} \leftarrow 0$
5: **for** $i \leftarrow t-1$ **to** 0 **do**
6: $\quad b \leftarrow k_i \oplus k_{i+1}$
7: $\quad X_{Q_1}, X_{Q_0} \leftarrow \texttt{CondSwap}(b, X_{Q_1}, X_{Q_0})$
8: $\quad Z_{Q_1}, Z_{Q_0} \leftarrow \texttt{CondSwap}(b, Z_{Q_1}, Z_{Q_0})$
9: $\quad X_{Q_1}, Z_{Q_1}, X_{Q_0}, Z_{Q_0} \leftarrow \texttt{Ladder}(X_{Q_0-Q_1}, Z_{Q_0-Q_1}, X_{Q_1}, Z_{Q_1}, X_{Q_0}, Z_{Q_0})$
10: **end for**
11: **return** $x(kP) \leftarrow X_{Q_0}(Z_{Q_0})^{-1}$

de 128 e 224 bits, respectivamente. Essas curvas permitem calcular a multiplicação de pontos com um menor número de operações aritméticas no corpo finito $\mathbb{F}_p$.

Nesta seção serão apresentadas a curva elíptica *Curve25519*, como implementar eficientemente a aritmética de pontos e a aritmética do corpo primo $\mathbb{F}_{2^{255}-19}$, onde as instruções vetoriais possuem um papel fundamental na aceleração do protocolo.

3.4.3.1. Descrição da curva elíptica Curve25519

A curva elíptica *Curve25519* denotada por E é definida pela seguinte equação:

$$E: \quad y^2 = x^3 + 486662x^2 + x \tag{12}$$

onde as coordenadas dos pontos pertencem ao corpo primo $\mathbb{F}_p$ sendo $p = 2^{255} - 19$.

Essa curva permite que a multiplicação de pontos seja feita usando apenas a coordenada x do ponto P; isto é, dado um inteiro k e a coordenada x do ponto P, existe um algoritmo que computa a coordenada x de ponto kP; esse algoritmo foi proposto por Montgomery [Mon87] e é apresentado no Algoritmo 5. O algoritmo utiliza as coordenadas projetivas, onde a coordenada x é representada por dois valores (X, Z) sendo $x = X/Z$. Para realizar o calculo de kP são usados dois pontos acumuladores Q_0 e Q_1. O primeiro contendo o valor de P e o segundo contendo $\mathcal{O}$. O conteúdo dos acumuladores será atualizado iterativamente em função do valor dos bits de k; assim sendo, se o i-ésimo bit de k for igual a 1, então o conteúdo dos acumuladores é trocado; feito isso, a função `Ladder` atualizará os acumuladores $Q_0 \leftarrow 2Q_0$ e $Q_1 \leftarrow Q_0 + Q_1$. Após percorrer todos os bits de k, o valor de kP se encontrará armazenado no acumulador $Q_0 = (X, Z)$, finalmente a coordenada x de kP é computada como $x = X/Z$.

A função `CondSwap` deve ser implementada em tempo constante, isto é, o

Algoritmo 6 Soma e duplicação de dois pontos na curva *Curve25519* (`Ladder`).

Entrada: $X_{P-Q}, Z_{P-Q}, X_P, Z_P, X_Q, Z_Q \in \mathbb{F}_p$, e seja $\hat{a}_2 = 486662$.
Saída: $X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q} \in \mathbb{F}_p$.

1:	$A \leftarrow X_P + Z_P$	$C \leftarrow X_Q + Z_Q$	[soma]
2:	$B \leftarrow X_P - Z_P$	$D \leftarrow X_Q - Z_Q$	[subt]
3:	$DA \leftarrow A \times D$	$CB \leftarrow C \times B$	[mult]
4:	$t_1 \leftarrow DA + CB$	$t_0 \leftarrow DA - CB$	[soma/subt]
5:	$t_1 \leftarrow t_1^2$	$t_0 \leftarrow t_0^2$	[quad]
6:	$X_{P+Q} \leftarrow t_1 \times Z_{P-Q}$	$Z_{P+Q} \leftarrow t_0 \times X_{P-Q}$	[mult]
7:	$A' \leftarrow A^2$	$B' \leftarrow B^2$	[quad]
8:	$A'x \leftarrow \frac{1}{4}(\hat{a}_2 + 2) \cdot A'$	$B'y \leftarrow \frac{1}{4}(\hat{a}_2 - 2) \cdot B'$	[mult]
9:	$E \leftarrow A' - B'$	$F \leftarrow A'x - B'y$	[subt]
10:	$X_{2P} \leftarrow A' \times B'$	$Z_{2P} \leftarrow E \times F$	[mult]
11:	**return** $X_{2P}, Z_{2P}, X_{P+Q}, Z_{P+Q}$.		

mesmo processamento deve ser executado independente do valor do bit b, seguindo as recomendações apresentadas na Seção 3.3.

A função `Ladder` consiste na computação de $2Q_0$ e $Q_0 + Q_1$ sobre a curva *Curve25519*, onde as formulas da lei de grupo para esta curva estão listadas no Algoritmo 6, lembrando que as operações deste algoritmo são realizadas na aritmética do corpo primo $\mathbb{F}_{2^{255}-19}$. Esse algoritmo tem a propriedade de que algumas operações aritméticas podem ser computadas em paralelo; como podemos observar, cada linha processa o mesmo tipo de operação em ambas colunas. Esta característica nos permite um melhor aproveitamento das instruções vetoriais.

3.4.3.2. Aritmética do corpo primo

Um corpo primo $\mathbb{F}_p$ é uma estrutura algébrica que define duas operações binárias sobre um conjunto finito de números; o qual pode ser representado como o conjunto dos números inteiros no intervalo de 0 a $p-1$. A aritmética do corpo segue as mesma operações do conjunto dos inteiros, mas os resultados devem ser reduzidos modulo p para continuarem no intervalo determinado. O primo usado na curva *Curve25519* é $p = 2^{255} - 19$ e ele pertence ao conjunto dos primos *pseudo-Mersenne*, que são apropriados para fazer a redução modular de forma eficiente.

A aritmética do corpo primo opera sobre números de 256 bits. Usualmente, uma implementação que usa instruções nativas de 64 bits armazena um elemento do corpo em quatro palavras de 64 bits; algumas operações envolvem a possível propagação de resultados intermediários entre as palavras que compõem o elemento do corpo e esta propagação de dados introduz dependências na execução das instruções, implicando em uma implementação lenta.

Uma forma de minimizar a propagação de dados consiste em representar os elementos do corpo de modo que as operações possam ser computadas em paralelo, o que permitiria usar instruções vetoriais ao invés de instruções nativas de 64 bits.

Assim, para representar eficientemente um elemento $\alpha \in \mathbb{F}_{2^{255}-19}$ a tupla de coeficientes será utilizada; de agora em diante α será denotado por $\mathbf{A} = \{a_0, \ldots, a_9\}$, tal que a seguinte equação seja satisfeita:

$$\alpha = a_0 + a_1 2^{26} + a_2 2^{51} + a_3 2^{77} + a_4 2^{102} + a_5 2^{128} + a_6 2^{153} + a_7 2^{179} + a_8 2^{204} + a_9 2^{230}, \tag{13}$$

o número será representado por cinco palavras de 26 bits e cinco palavras de 25 bits.

A representação dos elementos também pode ser vista como um polinômio em x de grau 9 tal que $x^i = 2^{\lceil 25,5i \rceil}$. Por conta disso, as operações aritméticas seguem a mesma lógica das operações sobre polinômios como descrito a seguir:

- **Soma e subtração.** A soma e substração são realizadas coeficiente a coeficiente que podem ser calculadas usando instruções de 32 ou 64 bits, pois os bits restantes servirão para armazenar os bits de *carry*. Desta forma, é possível realizar uma sequencia de somas e substrações antes de precisar realizar a redução modular; isto reduz a propagação de dados e acrescenta o grau de paralelismo nas computações.
- **Multiplicação.** A multiplicação de elementos no corpo pode ser feita como a multiplicação de dois polinômios, produzindo um polinômio de grau 18. A fim de manter uma representação compacta, se aplica a redução modulo $2^{255} - 19$; ela consiste em reduzir os monômios de grau maior ou igual a 10 pela equivalência $x^{10} = 2^{255} \equiv 19$, portanto os monômios com fator x^{10} são multiplicados por 19 e adicionados com os monômios correspondentes. A distribuição dos produtos da multiplicação modular é mostrada na Figura 3.8, onde cada coluna lista os produtos a serem adicionados na potência correspondente. Nesta representação do corpo $\mathbb{F}_{2^{255}-19}$, quando i e j são ímpares existe um caso especial onde $(a_i x^i)(b_j x^j) = 2a_i b_j x^{i+j}$, por conta disso na Figura 3.8 alguns produtos aparecem multiplicados por 2.
- **Inverso multiplicativo.** O inverso multiplicativo é calculado usando a seguinte equivalência $a^{-1} \equiv a^{p-2} \equiv (\alpha_{250})^{2^5} a^{11}$. Parte da exponenciação pode ser eficientemente computada pelo algoritmo de Itoh-Tsujii [IT88] que calcula o termo $\alpha_x = a^{2^x - 1}$ mediante uma cadeia de adição. Assim, para obter α_{250} calcula-se $\alpha_5 \to \alpha_{10} \to \alpha_{20} \to \alpha_{40} \to \alpha_{50} \to \alpha_{100} \to \alpha_{200} \to \alpha_{250}$, onde a transição $\alpha_x \to \alpha_y$ encontra-se definida como $\alpha_y = (\alpha_x)^{2^{y-x}} \alpha_{y-x}$ para $x \leq y \in \mathbb{Z}^+$.

A representação apresentada permite que o calculo das operações aritméticas seja feito por uma série de operações independentes, tentando acrescentar o grau de paralelismo e agilizar a execução das operações aritméticas. Como foi visto, essas operações servem como blocos básicos para a implementação da aritmética de curvas elípticas.

3.4.3.3. Implementação vetorial da curva Curve25519

Nas seções anteriores foram mostrados algoritmos para a computação da multiplicação de pontos e uma representação dos elementos do corpo primo. Nos dois casos,

x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
a_9b_0	$38a_9b_9$	$19a_9b_8$	$38a_9b_7$	$19a_9b_6$	$38a_9b_5$	$19a_9b_4$	$38a_9b_3$	$19a_9b_2$	$38a_9b_1$
a_8b_1	a_8b_0	$19a_8b_9$	$19a_8b_8$	$19a_8b_7$	$19a_8b_6$	$19a_8b_5$	$19a_8b_4$	$19a_8b_3$	$19a_8b_2$
a_7b_2	$2a_7b_1$	a_7b_0	$38a_7b_9$	$19a_7b_8$	$38a_7b_7$	$19a_7b_6$	$38a_7b_5$	$19a_7b_4$	$38a_7b_3$
a_6b_3	a_6b_2	a_6b_1	a_6b_0	$19a_6b_9$	$19a_6b_8$	$19a_6b_7$	$19a_6b_6$	$19a_6b_5$	$19a_6b_4$
a_5b_4	$2a_5b_3$	a_5b_2	$2a_5b_1$	a_5b_0	$38a_5b_9$	$19a_5b_8$	$38a_5b_7$	$19a_5b_6$	$38a_5b_5$
a_4b_5	a_4b_4	a_4b_3	a_4b_2	a_4b_1	a_4b_0	$19a_4b_9$	$19a_4b_8$	$19a_4b_7$	$19a_4b_6$
a_3b_6	$2a_3b_5$	a_3b_4	$2a_3b_3$	a_3b_2	$2a_3b_1$	a_3b_0	$38a_3b_9$	$19a_3b_8$	$38a_3b_7$
a_2b_7	a_2b_6	a_2b_5	a_2b_4	a_2b_3	a_2b_2	a_2b_1	a_2b_0	$19a_2b_9$	$19a_2b_8$
a_1b_8	$2a_1b_7$	a_1b_6	$2a_1b_5$	a_1b_4	$2a_1b_3$	a_1b_2	$2a_1b_1$	a_1b_0	$38a_1b_9$
a_0b_9	a_0b_8	a_0b_7	a_0b_6	a_0b_5	a_0b_4	a_0b_3	a_0b_2	a_0b_1	a_0b_0
c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Figura 3.8: Forma compacta de arranjar os produtos intermediários na computação da multiplicação modular de dois elementos no corpo, $\mathbf{C} = \mathbf{A} \times \mathbf{B} \in \mathbb{F}_{2^{255}-19}$.

os algoritmos apresentavam um certo grau de paralelismo que pode ser explorado usando instruções vetoriais. A seguir será discutido como utilizar as instruções do conjunto AVX2 para implementar o protocolo.

Cada linha do Algoritmo 6 calcula duas operações aritméticas independentes, uma por coluna. Como foi discutido em [FL15], o raciocínio por trás deste processamento é reduzir o uso de instruções de alta latência, isto é, aquelas que movem dados entre a parte baixa e alta dos registradores de 256 bits. Partindo dessa observação, os primeiros 128 bits dos registradores `YMM` são usados para processar as operações da coluna esquerda e os bits restantes para processar as operações da coluna direita. Assim, os elementos do corpo são associados aos registradores vetoriais da seguinte maneira: denota-se por $\langle \mathbf{A}, \mathbf{B} \rangle$ as *tuplas entrelaçadas* de $\mathbf{A}$ e $\mathbf{B}$ (duas tuplas representando dois elementos do corpo), para representar cinco registradores `YMM` contendo:

$$\langle \mathbf{A}, \mathbf{B} \rangle_i = [a_{2i+1}, a_{2i}, b_{2i+1}, b_{2i}] \quad \text{para } 0 \leq i < 5. \tag{14}$$

assim, o registrador de 256 bits conterá dois coeficientes de duas tuplas armazenados em palavras de 64 bits. Contudo, a implementação das operações aritméticas ainda é beneficiada pelas instruções vetoriais, pois as operações podem usar vetores de 128 bits.

Uma das operações mais críticas em termos de desempenho é a multiplicação modular, pois requer calcular uma grande quantidade de multiplicações de coeficientes. O Algoritmo 7 apresenta a sequência de instruções para o processamento da multiplicação modular no corpo $\mathbb{F}_{2^{255}-19}$. Nas linhas 2-5 é inicializado um conjunto de registradores que contém alguns coeficientes de $\langle \mathbf{B}, \mathbf{D} \rangle$ multiplicados por 2; durante o ciclo principal, nas linhas 6-18, as variáveis Z_i acumularão o conteúdo dos produtos intermediários a_ib_j; e no último ciclo, nas linhas 19-22, a redução modular é calculada sobre os coeficientes de grau maior ou igual a 10.

A instrução `MUL` possui uma latência de 5 ciclos e ao fim de cada ciclo é possível começar a execução de uma nova multiplicação, fazendo com que a latência total de uma sequência de multiplicações independentes seja reduzida. Adicionalmente, enquanto uma multiplicação está sendo processada na porta 0 as outras portas de execução do processador ficam disponíveis para execução de outros tipos de instruções; por exemplo, pode-se fazer operações de acesso a memória concorrentemente às multiplicações.

A saída da função de multiplicação produz uma tupla entrelaçada onde cada palavra de 64 bits contém um coeficiente de aproximadamente 52 bits. Posteriormente, essa tupla pode ser usada para computar operações de soma ou substração. Entretanto, se duas multiplicações precisam ser computadas sucessivamente, o tamanho dos coeficientes devem ser reduzidos tal que sejam menor que 32 bits visto que a instrução `MUL` computa apenas produtos de palavras de 32 bits.

O processamento antes mencionado é denominado de *redução de coeficientes* e assegura que cada coeficiente seja de aproximadamente 26 bits. Neste processamento, cada coeficiente c_i é dividido em três partes (l_i, m_i e h_i), de modo que l_i contém os primeiros 26 bits, m_i os próximos 25 bits e h_i os bits remanescentes. Após a divisão, a redução dos coeficientes é calculada por:

$$\begin{aligned} c_0' &= l_0 + 19(m_9 + h_8), \\ c_1' &= l_1 + m_0 + 19h_9, \\ c_i' &= l_i + m_{i-1} + h_{i-2} \text{ para } 2 \leq i < 10. \end{aligned} \tag{15}$$

O Algoritmo 8 mostra o processamento realizado usando as instruções AVX2. No primeiro ciclo, nas linhas 1-6, é feita a divisão das palavras para calcular os vetores l, m e h. No segundo ciclo, nas linhas 7-9, o algoritmo arranja os coeficientes do vetor m, alinhando-os com l e h. Das linhas 10-15, são computadas as multiplicações por 19 dos termos m_9, h_8 e h_9; finalmente, o último ciclo calcula a soma dos vetores l, m e h. Cada coeficiente da tupla entrelaçada que é retornada tem um tamanho de aproximadamente 26 bits.

3.4.3.4. Testes de desempenho

O impacto causado pelas instruções AVX2 na computação do protocolo ECDH foi mostrada no trabalho de [FL15]. A Tabela 3.4 mostra uma comparação das implementações da curva elíptica *Curve25519*. Note que as implementações que usam instruções vetoriais obtiveram um melhor desempenho sobre as implementações com instruções nativas de 64 bits.

Para mostrar o desempenho obtido no protocolo ECDH usando a curva elíptica *Curve25519*, o seguinte teste será feito.

Teste 5. *Calcular a quantidade de operações de acordo de chaves calculadas por segundo para o conjunto de curvas suportado pela biblioteca OpenSSL e a curva elíptica Curve25519 usando AVX2.*

Algoritmo 7 Sequencia de instruções para calcular a multiplicação modular no corpo $\mathbb{F}_{2^{255}-19}$ usando as instruções AVX2.

Entrada: $\langle \mathbf{A}, \mathbf{C} \rangle$ e $\langle \mathbf{B}, \mathbf{D} \rangle$, duas tuplas entrelaçadas.
Saída: $\langle \mathbf{E}, \mathbf{F} \rangle$ uma tupla entrelaçada tal que $\mathbf{E} = \mathbf{A} \times \mathbf{B}$ e $\mathbf{F} = \mathbf{C} \times \mathbf{D}$.

1: $Z_i \leftarrow 0$ para $0 \leq i < 10$
2: **for** $i \leftarrow 0$ **to** 4 **do**
3: $\quad \langle \mathbf{B}', \mathbf{D}' \rangle_i \leftarrow \texttt{ALIGNR}(\langle \mathbf{B}, \mathbf{D} \rangle_{i+1 \bmod 5}, \langle \mathbf{B}, \mathbf{D} \rangle_i)$
4: $\quad \langle \mathbf{B}', \mathbf{D}' \rangle_i \leftarrow \texttt{SHLV}(\langle \mathbf{B}', \mathbf{D}' \rangle_i, [0,1,0,1])$
5: **end for**
6: **for** $i \leftarrow 0$ **to** 4 **do**
7: $\quad U \leftarrow \texttt{SHUFFLE}(\langle \mathbf{A}, \mathbf{C} \rangle_i, 0)$
8: $\quad$ **for** $j \leftarrow 0$ **to** 4 **do**
9: $\quad\quad Z_{i+j} \leftarrow \texttt{ADD}(Z_{i+j}, \texttt{MUL}(U, \langle \mathbf{B}, \mathbf{D} \rangle_j))$
10: $\quad$ **end for**
11: $\quad V \leftarrow \texttt{SHUFFLE}(\langle \mathbf{A}, \mathbf{C} \rangle_i, 1)$
12: $\quad$ **for** $j \leftarrow 0$ **to** 3 **do**
13: $\quad\quad Z_{i+j+1} \leftarrow \texttt{ADD}(Z_{i+j+1}, \texttt{MUL}(V, \langle \mathbf{B}', \mathbf{D}' \rangle_j))$
14: $\quad$ **end for**
15: $\quad W \leftarrow \texttt{MUL}(V, \langle \mathbf{B}', \mathbf{D}' \rangle_4)$
16: $\quad Z_i \leftarrow \texttt{ADD}(Z_i, \texttt{BLEND}(W, [0,0,0,0], 0101))$
17: $\quad Z_{i+5} \leftarrow \texttt{ADD}(Z_{i+5}, \texttt{BLEND}(W, [0,0,0,0], 1010))$
18: **end for**
19: **for** $i \leftarrow 0$ **to** 4 **do**
20: $\quad 19Z_{i+5} \leftarrow \texttt{ADD}(\texttt{ADD}(\texttt{SHL}(Z_{i+5}, 4),\ \texttt{SHL}(Z_{i+5}, 1)), Z_{i+5})$
21: $\quad \langle \mathbf{E}, \mathbf{F} \rangle_i \leftarrow \texttt{ADD}(Z_i,\ 19Z_{i+5})$
22: **end for**
23: **return** $\langle \mathbf{E}, \mathbf{F} \rangle$

c_i		
13	25	26
h_i	m_i	l_i

(a) Divisão do coeficiente c_i.

l_9	l_8	l_7	l_6	l_5	l_4	l_3	l_2	l_1	l_0
+	+	+	+	+	+	+	+	+	+
m_8	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0	$19m_9$
+	+	+	+	+	+	+	+	+	+
h_7	h_6	h_5	h_4	h_3	h_2	h_1	h_0	$19h_9$	$19h_8$
c'_9	c'_8	c'_7	c'_6	c'_5	c'_4	c'_3	c'_2	c'_1	c'_0

(b) Após da soma os coeficientes são de aproximadamente 26 bits.

Figura 3.9: Redução de coeficientes. Na esquerda, cada coeficiente c_i é divido em trés partes chamadas l_i, m_i e h_i. Essas partes são adicionadas como é mostrado na figura da direita. Note que m_9, h_8 e h_9 são reduzidos modulo p.

Algoritmo 8 Sequencia de instruções para calcular a redução de coeficientes usando instruções AVX2.

Entrada: $\langle \mathbf{A}, \mathbf{B} \rangle$, uma tupla entrelaçada.
Saída: $\langle \mathbf{A}, \mathbf{B} \rangle$, tal que $|c_{2i}| \leq 27$ e $|c_{2i+1}| \leq 26$ para $0 \leq i < 5$ e $c \in \{a, b\}$.

1: **for** $i \leftarrow 0$ **to** 4 **do**
2: $\quad L_i \leftarrow \texttt{AND}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [2^{25}-1, 2^{26}-1, 2^{25}-1, 2^{26}-1])$
3: $\quad M_i \leftarrow \texttt{SHRV}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [25, 26, 25, 26])$
4: $\quad M_i \leftarrow \texttt{AND}(M_i, [2^{26}-1, 2^{25}-1, 2^{26}-1, 2^{25}-1])$
5: $\quad H_i \leftarrow \texttt{SHRV}(\langle \mathbf{A}, \mathbf{B} \rangle_i, [51, 51, 51, 51])$
6: **end for**
7: **for** $i \leftarrow 0$ **to** 4 **do**
8: $\quad M'_i \leftarrow \texttt{ALIGNR}(M_i, M_{i-1 \bmod 5})$
9: **end for**
10: $H_4 \leftarrow \texttt{SHRV}(\langle \mathbf{A}, \mathbf{B} \rangle_8, [51, 26, 51, 26])$
11: $U \leftarrow \texttt{ADD}(H_4, \texttt{SHR}(H_4, 64))$
12: $19U \leftarrow \texttt{ADD}(\texttt{ADD}(\texttt{SHR}(U, 4), \texttt{SHR}(U, 1)), U)$
13: $T \leftarrow \texttt{AND}(19U, [0, 2^{26}-1, 0, 2^{26}-1])$
14: $S \leftarrow \texttt{SHR}(19U, [0, 26, 0, 26])$
15: $H_4 \leftarrow \texttt{UPCK}(T, S)$
16: **for** $i \leftarrow 0$ **to** 4 **do**
17: $\quad \langle \mathbf{A}, \mathbf{B} \rangle_i \leftarrow \texttt{ADD}(\texttt{ADD}(L_i, M'_i), H_{i-1 \bmod 5})$
18: **end for**
19: **return** $\langle \mathbf{A}, \mathbf{B} \rangle$

Implementação	Processador	Instr. Vetoriais	Acordo de Chaves
NaCl [BLS13]	Core i7-4770	Não	261.000
amd64-51 [BL15b]	Core i7-4770	Não	163.200
amd64-51 [BL15b]	Xeon E3-1275 V3	Não	161.600
AVX [Cho15]	Core i5-3210M	Sim	159.100
AVX2 [FL15]	Core i7-4770 $^{(\alpha)}$	Sim	156.500

Tabela 3.4: Tempos de execução do protocolo de acordo de chaves usando a curva *Curve25519*; os tempos são reportados em ciclos.

Curva Elíptica	Operações/segundo
B-283	2.159
K-283	2.268
P-256	14.384
Curve25519	21.787

Tabela 3.5: Comparação do desempenho das curvas padrão do NIST contra a curva elíptica Curve25519.

Como pode ser visto na Tabela 3.5, as curvas binárias do NIST, B-283 e K-283, que estão implementadas na biblioteca do OpenSSL reportam uma taxa baixa de computações por segundo, atingindo na média dois mil operações por segundo. A implementação da curva P-256 é mais otimizada obtendo acima de quatorze mil operações por segundo; no entanto o desempenho oferecido pela curva *Curve25519* é 1.5 vezes mais rápido do que a curva NIST P-256.

Observações: A latência do protocolo ECDH pode ser reduzida com o uso de curvas específicas que reduzem o número de operações a serem calculadas; além disso, a aritmética do corpo primo foi adaptada para evitar propagação de dados, acrescentando o grau de paralelismo.

3.5. Comentários finais

Os algoritmos criptográficos garantem a segurança da informação através de meios de comunicação propensos a ataques. Algumas vezes, esses ataques são bem sucedidos devido à presença de implementações que não cumprem com os requisitos mínimos de segurança; por exemplo, a proteção contra ataques de canais laterais.

A implementação destes algoritmos precisa de um estudo meticuloso do fluxo da informação, dos acessos a memória, do tipo de instruções presentes na arquitetura e dos possíveis ataques. Portanto, a implementação de algoritmos criptográficos apresenta desafios em múltiplas dimensões.

A relevância que a segurança da informação tem ganhado nos últimos anos tem impactado no projeto dos processadores e arquiteturas atuais. Prova disso, é o suporte em *hardware* adicionado para o padrão de encriptação de dados AES, como foi visto nas sessões anteriores. Esse não é um caso isolado, pois a próxima arquitetura *Skylake* dará suporte à função de resumo SHA-2 [Cor13].

Este trabalho se aprofundou na eficiência das operações, visando aproveitar os recursos disponíveis nos processadores recentes. Atualmente existe uma tendência forte por explorar o paralelismo mediante o escalonamento de múltiplas instruções por ciclo, a inclusão de unidades de execução mais potentes, a divisão de tarefas, entre outras. Por conta disso é imprescindível encontrar técnicas que adaptem os algoritmos às arquiteturas modernas. Em particular, o conjunto de instruções vetoriais AVX2 se apresentou como uma alternativa bem sucedida para acelerar a execução de algoritmos criptográficos.

Consideramos que as técnicas aqui apresentadas podem ser dimensionadas para outros conjuntos de instruções vetoriais. Por exemplo, a arquitetura do co-processador Xeon Phi possui registradores de 512 bits, mais de 60 unidades de processamento e o novo conjunto de instruções vetoriais AVX-512 [Cor13]. Este tipo de arquitetura está voltado para o processamento de grandes quantidades de informação, o que a torna uma arquitetura ideal para a implementação de algoritmos criptográficos com alto grau de paralelismo.

Finalmente, é intenção dos autores incentivar ao leitor interessado nos tópicos aqui descritos a aprofundar mais seus conhecimentos na literatura recente. Caso mais informação seja necessária, sugerimos entrar em contato com a lista de autores do minicurso, pois alguns dos resultados aqui apresentados fazem parte de projetos de pesquisa em andamento.

A. Plataforma de teste de desempenho

Para os testes de desempenho realizados neste minicurso foram utilizados os computadores HW-DESKTOP e HW-ULTRA; ambos usam um processador *Haswell* da Intel. Eles são capazes de rodar instruções nativas de 64 bits, vetoriais de 128 e 256 bits e suportam os conjuntos de instruções AVX2 e AES-NI.

É importante mencionar que apesar de possuírem a mesma arquitetura, o computador HW-DESKTOP é um processador de propósito geral indicado para computadores de mesa, já o computador HW-ULTRA é um processador de baixo consumo energético, voltado para a computação móvel. Na Tabela 3.6 são mostradas as especificações técnicas de cada computador.

	HW-DESKTOP	HW-ULTRA
Processador	Core i7-4770	Core i5-4350U
Frequência	3.4 GHz	1.4 GHz
Memoria RAM	4 GB	4 GB
Sistema operacional	Fedora 18	Fedora 20

Tabela 3.6: Especificações técnicas dos computadores HW-DESKTOP e HW-ULTRA usados neste trabalho.

Referências

[AcKKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction. In Masayuki Abe, editor, *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BCJ+05] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In *Advances in Cryptology–EUROCRYPT 2005*, pages 36–57. Springer, 2005.

[BDPA07] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007.

[BDPA08] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *Advances in Cryptology – Eurocrypt 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008. `http://sponge.noekeon.org/`.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and GV Assche. The keccak reference. *Submission to NIST (Round 3)*, 13, 2011.

[BDPV14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sakura: A Flexible Coding for Tree Hashing. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, volume 8479 of *Lecture Notes in Computer Science*, pages 217–234. Springer International Publishing, 2014.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[Bih97] Eli Biham. A fast new DES implementation in software. In *Fast Software Encryption*, pages 260–272. Springer, 1997.

[BJS07] Elaine B. Barker, Don Johnson, and Miles E. Smid. SP 800-56A. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007.

[BL15a] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, September 2015.

[BL15b] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Accessed on 20 March 2015, March 2015.

[BL15c] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to` accessed 20 March 2015, 2015.

[BLS13] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. NaCl: Networking and Cryptography library. `http://nacl.cr.yp.to/`, October 2013.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks Against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer, 2011.

[Cho15] Tung Chou. Fastest Curve 25519 Implementation Ever. In National Institute of Standards and Technology, editors, *Workshop on Elliptic Curve Cryptography Standard*. National Institute of Standards and Technology, June 2015.

[CL14] Roberto Cabral and Julio López. Software implementation of SHA-3 family using AVX2. In *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, volume XIV, pages 330–333. Sociedade Brasileira de Computação, 2014.

[Cor] Intel Corporation. Hardware Design Site Archives. Intel® Pentium processor with MMX™ technology documentation. `http://www.intel.com/design/archives/Processors/mmx/`.

[Cor11] Intel Corporation. Intel® Advanced Vector Extensions Programming Reference, June 2011. Disponível em `https://software.intel.com/sites/default/files/m/f/7/c/36945`.

[Cor13] Intel Corporation. Intel Instruction Set Architecture Extensions. Available at `https://software.intel.com/en-us/intel-isa-extensions`, July 2013.

[Cor14] Intel Corporation. Intel® Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, February 2014.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[DR01] Joan Daemen and Vincent Rijmen. Algorithm Alley: Rijndael: The Advanced Encryption Standard. *Dr. Dobb's Journal of Software Tools*, 26(3):137–139, March 2001.

[FL15] Armando Faz-Hernández and Julio López. Fast Implementation of Curve25519 Using AVX2. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2015.

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[Fog14] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, December 2014.

[GGF+15] Vinodh Gopal, Sean Gulley, Wajdi Feghali, Ilya Albrekht, and Dan Zimmerman. Improving OpenSSL Performance. Technical report, Intel Corporation, May 2015. Disponível em `https://software.intel.com/en-us/articles/improving-openssl-performance`.

[Gue10] Shay Gueron. Intel advanced encryption standard (AES) instructions set. *Intel White Paper, Rev*, 3, 2010.

[Ham09] Mike Hamburg. Accelerating AES with Vector Permute Instructions. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2009.

[Ham15] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. `http://eprint.iacr.org/`.

[HSH+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[IMPR09] Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other non-random properties for step-reduced

SHA-256. In *Selected Areas in Cryptography*, pages 276–293. Springer, 2009.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Inf. Comput.*, 78(3):171–177, September 1988.

[jCPB+12] Shu jen Chang, Ray Perlner, William E Burr, Meltem Sönmez Turan, John M Kelsey, Souradyuti Paul, and Lawrence E Bassham. *Third-round report of the SHA-3 cryptographic hash algorithm competition.* US Department of Commerce, National Institute of Standards and Technology, 2012.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *16th Annual International Cryptology Conference (CRYPTO 1996)*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 1–17, 2009.

[KSD13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS), 2013.

[LHT15] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security, September 2015. Disponível em `https://datatracker.ietf.org/doc/draft-irtf-cfrg-curves/`.

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Stanford University, 1979.

[Mer88] RalphC. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin Heidelberg, 1988.

[MN07] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on Intel Core2 processor. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 121–134, 2007.

[Mon87] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[Nat93] National Institute of Standards and Technology. *FIPS PUB 180: Secure Hash Standard.* National Institute for Standards and Technology, Gaithersburg, MD, USA, May 1993.

[Nat95] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, April 1995. Supersedes FIPS PUB 180 1993 May 11.

[Nat01a] National Institute of Standards and Technology. *FIPS PUB 197, ADVANCED ENCRYPTION STANDARD (AES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, November 2001.

[Nat01b] National Institute of Standards and Technology. *NIST Special Publication 800-38A. Recommendation for Block Cipher Modes of Operation*. National Institute for Standards and Technology, Gaithersburg, MD, USA, December 2001.

[Nat02] National Institute of Standards and Technology. *FIPS PUB 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2*. National Institute for Standards and Technology, Gaithersburg, MD, USA, August 2002. Supersedes FIPS PUB 180-1 1995 April.

[Nat08a] National Institute of Standards and Technology. *FIPS PUB 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 2008. Supersedes FIPS PUB 180-2 2002 August.

[Nat08b] National Institute of Standards and Technology. *FIPS PUB 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 2008. Supersedes FIPS PUB 180-3 October 2008.

[Nat15] National Institute of Standards and Technology. *FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. National Institute for Standards and Technology, Gaithersburg, MD, USA, August 2015.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan 2005*, 2005.

[Rab78] Michael O Rabin. Digitalized signatures. *Foundations of secure computation*, 78:155–166, 1978.

[RO05] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In *Topics in Cryptology–CT-RSA 2005*, pages 58–71. Springer, 2005.

[Rot11] Jeffrey Rott. Intel AESNI Sample Library. Technical report, Intel Corporation, May 2011. Disponível em `https://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library`.

 ©2015 SBC — Soc. Bras. de Computação

[RSD06] Chester Rebeiro, David Selvakumar, and A Devi. Bitslice implementation of AES. *Cryptology and Network Security*, pages 203–212, 2006.

[Sch95] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Sta10] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

[The03] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/-TLS. `www.openssl.org`, April 2003.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.

[Yee15] Alexander Yee. FeatureDetector, April 2015. Disponível em `https://github.com/Mystical/FeatureDetector`.

[Yuv78] G. Yuval. *How to Swindle Rabin.* Rapport nr. IR. Vrije Universiteit, Wiskundig Seminarium, 1978.

 ©2015 SBC — Soc. Bras. de Computação