Tiago Vanderlei de Arruda

Análise de algoritmos paralelos de ECC em dispositivos móveis *multicore*

Sorocaba, SP

15 de Dezembro de 2014

Tiago Vanderlei de Arruda

Análise de algoritmos paralelos de ECC em dispositivos móveis *multicore*

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Sistemas Computacionais.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Tiemi Christine Sakata

Coorientador: Yeda Regina Venturini

Sorocaba, SP

15 de Dezembro de 2014

Tiago Vanderlei de Arruda
Análise de algoritmos paralelos de ECC em dispositivos móveis *multicore*/ Tiago Vanderlei de Arruda. – Sorocaba, SP, 15 de Dezembro de 2014-
139 p. : il. (algumas color.) ; 30 cm.

Orientador: Tiemi Christine Sakata

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCCS, 15 de Dezembro de 2014.

1. Curva elíptica. 2. Dispositivo móvel. 3. Multiplicação paralela. I. Tiemi Christine Sakata. II. Universidade Federal de São Carlos - UFSCar. III. Centro de Ciências em Gestão e Tecnologia - CCGT. IV. Análise de algoritmos paralelos de ECC em dispositivos móveis *multicore*.

CDU 02:141:005.7

Tiago Vanderlei de Arruda

Análise de algoritmos paralelos de ECC em dispositivos móveis *multicore*

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Sistemas Computacionais.

Trabalho aprovado. Sorocaba, SP, 15 de Dezembro de 2014:

Tiemi Christine Sakata
Orientador

Paulo Sérgio Licciardi Messeder Barreto
Convidado 1

Hélio Crestana Guardia
Convidado 2

Sorocaba, SP
15 de Dezembro de 2014

Agradecimentos

Agradeço,

a Deus, pela oportunidade, condição e força para concluir este trabalho.

aos meus pais Vanderlei Roberto de Arruda e Cleuza Mendonça Arruda pelo apoio e compreensão. À meus irmãos Roberto Vanderlei de Arruda e Vanderlei Roberto de Arruda Junior, pela parceria de longos anos.

às minhas amigas e orientadoras, Tiemi Christine Sakata e Yeda Regina Venturini pelas longas conversas, competência e paciência.

a todos os professores do PPGCCS pelas excelentes aulas ministradas, competência e apoio. Ao atual secretário do PPGCCS Roberto Romualdo Marvulle, pelo serviço prestado.

aos colegas de laboratório pelas conversas produtivas.

aos membros da banca Paulo Sérgio Licciardi Messeder Barreto e Hélio Crestana Guardia, pelas contribuições relevantes a este trabalho.

ao CNPq e aos membros do Laboratório de Arquitetura e Redes de Computadores (LARC) da Escola Politécnica da Universidade de São Paulo pelo apoio.

à CAPES pelo auxílio financeiro que possibilitou a realização deste trabalho.

"Por que quem compreendeu o intento do Senhor ?
ou quem foi seu conselheiro?
Ou quem lhe deu primeiro a Ele, para que lhe seja recompensado?
Por que Dele e por Ele, e para Ele, são todas as coisas;
glória pois a Ele eternamente. Amém."
(Aos Romanos, 11:34-36, Bíblia Sagrada)

Resumo

A adoção de processadores *multi-core* se deve à necessidade de expandir a capacidade computacional, o que vem sendo feito em dispositivos móveis, devido à alta disponibilidade de aplicações *online* em tais dispositivos. A criptografia de curvas elípticas (ECC) pode ser utilizada em tais aplicações, a fim de garantir o sigilo na comunicação realizada pelo dispositivo. Este algoritmo possui sua segurança baseada no problema do logaritmo discreto em curvas elípticas (ECDLP), que é mais difícil de solucionar que o problema do RSA, possuindo segurança equivalente ao custo de chaves muito menores, reduzindo portanto o custo computacional das soluções que o utilizam. A multiplicação escalar é a operação principal e mais custosa do ECC e envolve o cálculo de diversas operações modulares. Algoritmos de multiplicação modular paralelos foram avaliados neste trabalho, cujos tempos de execução foram comparados com os de alguns sequenciais. Foram realizados experimentos em uma placa de desenvolvimento SabreLite IMX6Quad, com arquitetura similar a de um dispositivo móvel. Nesta plataforma, foi avaliada a transição do modo de baixa para o de alta frequência, realizada pela CPU no modo *ondemand* durante a execução dos algoritmos. A relação de proporção entre os tempos dos algoritmos avaliados no modo *performance* foi similar à obtida no modo *powersave*. Alguns algoritmos paralelos foram mais rápidos que os sequenciais nas operações com operandos a partir de 768 bits. Ao avaliar o comportamento de cada algoritmo, quando incorporado no cálculo da multiplicação escalar, observou-se que os paralelos foram mais rápidos nas operações com uma curva supersingular de 1536 bits.

Palavras-chaves: Curva elíptica. Dispositivo móvel. Multiplicação paralela.

Abstract

Multicore processors adoption is due to the need of expansion on the computational capacity, what have been done in mobile devices, due to the high availability of online applications in such devices. Elliptic curve cryptography (ECC) can be used in these applications, to ensure the confidentiality in the communication performed by the mobile device. This algorithm has its security on the hardness to solve the elliptic curve discrete logarithm problem (ECDLP), what is harder to solve than RSA's problem, owning equivalent security at the cost of much smaller keys, hence reducing the computational cost of the solutions which implement it. Scalar multiplication is the main and most costly operation in ECC and is composed by the computation of many modular operations. Parallel modular multiplication algorithms where evaluated in this work, which timings were compared with timings of some sequential algorithms. Experiments were performed on a SabreLite IMX6Quad development board, with an architecture similar to a mobile device. On this platform, it was evaluated the transition from the low to the high frequency of CPU, which occurs in ondemand CPU mode during the execution of the algorithms. The relation of proportion among the timings of the algorithms evaluated on performance mode was similar to the powersave CPU mode. Some parallel algorithms were faster than the sequentials in operations among operands with at least 768 bits. Evaluating the behavior of each algorithm when integrated in the computation of scalar multiplication, it was observed that the parallels were faster in operations with a 1536-bit supersingular curve.

Key-words: Elliptic curve. Mobile device. Parallel multiplication.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CPU	*Central Processing Unit*
DSS	*Digital Signature Standard*
ECC	*Elliptic Curve Cryptography*
ECDH	*Elliptic curve Diffie–Hellman*
ECDLP	*Elliptic curve Discrete Logarithm Problem*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
ECElG	*Elliptic curve El Gamal*
ECMQV	*Elliptic Curve Menezes-Qu-Vanstone*
ECSS	*Elliptic Curve Schnorr Signature*
FIPS	*Federal Information Processing Standards*
NIST	*National Institute of Standards and Technology*
GCC	*GNU Compiler Collection*
GMP	*GNU Multiple Precision Arithmetic Library*
MARIA	*Modular ARIthmetic Algorithms*
MOF	*Mutual Opposite Form*
NAF	*Non-Adjacent Form*
RELIC	*RELIC is an Efficient LIbrary for Cryptography*
RSA	*Rivest-Shamir-Adleman*
SPA	*Simple Power Analysis*
VPN	*Virtual Private Network*

Lista de símbolos

$\approx$	Aproximadamente a
$\equiv$	Congruente
log	Logaritmo
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{R}$	Conjunto dos números reais
Δ	Delta
$\sigma(t)$	Desvio padrão do conjunto t
$\neq$	Diferente de
GF	Galois field
$=$	Igual a
mdc	Máximo divisor comum
$\bar{t}$	Média do conjunto t
$\tilde{t}$	Mediana do conjunto t
μs	Microsegundos
mod	Modulo
$\in$	Pertence a
$GF(p)$	Corpo finito primo
$GF(2^m)$	Corpo finito binário (ou característica dois)
$\mathcal{A}$	Sistema de coordenadas *Afim*
$\mathcal{P}$	Sistema de coordenadas *Standard*
$\mathcal{J}$	Sistema de coordenadas *Jacobian*
$\mathcal{J}^c$	Sistema de coordenadas *Chudnovsky*
$\mathcal{J}^m$	Sistema de coordenadas *Jacobian* modificado

$\mathcal{LD}$	Sistema de coordenadas *López-Dahab*
RPM	Redução parcial de Montgomery
RPB	Redução parcial de Barrett
$\mathbb{Z}/p\mathbb{Z}$	Inteiros módulo p
$(E, +)$	Operação de adição, sobre a curva elíptica E

Sumário

1 Introdução

O advento dos processadores *multi-core* tanto em computadores pessoais, como em dispositivos móveis e empresariais se deve às limitações de projeto existentes na construção de processadores mais rápidos utilizando a arquitetura *single core*.

A necessidade de expandir a capacidade computacional e viabilizar o uso de serviços até então acessíveis apenas em computadores pessoais, levou à busca por processadores que ao mesmo tempo fossem mais rápidos e eficientes. Tal fato tem impulsionado o desenvolvimento de processadores *multi-core* para a linha de dispositivos móveis por empresas como ARM e Intel.

A alta disponibilidade de serviços *online* em dispositivos móveis, tais como comércio eletrônico, rede social, *email*, serviço bancário e outros, faz necessário o uso de protocolos de comunicação segura para garantir a autenticação e o sigilo das informações trafegadas na rede durante a comunicação. Tal segurança pode ser garantida com o uso da criptografia.

O conceito de criptografia assimétrica foi introduzido inicialmente por Diffie et al. (DIFFIE; HELLMAN, 1976) e permite que as operações criptográficas de cifração e decifração sejam realizadas por meio de duas chaves diferentes: pública e privada. Os protocolos de criptografia assimétrica mais utilizados são: RSA e ECC.

O RSA foi o primeiro algoritmo de criptografia assimétrica proposto na literatura (RIVEST; SHAMIR; ADLEMAN, 1978), cuja segurança está na dificuldade da solução do problema da fatoração de inteiros grandes. Em 1985 foi proposto o algoritmo de Criptografia de Curvas Elípticas (ECC) independentemente por Koblitz (KOBLITZ, 1987) e Miller (MILLER, 1986), cuja segurança está na dificuldade de encontrar o logaritmo discreto para pontos em uma curva elíptica definida sobre grupos finitos de ordem alta (KOVALENKO; KOCHUBINSKII, 2003).

Da mesma forma que o RSA, a criptografia de curvas elípticas pode ser utilizada em protocolos de assinatura digital, distribuição segura de chave, e cifração. Protocolos criptográficos baseados em curvas elípticas requerem menos recursos computacionais do que o RSA, sendo portanto mais utilizados em dispositivos sem fio e celulares modernos (GUPTA; SILAKARI, 2012). Além disso, protocolos baseados em curvas elípticas permitem a implementação mais eficiente de segurança nos serviços disponíveis em dispositivos sem fio, dentre os quais pode-se destacar: *e-mail* seguro, navegação na *web*, rede privada virtual (VPN) (LAUTER, 2004).

O aumento crescente na capacidade computacional dos computadores e a constante descoberta de vulnerabilidades faz com que o tempo necessário para quebra da segurança

dos algoritmos de criptografia seja reduzido. O aumento na segurança é então suprido inicialmente pelo aumento no tamanho da chave utilizada. O tamanho da chave do RSA para uso seguro tem aumentado nos últimos anos, e consequentemente a carga de processamento das aplicações que utilizam este algoritmo também está aumentando. Para reduzir a carga de processamento de aplicações que realizam transações seguras, como sites de comércio eletrônico, algoritmos de criptografia mais eficientes são necessários.

Embora o nível de confiança do algoritmo ECC seja menor que o do RSA, devido a ser um algoritmo mais recente, ele oferece segurança equivalente com o uso de chaves menores, reduzindo a carga de processamento decorrente do cálculo criptográfico.

Um protocolo de criptografia baseado em curvas elípticas é composto por operações de adição de ponto em uma curva elíptica E $(E, +)$. A multiplicação escalar é operação principal presente nestes protocolos. Esta operação pode ser calculada segundo as operações de adição e dobro de ponto definidas para o sistema de criptografia adotado. As operações de ponto, por outro lado, são calculadas através de operações aritméticas modulares, que são o foco desta dissertação.

Existem trabalhos na literatura que paralelizaram a multiplicação escalar, calculando kP através da segmentação do escalar k, e paralelizando o produto de cada segmento por P (AL-SOMANI; IBRAHIM, 2009), (AL-SOMANI; FAYOUMI; IBRAHIM, 2014), (BASU, 2012). O paralelismo neste nível também foi implementado, calculando a adição e dobro de ponto por *threads* distintas (ANAGREH; SAMSUDIN; OMAR, 2014). Versões paralelas da multiplicação modular de Montgomery foram propostas tanto em FPGA (CHEN; SCHAUMONT, 2011), como em *software* (GIORGI; IMBERT; IZARD, 2013), (BAKTIR; SAVAS, 2013), (KAIHARA; TAKAGI, 2008), (SAKIYAMA et al., 2011).

Este trabalho tem como por objetivo avaliar o desempenho da multiplicação escalar, paralelizando, no entanto, as operações de multiplicação e quadrado modulares, que são as mais custosas, dentro do sistema de coordenadas avaliado.

Esta dissertação está organizada da seguinte maneira. Inicialmente, são apresentados no Capítulo 2 o embasamento teórico da estrutura geral dos protocolos baseados em curvas elípticas, seguido da descrição dos algoritmos mais conhecidos e utilizados em cada nível do ECC.

Os algoritmos de multiplicação modular sequenciais e paralelos são analisados no Capítulo 3. Foram realizados *benchmarks* destes algoritmos nos modos de CPU *powersave*, *performance* e *ondemand* com operandos entre 128 e 4096 bits. São analisados os resultados de diversos experimentos realizados em alguns cenários no modo de CPU *ondemand*, a fim de identificar a causa da variação significativa dos tempos dos experimentos neste modo.

A multiplicação escalar é analisada no Capítulo 4. Para realizar as operações, são

utilizadas curvas NIST 256, 384 e 521, além de uma curva supersingular de 1536 bits disponível na biblioteca RELIC. Os algoritmos de multiplicação modular avaliados no Capítulo 3 foram integrados no cálculo da multiplicação escalar, sendo necessário adaptar algumas das implementações para realização dos cálculos com operandos de 256, 384, 521 e 1536 bits em uma arquitetura de 32 bits. São analisados os tempos de execução nos três modos de CPU (*ondemand*, *powersave* e *performance*).

2 Protocolos de criptografia baseados em curvas elípticas

A sigla ECC (*Elliptic Curve Cryptography*, em português, criptografia baseada em curvas elípticas) refere-se à aplicação de curvas elípticas em protocolos criptográficos. Os protocolos de criptografia baseados em curvas elípticas utilizam curvas definidas sobre um grupo abeliano finito, tal que todas as variáveis e coeficientes pertencem a um corpo finito (GF).

A segurança dos protocolos baseados em ECC (criptografia de curvas elípticas) depende da dificuldade de solução do problema do logaritmo discreto em curvas elípticas (ECDLP). Seja E uma curva elíptica definida sobre um corpo finito $GF(q)$, $P \in E(GF(q))$ um ponto de ordem h, e $Q \in \langle P \rangle$ um ponto, sendo $\langle P \rangle$ é um conjunto composto por todos os pontos gerados por P. A dificuldade está em encontrar $l \in [0, h-1]$, tal que $Q = lP$. O inteiro l é chamado o logaritmo discreto de Q na base P ($l = \log_P Q$) (HANKERSON; MENEZES; VANSTONE, 2003).

Existem diversos protocolos de criptografia baseados no ECC, dentre os quais pode-se citar: ECDH (*Elliptic curve Diffie–Hellman*) (DIFFIE; HELLMAN, 1976), ECMQV (*Elliptic Curve Menezes-Qu-Vanstone*) (LAW et al., 2003), ECElG (*Elliptic curve El Gamal*) (HANKERSON; MENEZES; VANSTONE, 2003), ECDSA (*Elliptic Curve Digital Signature Algorithm*), ECSS (*Elliptic Curve Schnorr Signature*). Todos estes algoritmos possuem sua segurança na dificuldade de solução do ECDLP.

Os protocolos ECDH, ECMQV e ECElG são utilizados na troca de chaves simétricas. ECDH e ECElG são protocolos não-autenticados, enquanto o ECMQV é autenticado. ECDH e ECElG são baseados no protocolo de troca de chaves Diffie-Hellman.

Os protocolos ECDSA e ECSS são utilizados na assinatura digital. ECDSA é análogo ao DSA (HANKERSON; MENEZES; VANSTONE, 2003), e surgiu em resposta à solicitação do NIST (RIVEST et al., 1992) para comentários em sua proposta inicial do DSS (Digital Signature Standard) (JOHNSON; MENEZES; VANSTONE, 2001). O protocolo ECSS é uma variação do *Schnorr Signature* (SCHNORR, 1989).

Na Figura 1 são apresentadas as camadas presentes em alguns protocolos de criptografia baseados em curvas elípticas. As camadas representam as operações, sendo que as camadas superiores são calculadas pelas camadas inferiores.

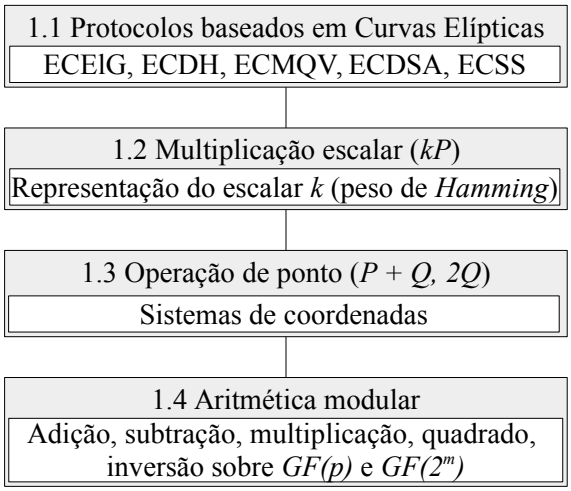

Figura 1: Hierarquia das operações em ECC

Este capítulo apresenta o funcionamento das operações em ECC, bem como a descrição dos principais algoritmos estudados para realização deste trabalho. Além disso, são discutidas as propostas de trabalhos relacionados, e apresentados os detalhes dos experimentos realizados nos capítulos seguintes.

O protocolo ECElG é detalhado na seção a seguir, a fim de demonstrar uma das formas de aplicação de curvas elípticas em um protocolo de criptografia.

2.1 El Gamal baseado em curvas elípticas (ECElG)

O protocolo de criptografia El Gamal foi introduzido por Gamal et al. (GAMAL, 1985). Este protocolo é normalmente definido sobre $\mathbb{Z}/p\mathbb{Z}$ para p primo, mas pode ser definido sobre qualquer grupo cíclico abeliano (tal como o grupo definido pelas curvas elípticas sobre corpos finitos).

O procedimento de cifração e decifração utilizando o protocolo análogo ao ElGamal baseado em curvas elípticas (ECElG) sobre $E(GF(p))$ (HANKERSON; MENEZES; VANSTONE, 2003) é descrito a seguir. Sejam Alice e Bob duas entidades. Bob deseja transmitir uma mensagem confidencial m para Alice por um canal inseguro. A mensagem m pode ser um texto comum ou uma chave simétrica. Para isso, Bob precisa conhecer a chave pública p_u de Alice, para então poder transmitir sua mensagem pelo canal com a garantia de que apenas quem possui a chave privada p_r conseguirá decifrar o conteúdo da informação transmitida. O ponto base (ou gerador) G pode ser escolhido arbitrariamente e deve, juntamente com o primo p, ser conhecido por ambas as entidades. O procedimento completo do ECElG é apresentado na Figura 2.

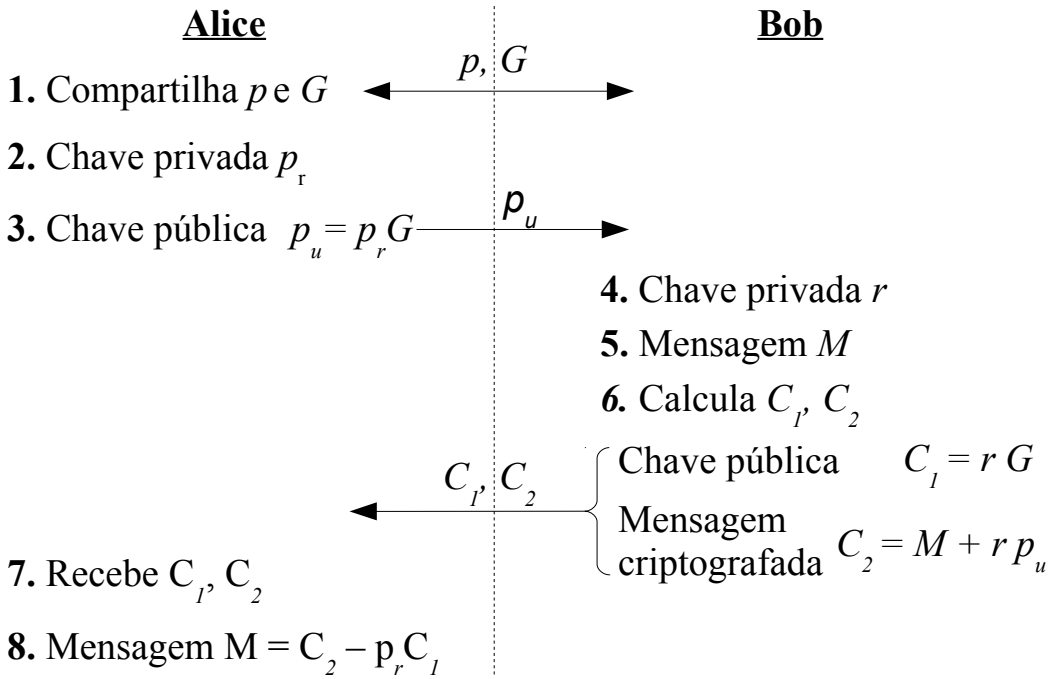

Figura 2: ECElG - Protocolo de criptografia ElGamal baseado em ECC

A descrição de cada passo de comunicação (Figura 2), juntamente com sua respectiva numeração são apresentadas a seguir:

1. Bob e Alice compartilham um número primo p e um ponto base G, tal que $G \in E(GF(p))$. Estes parâmetros são conhecidos *a priori* pelas entidades, e acordados após o *handshake*, no qual é estabelecida a curva elíptica utilizada.

2. Alice gera um inteiro aleatório p_r, tal que $p_r \in [1, p-1]$.

3. Alice calcula sua chave pública p_u, tal que $p_u = p_r G$, e a envia para Bob.

4. Bob gera um inteiro aleatório r, tal que $r \in [1, p-1]$.

5. Bob representa a mensagem m como um ponto $M \in E(GF(p))$.

6. Bob calcula C_1 e C_2 e os envia para Alice pelo canal:

   $C_1 = rG$

   $C_2 = M + rp_u$ (mascaramento da mensagem)

7. Alice recebe C_1 e C_2 representando o texto cifrado.

8. Alice recupera a mensagem enviada por Bob calculando $M = (C_2 - p_r C_1) = (M + r(p_r G)) - (rp_r G) = M$, pois o conjunto de pontos da curva $E(GF(p))$ é um grupo abeliano finito (ANEXO A.1).

9. Alice extrai m de M.

O protocolo ECElG (Figura 2) é composto pela multiplicação de um escalar por um ponto (kP) nos passos 3 (p_rG), 6 (rG, rp_u) e 8 (p_rC_1) e operações de ponto ($P-Q$ e $P+Q$) nos passos 6 ($M+rp_u$) e 8 ($C_2-p_rC_1$), onde $k \in \mathbb{Z}_p$ e $P, Q \in E$. O cálculo $P-Q$ consiste na adição do ponto P ao inverso de Q. A multiplicação escalar kP é a operação mais custosa, e se baseia no cálculo de sucessivas operações de ponto dependentes. As operações de ponto são realizadas utilizando aritmética modular ou em base polinomial/normal sobre um corpo finito (GF).

Na Figura 3 é detalhada a multiplicação escalar (kP), as operações de ponto ($P+Q, 2Q$) e a aritmética modular apresentadas em alto nível na Figura 1. Nesta figura são apresentadas as dependências entre as operações, e alguns algoritmos que podem ser utilizados no cálculo de cada operação. As sessões seguintes descrevem essas operações.

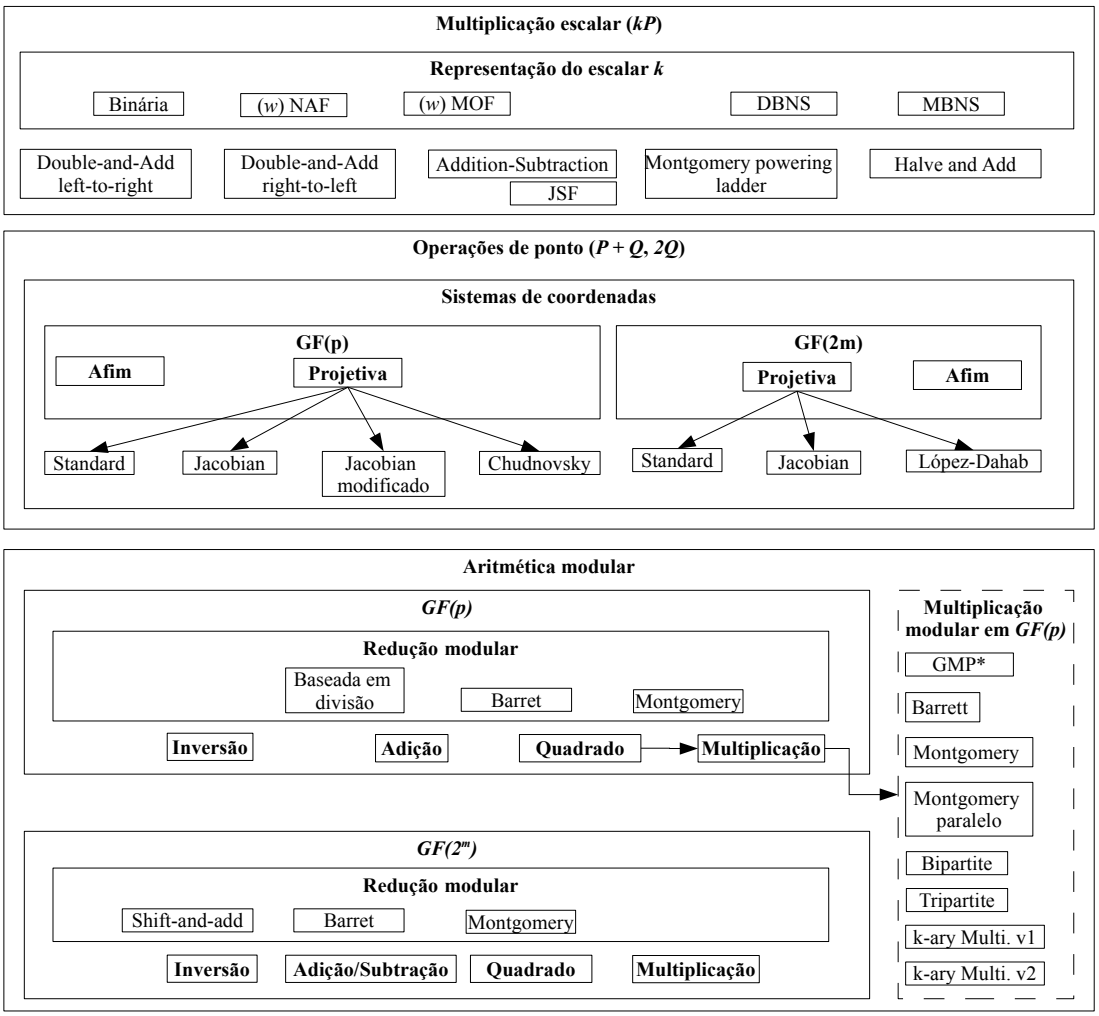

Figura 3: Multiplicação escalar para criptografia de curvas elípticas. * A multiplicação modular GMP é composta pelas operações de multiplicação e módulo disponíveis na biblioteca GMP.

2.2 Multiplicação escalar

Como já descrito no início deste capítulo, a multiplicação escalar (kP) é a operação principal presente em um protocolo de criptografia baseado em curvas elípticas, pois é nela que se encontra a segurança do mesmo. Esta operação é a mais custosa, e pode ser representada como repetidas adições de ponto [1] (STALLINGS, 2010):

$$k \times P = \underbrace{(P + P + ... + P)}_{k \text{ vezes}}$$

O peso de *Hamming* da representação binária de k é dado pela quantidade de bits diferentes de zero que ela possui. A representação de k interfere diretamente no custo da multiplicação escalar. Seja h o número de bits de k, o peso de *Hamming* da representação binária comum de k, é em média $(h/2)$ bits. A redução do peso de *Hamming* implica na redução do número de adições de ponto realizadas na multiplicação escalar *Double-and-Add* (Seção 2.2.1) e consequentemente uma melhora no tempo de computação (KARTHIKEYAN, 2012).

Foram propostas na literatura diversas formas de representação, a fim de reduzir o peso de *Hamming* do escalar k. Dentre elas, pode-se citar: *NAF*, *w-NAF*, *MOF*, *w-MOF*, *DBNS*, *MBNS*, *JSF*.

A representação escalar *Non-Adjacent Form* (*NAF*) (BOOTH, 1951), (REITWIESNER, 1960) é obtida a partir da recodificação do escalar k. A recodificação é realizada do dígito menos significante ao mais significante (*right-to-left*).

Algoritmo 1 Non-Adjacent Form (*NAF*)

```
1:  procedimento NAF(k)                          ▷ representação binária de k
2:      c ← k
3:      l ← 0
4:      enquanto c > 0 faça
5:          se c mod 2 = 1 então
6:              s[l] ← 2 − (c mod 4)
7:              c ← c − s[l]
8:          senão
9:              s[l] ← 0
10:         fim se
11:         c ← c ≫ 1                            ▷ deslocamento de 1 bit à direita
12:         l ← l + 1
13:     fim enquanto
14:     retorne s                                ▷ representação NAF de k
15: fim procedimento
```

[1] esta operação corresponde ao equivalente da exponenciação para o algoritmo de criptografia RSA.

O Algoritmo 1 apresenta a recodificação *NAF*. Esta recodificação reduz o peso de *Hamming* médio aproximado de k para $(h/3)$ bits, oferecendo uma melhora significativa no desempenho da multiplicação escalar (KARTHIKEYAN, 2012). Exemplo: seja $(101101101111)_2$ a representação binária de $(2927)_{10}$. O binário $(101101101111)_2$ pode ser representado em NAF como o ternário $(10{-}100{-}100{-}1000{-}1)_{NAF}$ (KARTHIKEYAN, 2012). O uso desta representação com o algoritmo *Double-and-Add* (Seção 2.2.1) resulta no algoritmo *Addition-Subtraction*, apresentado na Seção 2.2.2. O *NAF* possui uma variação denominada *w-NAF* (KOYAMA; TSURUOKA, 1993), que utiliza janelas de tamanho w. O peso de *Hamming* médio desta representação é $n/(w+1)$ (KARTHIKEYAN, 2012). A representação escalar *Mutual Opposite Form* (*MOF*) (OKEYA et al., 2004) é obtida a partir da recodificação do escalar k, de forma a possuir peso de *Hamming* médio aproximado de $(h/3)$ bits. Esta representação também possui uma variação que utiliza janelas de tamanho w, denominada *w-MOF*, o que reduz o peso de *Hamming* médio para $h/(w+1)$ (KARTHIKEYAN, 2012). As recodificações *MOF* e *w-MOF* podem ser utilizadas eficientemente em conjunto com o algoritmo de multiplicação escalar *Double-and-Add left-to-right* e *right-to-left*.

A representação DBNS (*Double-Base Number System*) é formada pela soma de potências de bases 2 e 3, ou na forma 2^b3^t, tal que b e t são inteiros não negativos. A alta redundância destes algoritmos permite diferentes representações para o mesmo escalar, oferecendo proteção contra ataques *side-channel* (KARTHIKEYAN, 2012). A representação *Multi-Base Number System* (MBNS) é uma generalização da *DBNS*, no entanto, a representação do escalar é formada por mais de duas bases (CHABRIER; TISSERAND, 2013).

O algoritmo de recodificação deve ser escolhido cuidadosamente, para utilizar eficientemente os recursos do sistema computacional subjacente. Nas recodificações baseadas em janela, quanto maior a janela, maior a quantidade de pontos pré-computados armazenados em memória. A escolha de uma recodificação *left-to-right* ou *right-to-left* pode interferir no desempenho de um determinado algoritmo de multiplicação escalar.

Diversos algoritmos que calculam a multiplicação escalar (kP) foram propostos na literatura e os principais são apresentados nas seções seguintes.

2.2.1 *Double-and-Add*

A adição e dobro de ponto são as principais operações do algoritmo de multiplicação escalar *Double-and-Add* (a.k.a. *Binary*) (AHMADI; HANKERSON; RODRÍGUEZ-HENRÍQUEZ, 2008). Este algoritmo pode ser estruturado como *left-to-right* ou *right-to-left*, oferecendo em geral a mesma eficiência em ambos os sentidos (OKEYA et al., 2004). Utilizando a representação binária comum (sem recodificação), este algoritmo calcula em média $(h-1)$ dobros e $(h-1)/2$ adições de ponto (AHMADI; HANKERSON; RODRÍGUEZ-

HENRÍQUEZ, 2008; KARTHIKEYAN, 2012).

O algoritmo *Double-and-Add right-to-left* (Algoritmo 2) armazena temporariamente 2^iP em S, que é adicionado a Q quando o i-ésimo bit é igual a 1 (OKEYA et al., 2004). Este algoritmo pode ser ajustado para utilizar as representações *w-NAF*, *w-MOF* mais eficientemente que a versão *left-to-right* (OKEYA et al., 2004).

Algoritmo 2 Multiplicação escalar *Double-and-Add right-to-left*

```
1:  procedimento DoubleAddRL(k, P)        ▷ representação binária de k e ponto P
2:      Q ← O                                              ▷ O é o ponto no finito
3:      S ← P
4:      para i de 0 até h − 1 passo 1 faça                 ▷ h é o número de bits de k
5:          se k_i = 1 então
6:              Q ← Q + S                                  ▷ Q + S : adição de ponto
7:          fim se
8:          S ← 2S                                         ▷ 2S : dobro de ponto
9:      fim para
10:     retorne Q                                          ▷ Q = kP
11: fim procedimento
```

O algoritmo *Double-and-Add left-to-right* (Algoritmo 3), no entanto, realiza o cálculo da multiplicação escalar bit a bit, da esquerda para a direita. Se comparado com a versão *right-to-left* (Algoritmo 2), ele utiliza menos memória, pois não faz uso do registro auxiliar S.

Algoritmo 3 Multiplicação escalar *Double-and-Add left-to-right*

```
1:  procedimento DoubleAddLR(k, P)        ▷ representação binária de k e ponto P
2:      Q ← O                                              ▷ O é o ponto no finito
3:      para i de h − 1 até 0 passo −1 faça                ▷ h é o número de bits de k
4:          Q ← 2Q                                         ▷ 2Q : dobro de ponto
5:          se k_i = 1 então
6:              Q ← Q + P                                  ▷ Q + P : adição de ponto
7:          fim se
8:      fim para
9:      retorne Q                                          ▷ Q = kP
10: fim procedimento
```

A multiplicação escalar *Double-and-Add left-to-right* pode ser utilizada eficientemente em conjunto com os algoritmos de recodificação *MOF* ou *w-MOF* (calculados no sentido *right-to-left* ou *left-to-right*), pois seu cálculo pode ser realizado à medida que os bits do escalar são recodificados. Em um eventual uso da recodificação *w-NAF* (*right-to-left*), seria necessário o cálculo completo e armazenamento da representação escalar antes do início do estágio de avaliação. Portanto, o uso do *w-NAF* em conjunto com a multiplicação escalar *Double-and-Add left-to-right* não é vantajoso, quando implementado em dispositivos com memória limitada (OKEYA et al., 2004).

A fim de acelerar o cálculo da operação de multiplicação escalar, o algoritmo *Double-and-Add* pode ser modificado para utilizar uma tabela de pontos pré-computada, vantajosa quando o ponto gerador da curva é fixo. Sua pré-computação consiste de passos sequenciais que não podem ser paralelizados (AL-SOMANI; IBRAHIM, 2009).

No Algoritmo 4 é apresentada uma variação do algoritmo *Double-and-Add right-to-left* que faz uso de uma tabela pré-computada de pontos (com múltiplos do ponto P) no cálculo da multiplicação escalar. Este algoritmo realiza o cálculo prévio de 2^iP, tal que $0 \leq i < h$, sendo necessário apenas a adição de pontos pré-computados para o cálculo de kP.

Algoritmo 4 Multiplicação escalar *Double-and-Add right-to-left* com pré-computação

```
1: procedimento DoubleAddRLPrec(k, P) ▷ representação binária de k e ponto P
2:     Q ← O                                                    ▷ O é o ponto no finito
3:     s[] ← PREC(k, P)        ▷ vetor s pré-computado, com h múltiplos do ponto P
4:     para i de 0 até h − 1 passo 1 faça                       ▷ h é o número de bits de k
5:         se k_i = 1 então
6:             Q ← Q + s[i]                                     ▷ Q + s[i] : adição de ponto
7:         fim se
8:     fim para
9:     retorne Q                                                ▷ Q = kP
10: fim procedimento
```

O Algoritmo 4 pode ser integrado tanto nas versões *left-to-right* quanto *right-to-left* do algoritmo *Double-and-Add*. No entanto, em ambas as implementações, o custo deste algoritmo é próximo ao do *right-to-left*, devido à pré-computação de um vetor de pontos.

2.2.2 Addition-Subtraction

O algoritmo de multiplicação escalar *Addition-Subtraction* (Algoritmo 5) é uma variação do *Double-and-Add* (Algoritmo 3). Este algoritmo utiliza *NAF* (Algoritmo 1) como forma de representação do escalar, reduzindo o peso de *hamming* do mesmo. Este algoritmo calcula em média h dobros e $h/3$ adições de ponto (KARTHIKEYAN, 2012).

Pode-se observar que este algoritmo realiza as operações de adição, subtração e dobro de ponto. A subtração $Q - P$ é calculada como a soma de Q ao inverso de P. Como pode ser observado na Seção 2.3, o inverso de um ponto é calculado facilmente, sem custo adicional.

Algoritmo 5 Multiplicação escalar *Addition-Subtraction*

procedimento ADDSUB(k, P) ▷ representação binária de k e ponto P
  $s[] \leftarrow NAF(k)$ ▷ NAF (Algoritmo 1) de k é armazenado em s
  $Q \leftarrow O$
  para i **de** $h-1$ **até** 0 **passo** -1 **faça** ▷ h é o tamanho de s
    $Q \leftarrow 2Q$ ▷ $2Q$: dobro de ponto
    se $s_i = 1$ **então**
      $Q \leftarrow Q + P$ ▷ $Q + P$: adição de ponto
    senão se $s_i = -1$ **então**
      $Q \leftarrow Q - P$ ▷ $Q - P$: adição de ponto
    fim se
  fim para
  retorne Q ▷ $Q = kP$
fim procedimento

2.2.3 Montgomery Ladder

Montgomery (MONTGOMERY, 1987) propôs em 1987 o algoritmo de multiplicação escalar *Montgomery Ladder*, baseado no *Double-and-Add* (Seção 2.2.1). Este algoritmo calcula múltiplos de pontos para um tipo especial de curva elíptica sobre corpos finitos primos. Este algoritmo é eficiente contra ataques de *timing* e SPA (*Simple Power Analysis*) (DOMINGUEZ-OVIEDO; HASAN, 2011).

2.2.4 Halve-and-Add

O algoritmo de multiplicação escalar *Halve-and-Add* foi proposto independemente por Knudsen (KNUDSEN, 1999) e Schroeppel (SCHROEPPEL, 2000) para curvas binárias (sobre $GF(2^m)$) na forma $y^2 + xy = x^3 + ax^2 + b$ (TAVERNE et al., 2011).

Os diferentes sistemas de coordenadas disponíveis para o cálculo das operações de adição e dobro de ponto, são apresentados na Seção 2.3.

2.3 Operações de ponto

Os algoritmos de multiplicação escalar (kP) (Seção 2.2) baseados no *Double-and-Add* (Seção 2.2.1), são compostos pelas operações de adição ($P+Q$) e dobro ($2Q$) de ponto, tal que estas são as principais operações realizadas dentro de um protocolo de criptografia baseado em curvas elípticas. Na Figura 4 é apresentada a representação geométrica destas operações em uma curva elíptica (em $\mathbb{R}$).

Sejam P e Q pontos da curva elíptica E (Figura 4). A adição $R_A = P + Q$ pode ser definida como a reflexão (linha tracejada) da intersecção entre E e a reta que passa por P e Q. Quando $P = Q$, o cálculo de $R_D = P + Q = 2P = 2Q$ é dado pela reflexão

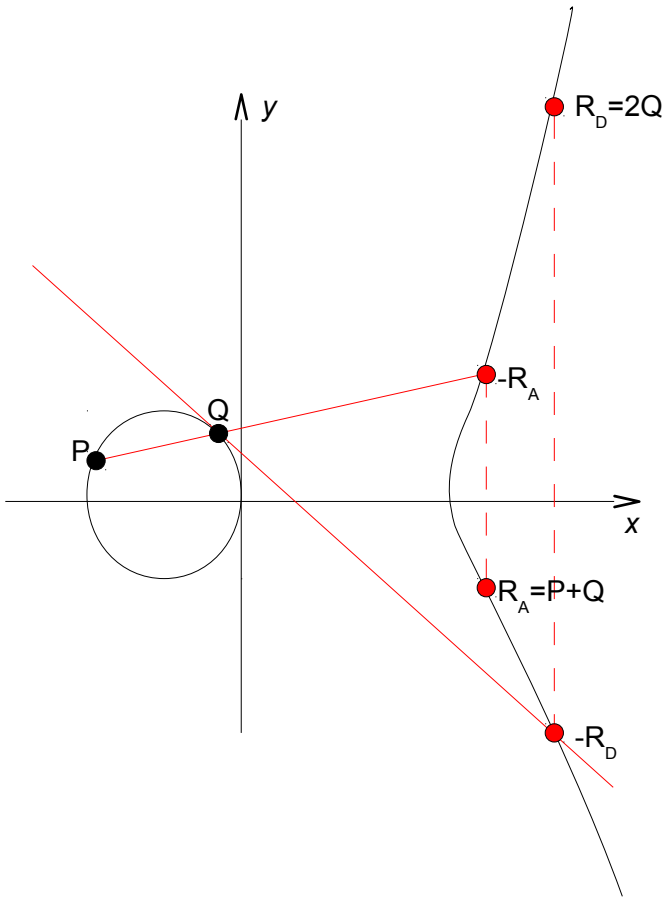

Figura 4: Representação geométrica da adição e dobro de pontos em uma curva elíptica

da intersecção entre a curva E e a reta tangente à curva E no ponto Q. Nesta figura, observa-se também que o inverso de R_A é representado por $-R_A$.

Em criptografia, no entanto, as operações de adição e dobro de ponto são realizadas com as coordenadas e coeficientes da curva pertencentes a um corpo finito.

Segundo Stallings (STALLINGS, 2010), as curvas primas sobre $GF(p)$ e binárias sobre $GF(2^m)$ (ANEXO A.3) são aplicadas à criptografia. Diversas curvas sobre $GF(p)$ e sobre $GF(2^m)$ foram padronizadas pelo NIST (*National Institute of Standards and Technology*) para uso em assinatura digital (GALLAGHER; FURLANI, 2009).

$$E(L) = \{(x, y) \in L \times L : y^2 + axy + by = x^3 + cx^2 + dx + e\} \cup \{O\} \tag{2.1}$$

O é o ponto da linha no infinito que satisfaz a equação de *Weierstrass*.

Seja q um inteiro, tal que $q = p$ ou $q = 2^m$. Uma curva elíptica E sobre $GF(q)$ é definida pela equação de *Weierstrass* como $E/GF(q)$, onde os coeficientes $a, b, c, d, e \in GF(q)$, e $\Delta \neq 0$ é a discriminante de E. Como E está definido sobre $GF(q)$, então também está definido sobre qualquer corpo de extensão de $GF(q)$. Seja L um corpo de extensão de

$GF(q)$, e O o ponto no infinito, tal que $O \in E(L)$, o conjunto de pontos racionais em E sobre L é dado pelo conjunto de coordenadas $x, y \in L$ que satisfazem a Equação 2.1 e o ponto no infinito O.

A curva E sobre $GF(p)$ pode ser representada na forma *short Weierstrass* (Equação 2.2) (HANKERSON; MENEZES; VANSTONE, 2003), tal que $p \neq 2, 3$ e $a, b \in GF(p)$.

$$y^2 = x^3 + ax + b \tag{2.2}$$

Sejam a, b coeficientes da curva E, que satisfazem a Equação 2.2. A curva $E(a, b)$ define um grupo sobre $GF(p)$, tal que $4a^3 + 27b^2 \neq 0$. A discriminante desta curva é $\Delta = -16(4a^3 + 27b^2)$ (HANKERSON; MENEZES; VANSTONE, 2003).

Nas curvas $E(GF(p))$, as variáveis e coeficientes possuem valor no intervalo $[0, p-1]$, e os cálculos são realizados módulo p (ANEXO A.3.1).

Na Figura 5 é apresentado um exemplo de curva elíptica definida sobre o corpo finito primo $GF(29)$. O conjunto de pontos da curva é formado pelas coordenadas x, y que resolvem a Equação 2.2, tal que $x, y \in GF(29)$.

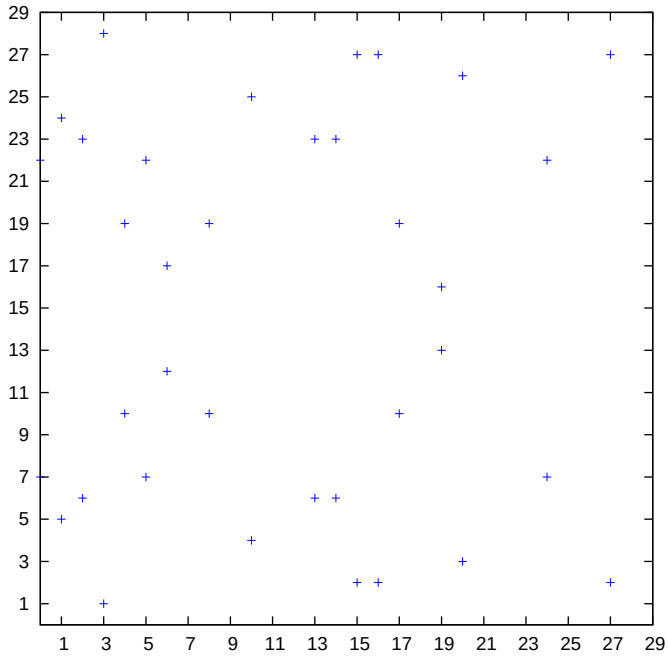

Figura 5: Pontos da curva $E(GF(29)) : y^2 = x^3 + 4x + 20$

A curva E sobre $GF(2^m)$ pode ser representada na forma *short Weierstrass* (Equação 2.3) (HANKERSON; MENEZES; VANSTONE, 2003), tal que $a \neq 0$ e E é *supersingu-*

lar[2].

$$y^2 + xy = x^3 + ax^2 + b \tag{2.3}$$

onde $a, b \in GF(2^m)$.

Sejam a, b os coeficientes da curva E, que satisfazem a Equação 2.3. A curva $E(a, b)$ define um grupo sobre $GF(2^m)$, tal que $b \neq 0$ (ANEXO A.3.2). A discriminante desta curva é $\Delta = b$ (HANKERSON; MENEZES; VANSTONE, 2003).

As curvas elípticas são definidas sobre um grupo abeliano finito com relação à operação de adição $(E, +)$. Seja O um ponto pertencente à curva elíptica denominado ponto no infinito. Uma curva elíptica possui as seguintes propriedades, com relação à operação de adição de ponto $\{E, +\}$:

- Fechamento: se $P, Q \in E$ então $P + Q \in E$.
- Associatividade: se $P, Q, R \in E$ então $(P + Q) + R = P + (Q + R)$.
- Identidade: existe um elemento O denominado ponto no infinito, tal que $O \in E$, e para todo ponto $P \in E$ temos $P + O = O + P = P$ e $-O = O$.
- Inverso: para todo ponto $P = (x, y) \in E$ existe um ponto $-P \in E$, inverso de P, tal que $P + (-P) = (-P) + P = O$.

O conjunto de operações e a representação dos pontos (inclusive o inverso e identidade) variam de acordo com o sistema de coordenadas utilizado. O sistema de coordenadas pode ser Afim ou pertencer à classe dos sistemas de coordenadas projetivas.

As operações de adição e dobro de ponto na forma Afim (x, y) em $GF(p)$ e $GF(2^m)$ realizam os cálculos de inversão, adição, subtração, multiplicação e quadrado modulares. Os sistemas de coordenadas projetivas não realizam nenhuma inversão modular nas operações de ponto. Estes sistemas realizam mais operações modulares que o sistema de coordenadas Afim. Pode ser mais vantajoso utilizar coordenadas projetivas para representar os pontos da curva, caso a inversão em GF seja mais custosa que as multiplicações e quadrados adicionais (HANKERSON; MENEZES; VANSTONE, 2003; ANSARI; HASAN, 2008).

As operações com coordenadas mistas permitem a adição de pontos em diferentes sistemas de coordenadas, e a representação do resultado em outro (COHEN et al., 2006).

Os sistemas de coordenadas podem ser definidos com aritmética sobre $GF(p)$ ou $GF(2^m)$. Alguns destes são apresentados a seguir.

[2] Uma curva E é supersingular se esta não possui singularidades, ou seja, pode-se traçar uma reta tangente a todos os pontos de E, de modo que $\Delta \neq 0$. Caso contrário, E é singular ou ordinária.

2.3.1 Curva elíptica E sobre $GF(2^m)$

Os sistemas de coordenadas definidos sobre $GF(2^m)$, definem que nas operações de adição e dobro de ponto, todos os cálculos aritméticos são realizados módulo um polinômio irredutível $f(x)$.

O sistema de coordenadas pode ser *Afim* ($\mathcal{A}$) ou pertencer à classe dos sistemas de coordenadas projetivas: *Standard* ($\mathcal{P}$), *Jacobian* ($\mathcal{J}$), *López-Dahab* ($\mathcal{LD}$) (COHEN et al., 2006). No ANEXO B.2-Tabela 45 é apresentado o custo das operações de ponto destes sistemas de coordenadas.

Nas seções seguintes são descritas as operações de adição e dobro nos sistemas de coordenadas $\mathcal{A}$, $\mathcal{P}$, $\mathcal{J}$, $\mathcal{LD}$, respectivamente.

2.3.1.1 Sistema de coordenadas *Afim* ($\mathcal{A}$)

A equação de uma curva elíptica ordinária sobre $GF(2^m)$ em sua forma afim é: $E : y^2 + xy = x^3 + ax^2 + b$. Os sistemas de coordenadas *Afim* ($\mathcal{A}$) aplicam as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto:

1. Adição de ponto: Seja $P = (x_1, y_1) \in E(GF(2^m))$ e $Q = (x_2, y_2) \in E(GF(2^m))$, onde $P \neq \pm Q$, então $P + Q = (x_3, y_3)$, onde

$$x_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)^2 + x_1 + x_2 \text{ e } y_3 = \left(\frac{y_1 + y_2}{x_1 + x_2}\right)(x_1 + x_3) + y_1 + c \tag{2.4}$$

2. Dobro de ponto: Seja $P = (x_1, y_1) \in E(GF(2^m))$, onde $P \neq -P$, então $2P = (x_3, y_3)$, onde

$$x_3 = \left(\frac{x_1^2 + a}{c}\right)^2 \text{ e } y_3 = \left(\frac{x_1^2 + a}{c}\right)(x_1 + x_3) + y_1 + c \tag{2.5}$$

2.3.1.2 Sistemas de coordenadas projetivas *Standard* ($\mathcal{P}$)

No sistema de coordenadas *Standard* ($\mathcal{P}$), o ponto projetivo $(X_1 : Y_1 : Z_1)$, $Z_1 \neq 0$ corresponde ao afim $(X_1/Z_1, Y_1/Z_1)$. A equação projetiva da curva elíptica neste sistema é: $y^2z + xyz = x^3 + cx^2z + ez^3$. Este sistema aplica as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto (HANKERSON; MENEZES; VANSTONE, 2003):

1. Adição de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ tal que $P \neq \pm Q$, então $P + Q = (X_3, Y_3, Z_3)$, é dado por:

$$\begin{cases} X_3 = BE, \\ Y_3 = C(AX_1 + Y_1B)Z_2 + (A + B)E, \\ Z_3 = B^3D \end{cases} \tag{2.6}$$

tal que $A = Y_1Z_2 + Z_1Y_2, B = X_1Z_2 + Z_1X_2, C = B^2, D = Z_1Z_2, E = (A^2 + AB + cC)D + BC$

2. Dobro de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, então $2P = (X_3 : Y_3 : Z_3)$ é dado por:

$$\begin{cases} X_3 = CE, \\ Y_3 = (B + C)E + A^2C, \\ Z_3 = CD \end{cases} \tag{2.7}$$

tal que $A = X_1^2, B = A + Y_1Z_1, C = X_1Z_1, D = C^2, E = (B^2 + BC + cD)$

O elemento identidade O corresponde a $(0 : 1 : 0)$, e o inverso de $P = (X : Y : Z)$ é $-P = (X : X + Y : Z)$.

2.3.1.3 Sistemas de coordenadas projetivas *Jacobian* ($\mathcal{J}$)

No sistema de coordenadas *Jacobian* ($\mathcal{J}$), o ponto projetivo $(X_1 : Y_1 : Z_1)$, $Z_1 \neq 0$ corresponde ao afim $(X_1/Z_1^2, Y/Z_1^3)$. A equação projetiva da curva elíptica neste sistema é: $y^2 + xyz = x^3 + cx^2z^2 + ez^6$. Este sistema aplica as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto (HANKERSON; MENEZES; VANSTONE, 2003):

1. Adição de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ tal que $P \neq \pm Q$, então $P + Q = (X_3, Y_3, Z_3)$, é dado por:

$$\begin{cases} Z_3 = GZ_2, \\ X_3 = A_2Z_3^2 + FI + E^3, \\ Y_3 = IX_3 + G^2H \end{cases} \tag{2.8}$$

tal que $A = X_1Z_2^2, B = X_2Z_1^2, C = Y1Z_2^3, D = Y_2Z_1^3, E = A + B, F = C + D, G = EZ_1, H = FX_2 + GY_2, I = F + Z_3$

2. Dobro de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, então $2P = (X_3 : Y_3 : Z_3)$ é dado por:

$$\begin{cases} Z_3 = X_1 C, \\ X_3 = B + eC^4, \\ Y_3 = BZ_3 + (A + Y_1 Z_1 + Z_3)X_3 \end{cases} \tag{2.9}$$

tal que $A = X_1^2, B = A^2, C = Z_1^2$

O elemento identidade O corresponde a $(1 : 1 : 0)$, e o inverso de $P = (X : Y : Z)$ é $-P = (X : XZ + Y : Z)$.

2.3.1.4 Sistemas de coordenadas projetivas *López-Dahab* ($\mathcal{LD}$)

No sistema de coordenadas *López-Dahab* ($\mathcal{LD}$), o ponto projetivo $(X : Y : Z)$, $Z \neq 0$ corresponde ao afim $(X/Z, Y/Z^2)$. A equação projetiva da curva elíptica neste sistema é: $y^2 + xyz = x^3 z + cx^2 z^2 + ez^4$. Este sistema aplica as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto (HANKERSON; MENEZES; VANSTONE, 2003):

1. Adição de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ tal que $P \neq \pm Q$, então $P + Q = (X_3, Y_3, Z_3)$, é dado por:

$$\begin{cases} Z_3 = FZ_1 Z_2, \\ X_3 = A(H + D) + B(C + G), \\ Y_3 = (AJ + FG)F + (J + Z_3)X_3 \end{cases} \tag{2.10}$$

   tal que $A = X_1 Z_2, B = X_2 Z_1, C = A^2, D = B^2, E = A + B, F = C + D, G = Y_1 Z_2^2, H = Y_2 Z_1^2, I = G + H, J = IE$

2. Dobro de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, então $2P = (X_3 : Y_3 : Z_3)$ é dado por:

$$\begin{cases} Z_3 = AC, \\ X_3 = C^2 + B, \\ Y_3 = (Y_1^2 + cZ_3 + B)X_3 + Z_3 B \end{cases} \tag{2.11}$$

tal que $A = Z_1^2, B = eA^2, C = X_1^2$

O elemento identidade O corresponde a $(1 : 0 : 0)$, e o inverso de $P = (X : Y : Z)$ é $-P = (X : XZ + Y : Z)$.

2.3.2 Curva elíptica E sobre $GF(p)$

Os sistemas de coordenadas sobre $GF(p)$, definem que nas operações de adição e dobro de ponto, todos os cálculos aritméticos são realizados módulo um número primo p.

O sistema de coordenadas pode ser *Afim* ($\mathcal{A}$) ou pertencer à classe dos sistemas de coordenadas projetivas: *Standard* ($\mathcal{P}$), *Jacobian* ($\mathcal{J}$), *Chudnovsky* ($\mathcal{J}^c$) e *Jacobian* modificadas ($\mathcal{J}^m$) (HANKERSON; MENEZES; VANSTONE, 2003),(COHEN et al., 2006). Na Tabela 1 é apresentado o custo das operações de ponto destes sistemas de coordenadas.

Tabela 1: Operações necessárias para adição e dobro em $GF(p)$.

Dobro		Adição		Adição mista	
Operação	Custo	Operação	Custo	Operação	Custo
$2\mathcal{P}$	$7M+5S$	$\mathcal{P}+\mathcal{P}$	$12M+2S$	$\mathcal{J}^m+\mathcal{J}^c=\mathcal{J}^m$	$12M+5S$
$2\mathcal{J}^c$	$5M+6S$	$\mathcal{J}^c+\mathcal{J}^c=J^m$	$11M+4S$	$\mathcal{J}+\mathcal{J}^c=\mathcal{J}^m$	$12M+5S$
$2\mathcal{J}$	$4M+6S$	$\mathcal{J}+\mathcal{J}$	$12M+4S$	$\mathcal{J}^c+\mathcal{J}$	$11M+3S$
$2\mathcal{J}^m$	$4M+4S$	$\mathcal{J}^m+\mathcal{J}^m$	$13M+6S$	$\mathcal{J}+\mathcal{A}=\mathcal{J}^m$	$9M+5S$
$2\mathcal{A}$	$I+2M+2S$	$\mathcal{A}+\mathcal{A}$	$I+2M+S$	$\mathcal{J}^m+\mathcal{A}=\mathcal{J}^m$	$9M+5S$
$2\mathcal{J}^m=J^c$	$4M+5S$	$\mathcal{J}^c+\mathcal{J}^c$	$11M+3S$	$\mathcal{J}^c+\mathcal{A}=\mathcal{J}^m$	$8M+4S$
$2\mathcal{A}=J^c$	$3M+5S$	$J^c+\mathcal{J}^c=J$	$10M+2S$	$\mathcal{J}^c+\mathcal{A}=\mathcal{J}^c$	$8M+3S$
$2\mathcal{J}^m=J$	$3M+4S$	$\mathcal{A}+\mathcal{A}=\mathcal{J}^m$	$5M+4S$	$\mathcal{J}+\mathcal{A}=\mathcal{J}$	$8M+3S$
$2\mathcal{A}=J^m$	$3M+4S$	$\mathcal{A}+\mathcal{A}=\mathcal{J}^c$	$5M+3S$	$\mathcal{J}^m+\mathcal{A}=\mathcal{J}$	$8M+3S$
$2\mathcal{A}=J$	$2M+4S$	-	-	-	-

Fonte: (COHEN et al., 2006)

O custo das operações de ponto para as curvas com coeficiente $a=-3$ é apresentado no ANEXO B.1-Tabela 44.

Nas seções seguintes são descritas as operações de adição e dobro nos sistemas de coordenadas $\mathcal{A}$, $\mathcal{P}$, $\mathcal{J}$, $\mathcal{J}^c$ e $\mathcal{J}^m$.

2.3.2.1 Sistema de coordenadas *Afim* ($\mathcal{A}$)

A equação de uma curva elíptica sobre $GF(p)$ em sua forma afim é: $E: y^2 = x^3+dx+e$. Os sistemas de coordenadas *Afim* ($\mathcal{A}$) aplicam as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto:

1. Adição de ponto: Seja $P=(x_1,y_1)\in E(GF(p))$ e $Q=(x_2,y_2)\in E(GF(p))$, onde $P\neq\pm Q$, então $P+Q=(x_3,y_3)$, onde

$$x_3=\left(\frac{y_2-y_1}{x_2-x_1}\right)^2-x_1-x_2 \text{ e } y_3=\left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1-x_3)-y_1 \tag{2.12}$$

2. Dobro de ponto: Seja $P = (x_1, y_1) \in E(GF(p))$, onde $P \neq -P$, então $2P = (x_3, y_3), onde$

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ e } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{2.13}$$

2.3.2.2 Sistemas de coordenadas projetivas *Standard* ($\mathcal{P}$)

No sistema de coordenadas *Standard* ($\mathcal{P}$), o ponto projetivo $(X_1 : Y_1 : Z_1)$, $Z_1 \neq 0$ corresponde ao afim $(X_1/Z_1, Y_1/Z_1)$. A equação projetiva da curva elíptica neste sistema é: $E : y^2z = x^3 + dxz^2 + ez^3$. Este sistema aplica as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto (HANKERSON; MENEZES; VANSTONE, 2003), (COHEN et al., 2006):

1. Adição de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ tal que $P \neq \pm Q$, então $P + Q = (X_3, Y_3, Z_3)$, é dado por:

$$\begin{cases} X_3 = BC, \\ Y_3 = A(B^2X_1Z_2 - C) - B^3Y_1Z_2, \\ Z_3 = B^3Z_1Z_2 \end{cases} \tag{2.14}$$

   tal que $A = Y_2Z_1 - Y_1Z_2, B = X_2Z_1 - X_1Z_2, C = A^2Z_1Z_2 - B^3 - 2B^2X_1Z_2$

2. Dobro de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, então $2P = (X_3 : Y_3 : Z_3)$ é dado por:

$$\begin{cases} X_3 = 2BD, \\ Y_3 = A(4C - D) - 8Y_1^2B^2, \\ Z_3 = 8B^3 \end{cases} \tag{2.15}$$

   tal que $A = dZ_1^2 + 3X_1^2, B = Y_1Z_1, C = X_1Y_1B, D = A^2 - 8C$

O elemento identidade O corresponde a $(0 : 1 : 0)$, e o inverso de $P = (X : Y : Z)$ é $-P = (X : -Y : Z)$.

2.3.2.3 Sistemas de coordenadas projetivas *Jacobian* ($\mathcal{J}$)

No sistema de coordenadas *Jacobian* ($\mathcal{J}$), o ponto projetivo $(X_1 : Y_1 : Z_1)$, $Z_1 \neq 0$ corresponde ao afim $(X_1/Z_1^2, Y_1/Z_1^3)$. A equação projetiva da curva elíptica neste sistema é: $E : y^2 = x^3 + dxz^4 + ez^6$. Este sistema aplica as seguintes fórmulas algébricas no cálculo das operações de adição e dobro de ponto (HANKERSON; MENEZES; VANSTONE, 2003), (COHEN et al., 2006):

1. Adição de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$ tal que $P \neq \pm Q$, então $P + Q = (X_3, Y_3, Z_3)$, é dado por:

$$\begin{cases} X_3 = -E^3 - 2AE^2 + F^2, \\ Y_3 = -CE^3 + F(AE^2 - X_3), \\ Z_3 = Z_1 Z_2 E \end{cases} \tag{2.16}$$

   tal que $A = X_1 Z_2^2, B = X_2 Z_1^2, C = Y_1 Z_2^3, D = Y_2 Z_1^3, E = B - A, F = D - C$

2. Dobro de ponto: Seja $P = (X_1 : Y_1 : Z_1)$, então $2P = (X_3 : Y_3 : Z_3)$ é dado por:

$$\begin{cases} X_3 = -2A + B^2, \\ Y_3 = -8Y_1^4 + B(A - X_3), \\ Z_3 = 2Y_1 Z_1 \end{cases} \tag{2.17}$$

   tal que $A = 4X_1 Y_1^2, B = 3X_1^2 + dZ_1^4$

O elemento identidade O corresponde a $(1 : 1 : 0)$, e o inverso de $P = (X : Y : Z)$ é $-P = (X : -Y : Z)$.

Nas coordenadas *Jacobian*, as operações de dobro são mais rápidas e as de adição mais lentas que nas coordenadas *Standard*. Um ponto P pode ser representado como uma quíntupla $(X_1, Y_1, Z_1, Z_1^2, z_1^3)$. No sistema de coordenadas *Jacobian Chudnovsky* ($\mathcal{J}^c$), O ponto de *Jacobian* $(X : Y : Z)$, é representado como $(X : Y : Z : Z^2 : Z^3)$.

No sistema de coordenadas *Jacobian* modificado ($\mathcal{J}^m$), as coordenadas são baseadas nas coordenadas *Jacobian*. A representação do ponto P é o quádruplo (X_1, Y_1, Z_1, dZ_1^4), e as fórmulas são essencialmente as mesmas (COHEN et al., 2006).

As operações aritméticas realizadas nos sistemas de coordenadas sobre $GF(p)$ são modulares. Algoritmos eficientes para cálculo modular utilizados em aplicações criptográficas são apresentados na seção a seguir.

2.4 Aritmética modular

As operações realizadas sobre $GF(p)$ e $GF(2^m)$ são modulares, devido à propriedade de fechamento do corpo finito, sob o qual uma curva elíptica é definida. A aritmética modular sobre $GF(p)$ é módulo um inteiro primo p, enquanto a aritmética sobre $GF(2^m)$ é realizada módulo um polinômio irredutível $f(x)$ (ANEXO A).

Dados $C, M \in \mathbb{Z}$ tal que $M < C$, um algoritmo de redução modular calcula $Z = C \bmod M$, i.e. o resto Z da divisão de C por M (GIORGI; IMBERT; IZARD, 2013), tal que:

$$Z = C - \left\lfloor \frac{C}{M} \right\rfloor M \tag{2.18}$$

O desempenho da aritmética modular é fundamental para a eficiência dos protocolos baseados em curvas elípticas. Porém, o custo para o cálculo da redução modular é elevado devido à divisão $\lfloor C/M \rfloor$. Diversos algoritmos foram propostos para o cálculo de operações aritméticas modulares sobre $GF(p)$ e $GF(2^m)$ de forma a minimizar esse custo. A seguir são descritos os principais algoritmos utilizados no cálculo modular em aplicações criptográficas sobre $GF(p)$.

2.4.1 Redução de Montgomery

O algoritmo de redução de Montgomery (MONTGOMERY, 1985; AFREEN; MEHROTRA, 2011) é um algoritmo de redução modular extensivamente utilizado em criptografia de chave pública. O algoritmo recebe um inteiro C na forma de resíduo M de Montgomery e retorna o resto modular também na forma de resíduo de Montgomery. Sejam $A = Xr^n \bmod M$, $B = Yr^n \bmod M$ e $C = AB$. O algoritmo calcula $Cr^{-n} \bmod M$ resultando em $XYr^n \bmod M$, tal que r^n é uma raiz relativamente prima a M, $r = 2^w$, w é normalmente o tamanho da palavra do processador e n é o tamanho de M (em dígitos). A redução módulo r^n é realizada mais eficientemente a partir de alguns deslocamentos (*shifts*) (KOC, 1994) (GIORGI; IMBERT; IZARD, 2013). O uso deste algoritmo é vantajoso quando várias operações de mesmo módulo são realizadas, pois as operações de pré-processamento, como a conversão para forma de resíduo M e cálculo da constante de Montgomery, consomem bastante tempo (KOC, 1994). A constante de Montgomery pré-computada é: $\mu = -1/M \bmod r^n$.

Algoritmo 6 Redução de Montgomery

1: **procedimento** MONTGOMERY(C) ▷ Calcula $Cr^{-n} \bmod M$
2: ▷ $r^{n-1} < M < r^n$,mdc$(M, r^n) = 1, 0 \leq C \leq M^2$
3: $m \leftarrow (C \bmod r^n)\mu \bmod r^n$
4: $Z \leftarrow (C + mM)/r^n$
5: **se** $Z \geq M$ **então**
6: $Z \leftarrow Z - M$
7: **fim se**
8: **retorne** Z
9: **fim procedimento**

2.4.2 Redução de Barrett

O algoritmo de redução de *Barrett* (BARRETT, 1987; BROWN et al., 2001) é uma alternativa ao uso da redução de Montgomery pois, de modo similar, realiza divisão módulo uma potência de 2 ($r = 2^w$), ao invés de módulo M, podendo ser calculada mais eficientemente a partir de alguns deslocamentos (*shifts*) (GIORGI; IMBERT; IZARD, 2013). Seja M o módulo primo. Este algoritmo calcula XY mod M, tal que $X, Y \in \mathbb{Z}$. Ele pré-computa uma constante ν para aproximar o quociente Q com apenas uma multiplicação, conforme a Equação 2.19.

$$Q = \left\lfloor \left\lfloor \frac{C}{r^{n-1}} \right\rfloor \nu / r^{n+1} \right\rfloor \tag{2.19}$$

A constante pré-computada é: $\nu = \lfloor r^{2n}/M \rfloor$, que permite o cálculo eficiente da aproximação Q de $\lfloor C/M \rfloor$. O resto completamente reduzido pode ser obtido calculando $C - QM$, seguido de no máximo duas subtrações (BARRETT, 1987).

Algoritmo 7 Redução de Barrett

1: **procedimento** BARRETT(C) ▷ Calcula C mod M
2: ▷ $0 \leq C < M^2, r^{n-1} < M < r^n, 3 < r$
3: $Q \leftarrow \lfloor C\nu / r^{n-1} \rfloor / r^{n+1}$
4: $Z \leftarrow C - QM$ ▷ $0 \leq Z < 3M$
5: **enquanto** $Z \geq M$ **faça**
6: $Z \leftarrow Z - M$
7: **fim enquanto**
8: **retorne** Z
9: **fim procedimento**

2.5 Multiplicação modular paralela

Diversos algoritmos de multiplicação modular foram propostos e avaliados em arquiteturas paralelas de memória compartilhada. Alguns algoritmos são implementações paralelas da multiplicação de Montgomery, tais como o Bipartite (KAIHARA; TAKAGI, 2008), Tripartite (SAKIYAMA et al., 2011), k-ary Multiparte v1 e v2 (GIORGI; IMBERT; IZARD, 2013) e Montgomery Paralelo (BAKTIR; SAVAS, 2013) e (GIORGI; IMBERT; IZARD, 2013). Estes algoritmos realizam a conversão dos operandos para a forma de resíduo de Montgomery com uma raiz r^n ou r^t, tal que n é o tamanho de M (em palavras) e $0 \leq t \leq n$. Giorgi et al. também propuseram o paralelismo da multiplicação de inteiros presente na multiplicação modular de Barrett, de modo semelhante ao que foi feito para paralelizar a multiplicação de Montgomery. O algoritmo *RNS* (*Residue Number System*), no entanto, não é baseado no de Montgomery, e permite o paralelismo das operações

de adição e multiplicação. Este algoritmo faz uso de uma representação de resíduo de elementos de corpos finitos (ANTÃO; BAJARD; SOUSA, 2011).

2.5.1 Multiplicação modular de Montgomery paralela

A multiplicação modular de Montgomery (MONTGOMERY, 1985) calcula a multiplicação modular entre inteiros de multi precisão. Baktir et al. (BAKTIR; SAVAS, 2013) e Giorgi et al. (GIORGI; IMBERT; IZARD, 2013) propuseram independemente o paralelismo na multiplicação entre inteiros de multi precisão, empregado no algoritmo de multiplicação modular de Montgomery. O pseudocódigo deste algoritmo proposto por Baktir et al. é apresentado no Algoritmo 8. Neste algoritmo, o multiplicando é dividido em k partes, e o produto de cada parte pelo multiplicador é calculado em paralelo (Algoritmo 9).

Sejam M o módulo primo, tal que $r^{n-1} < M < r^n$ e $\text{mdc}(M, r^n) = 1$, e a constante pré-computada $\mu = -1/M \bmod r^n$. Os operandos X e Y na forma de resíduo de Montgomery são A e B respectivamente, tal que $A = Xr^n \bmod M$, $B = Yr^n \bmod M$ e $0 \leq AB \leq M^2$. Este algoritmo calcula $ABr^{-n} \bmod M$, o que resulta no resto completamente reduzido na forma de resíduo de Montgomery.

Algoritmo 8 Multiplicação modular de Montgomery paralela

```
1:  procedimento MontgomeryParalelo(A, B)
2:      C ←MultiplicacaoParalela (A, B)              ▷ Algoritmo 9
3:      m ←MultiplicacaoParalela (C, μ) mod r^n      ▷ Algoritmo 9
4:      m ←MultiplicacaoParalela (m, M)              ▷ Algoritmo 9
5:      Z ← (C + m)/r^n
6:      se Z ≥ M então
7:          Z ← Z − M
8:      fim se
9:      retorne Z
10: fim procedimento
```

Sejam s o número de núcleos de processamento disponíveis, n o número de palavras de B e d o tamanho em bits de cada partição. B é dividido em s partes com d dígitos cada. O acúmulo paralelo dos produtos parciais pode ser realizado na forma de árvore binária com no máximo $\lceil log_2 s \rceil$ passos (Algoritmo 9) (BAKTIR; SAVAS, 2013).

O Algoritmo 9 foi proposto por Baktir (BAKTIR; SAVAS, 2013) para paralelizar a multiplicação entre inteiros. Pode-se observar que o acúmulo paralelo dos produtos parciais fica inconsistente, quando s é ímpar. A implementação paralela deste algoritmo, proposta por Giorgi et al. (GIORGI; IMBERT; IZARD, 2013), prevê o paralelismo em um número ímpar de *threads*. Giorgi et al. implementaram o acúmulo parcial estaticamente, com a criação de um bloco paralelo e o uso de barreiras de sincronismo. Também estenderam esta abordagem e propuseram o algoritmo de multiplicação modular de Barrett paralelo, paralelizando da mesma forma a multiplicação de inteiros.

Algoritmo 9 Multiplicação de inteiros paralela

procedimento MULTIPLICACAOPARALELA(A, B) ▷ Calcula $A * B$
  para i **de** 0 **até** $s-1$ **passo** 1 **faça** {bloco paralelo, processado pela *thread* i}
    $t_i \leftarrow Ab_i r^{i*d}$ ▷ $B = [b_{s-1}b_{s-2}...b_0]$
  fim para
  para i **de** 1 **até** $log_2 s$ **passo** 1 **faça** {acúmulo dos produtos parciais em $log_2 s$ passos}
    para j **de** 0 **até** $s/2^i - 1$ **passo** 1 **faça** {bloco paralelo}
      $t_j \leftarrow t_j + t_{j+s/2^i}$
    fim para
  fim para
  retorne t_0
fim procedimento

2.5.2 Multiplicação modular Bipartite

O algoritmo de multiplicação modular Bipartite paraleliza a multiplicação modular de Montgomery, dividindo o multiplicador B em duas partes: B_0 e B_1. O algoritmo de multiplicação de Montgomery processa as palavras do multiplicador B a partir do dígito menos significante (*right-to-left*) e o clássico *Interleaved*, a partir do dígito mais significante (*left-to-right*) (KAIHARA; TAKAGI, 2008).

Em seu trabalho, Giorgi et al. (GIORGI; IMBERT; IZARD, 2013) implementaram a Redução Parcial de Barrett (RPB) no algoritmo Bipartite, ao invés da redução modular clássica *Interleaved*, junto com a Redução Parcial de Montgomery (RPM) (Algoritmo 10).

A multiplicação modular Bipartite calcula os produtos modulares $AB_0r^{-t} \bmod M$ e $AB_1 \bmod M$ em paralelo, realizando no máximo uma subtração após a junção das reduções parciais.

Algoritmo 10 Multiplicação modular Bipartite

procedimento BIPARTITE(A, B) ▷ Calcula $ABr^{-t} \bmod M$
  para i **de** 0 **até** 1 **passo** 1 **faça** {bloco paralelo, processado pela *thread* i}
    se $i = 0$ **então**
      $Z_0 \leftarrow$ **RPM** (M, AB_0, t, μ)
        ▷ $B = B_1.r^t + B_0$ ▷ Algoritmo 11
    senão
      $Z_1 \leftarrow$ **RPB** (M, AB_1, t, ν) ▷ Algoritmo 12
    fim se
  fim para
  $Z \leftarrow (Z_0 + Z_1)$
  se $Z \geq M$ **então**
    $Z \leftarrow Z - M$
  fim se
  retorne Z
fim procedimento

Sejam A, B os inteiros X, Y na forma de Montgomery e n o número de palavras de M, tal que $A = Xr^t \bmod M$, $B = Yr^t \bmod M$, com $t = n/2$. Este algoritmo calcula $ABr^{-t} \bmod M$ (veja equação abaixo), o que resulta no resto completamente reduzido, também na forma de resíduo de Montgomery. Uma nova raiz r^t é utilizada, tal que $0 \leq t \leq n$, $t \in \mathbb{Z}$ e $\text{mdc}(r^t, M) = 1$.

$$\begin{aligned} ABr^{-t} \bmod M &= A(B_0 + B_1 r^t)r^{-t} \bmod M \\ &= (AB_0 r^{-t} \bmod M + AB_1 \bmod M) \bmod M \end{aligned}$$

Segundo (KAIHARA; TAKAGI, 2008), t pode ser calculado como $t = \lceil \alpha n \rceil$, tal que $0 < \alpha < 1$. Pode-se assumir que $t = \lceil n/2 \rceil$, e portanto a equação da multiplicação modular Bipartite pode ser dada por $ABr^{-\lceil n/2 \rceil} \bmod M$.

Seja n o tamanho do primo M, a Redução Parcial de Montgomery (RPM) (Algoritmo 11) calcula $Z \equiv Cr^{-t} \pmod M$, tal que $0 \leq Z \leq \max(r^{m-t}, r^n)$ e m é o tamanho de C. Para isso, é calculado o menor inteiro Q, de modo que $(C + QM)/r^t$ é exato e congruente a $Cr^{-t} \pmod M$. $Q = \mu C \bmod r^t$, onde $\mu = -M^{-1} \bmod r^t$ é pré-computado e Z é o resto.

Algoritmo 11 Redução parcial de Montgomery

1: **procedimento** RPM(M, C, t, μ) ▷ $r^{n-1} < M < r^n$, $\text{mdc}(M, r) = 1, 0 \leq C \leq M^2$,
2: ▷ $C < r^m, 0 \leq t \leq n$ e $\mu = -M^{-1} \bmod r^t$
3: $Q \leftarrow (C \bmod r^t)\mu \bmod r^t$
4: $Z \leftarrow (C + QM)/r^t$
5: **se** $Z \geq \max(r^{m-t}, r^n)$ **então**
6: $Z \leftarrow Z - M$
7: **fim se**
8: **retorne** Z ▷ $Z \equiv Cr^{-t} \bmod M$
9: **fim procedimento**

A Redução Parcial de Barrett (RPB) (Algoritmo 12) calcula $Z \equiv C \pmod{M}$, tal que $0 \leq Z < r^{m-t}$ e $t \leq m - n$. Para calcular a aproximação do quociente Q, a constante pré-computada de Barrett é substituída por $\nu = \lfloor r^{n+t}/M \rfloor$ e Q é calculado conforme a Equação 2.20.

$$Q = \left\lfloor \frac{\left\lfloor \frac{C}{r^{m-t}} \right\rfloor \frac{r^{n+t}}{M}}{r^t} \right\rfloor r^{m-n-t} \tag{2.20}$$

Algoritmo 12 Redução parcial de Barrett

1: **procedimento** RPB(M, C, t, ν) ▷ $r^{n-1} < M < r^n,\ 0 \leq C < M^2,\ C < r^n,$
2: ▷ $0 \leq t \leq m - n$ e $\nu = \lfloor r^{n+t}/M \rfloor$
3: $Q \leftarrow \lfloor C_1 \nu / r^t \rfloor / r^{m-n-t}$ ▷ $C = C_1 r^{m-t} + C_0$, onde $0 \leq C_0 < r^{m-t}$
4: $Z \leftarrow C - QM$
5: **enquanto** $Z \geq M$ **faça**
6: $Z \leftarrow Z - M$
7: **fim enquanto**
8: **retorne** Z ▷ $Z \equiv C \bmod M$
9: **fim procedimento**

2.5.3 Multiplicação modular Tripartite

O algoritmo de multiplicação modular Tripartite (Algoritmo 13) (SAKIYAMA et al., 2011) foi proposto com o objetivo de maximizar o nível de paralelismo obtido em relação ao Bipartite (GIORGI; IMBERT; IZARD, 2013). Este algoritmo utiliza o método de Karatsuba, com complexidade sub-quadrática e um *overhead* de sincronismo alto, durante o cálculo da multiplicação. O multiplicador e o multiplicando são particionados e três termos são calculados em paralelo. Segue abaixo a fórmula geral deste algoritmo:

$$\begin{aligned} ABr^{-n/2} \bmod M &= (A_0 + A_1 r^{n/2})(B_0 + B_1 r^{n/2}) r^{-n/2} \bmod M \\ &= (A_0 B_0 r^{-n/2} + A_0 B_1 + A_1 B_0 + A_1 B_1 r^{n/2}) \bmod M \end{aligned}$$

O método de Karatsuba reduz a quantidade de multiplicações presentes na fórmula geral, com os cálculos de X_0, X_1 e X_2: $X_0 = A_0 B_0, X_1 = (A_1 + A_0)(B_1 + B_0), X_2 = A_1 B_1$.

A multiplicação modular Tripartite é portanto calculada pela Equação 2.21.

$$(X_0 r^{-n/2} + (X_1 - X_2 - X_0) + X_2 r^{n/2}) \bmod M \tag{2.21}$$

Algoritmo 13 Multiplicação modular Tripartite

1: **procedimento** TRIPARTITE(A, B, M)
2: **para** i **de** 0 **até** 2 **passo** 1 **faça** {bloco paralelo, processado pela *thread* i}
3: **se** $i = 0$ **então**
4: $X_0 \leftarrow A_0 B_0$ ▷ $A = A_1.r^{n/2} + A_0$ e $B = B_1.r^{n/2} + B_0$
5: $Z_0 \leftarrow$ **RPM** $(M, X_0, n/2, \mu)$ ▷ Algoritmo 11
6: **senão se** $i = 1$ **então**
7: $X_1 \leftarrow (A_1 + A_0)(B_1 + B_0)$
8: **senão**
9: $X_2 \leftarrow A_1 B_1$
10: $Z_2 \leftarrow$ **RPB** $(M, X_2 r^{n/2}, n/2, \nu)$ ▷ Algoritmo 12
11: **fim se**
12: **fim para**
13: $Z_1 \leftarrow (X_1 - X_2 - X_0)$
14: $Z \leftarrow (Z_0 + Z_1 + Z_2)$
15: **enquanto** $Z \geq M$ **faça**
16: $Z \leftarrow Z - M$
17: **fim enquanto**
18: **retorne** Z ▷ $Z \equiv ABr^{-n/2} \bmod M$
19: **fim procedimento**

2.5.4 Multiplicação modular Multipartite

A multiplicação modular k-ary Multipartite (GIORGI; IMBERT; IZARD, 2013) é uma generalização dos algoritmos Bipartite e Tripartite, onde k representa o número de partições em que os operandos são divididos. Diferente do algoritmo Tripartite, todos os termos calculados em paralelo são completamente independentes (GIORGI; IMBERT; IZARD, 2013), ou seja, sem sincronismo durante o processamento de cada termo.

Seja $0 \leq A, B < M < r^n$ e $\text{mdc}(M, r^n) = 1$. O produto modular de A por B (divididos em k partições) é apresentado na equação abaixo:

$$ABr^{-n/2} \bmod M = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} A_i B_j r^{d_{i,j}} \bmod M,$$

tal que $d_{i,j} = n(i + j)/k - n/2$, e $-n/2 \leq d_{i,j} \leq 3n/2 - 2n/k$.

A multiplicação modular k-ary Multipartite v1 (Algoritmo 14) otimiza o cálculo de Q (das reduções parciais de Montgomery ou Barrett), tal que todos os produtos parciais de mesmo peso são somados antes do cálculo do respectivo valor de Q.

A multiplicação modular k-ary Multipartite v2 (Algoritmo 15) otimiza o cálculo de Q, tal que todos os produtos parciais de mesmo peso são somados antes de uma única chamada à RPM (Algoritmo 11) ou RPB (Algoritmo 12) para calcular o resto. Quando a redução não é necessária, o algoritmo inclui o produto na lista de resto. Após somar

Algoritmo 14 Multiplicação modular Multipartite v1

procedimento MULTIPARTITEV1(A, B, M)
  para $d_{i,j}$ **de** $-n/2$ **até** $3n/2 - 2n/k$ **passo** n/k **faça** {bloco paralelo}
    $C_{d_{i,j}} \leftarrow \sum A_i B_j$, tal que $(i+j) = \left(\frac{d_{i,j}+n/2}{n}\right) k$
    se $d_{i,j} < 0$ **então**
      $Q_{d_{i,j}} \leftarrow -(C_{d_{i,j}} \mu \bmod r^{t_{i,j}}) r^{n/2+d_{i,j}}$ ▷ $t_{i,j} = -d_{i,j}$
    senão se $d_{i,j} > n - 2n/k$ **então**
      $Q_{d_{i,j}} \leftarrow \left\lfloor \lfloor C_{d_{i,j}} r^{d_{i,j}-n} \rfloor \lfloor \nu r^{t_{i,j}-n/2} / r^{t_{i,j}} \rfloor \right\rfloor r^{n/2}$ ▷ $t_{i,j} = d_{i,j} + 2n/k - n$
    senão
      $Q_{d_{i,j}} \leftarrow 0$
    fim se
  fim para
  $Q \leftarrow \sum Q_{d_{i,j}}$

  $Z \leftarrow \left(\sum C_{d_{i,j}} r^{n/2+d_{i,j}} - QP\right) r^{-n/2}$
  enquanto $Z \geq M$ **faça**
    $Z \leftarrow Z - M$
  fim enquanto
  retorne Z ▷ $Z \equiv ABr^{-n/2} \bmod M$
fim procedimento

Algoritmo 15 Multiplicação modular Multipartite v2

procedimento MULTIPARTITEV2(A, B, M)
  para $d_{i,j}$ **de** $-n/2$ **até** $3n/2 - 2n/k$ **passo** n/k **faça** {bloco paralelo}
    $C_{d_{i,j}} \leftarrow \sum A_i B_j$, tal que $(i+j) = \left(\frac{d_{i,j}+n/2}{n}\right) k$
    se $d_{i,j} < 0$ **então**
      $Z_{d_{i,j}} \leftarrow \mathbf{RPM}(M, C_{d_{i,j}}, t_{i,j}, \mu \bmod r^{t_{i,j}})$ ▷ $t_{i,j} = -d_{i,j}$
    senão se $d_{i,j} > n - 2n/k$ **então**
      $Z_{d_{i,j}} \leftarrow \mathbf{RPB}(M, C_{d_{i,j}} r^{d_{i,j}}, t_{i,j}, \lfloor \nu / r^{n/2-t_{i,j}} \rfloor)$ ▷ $t_{i,j} = d_{i,j} + 2n/k - n$
    senão
      $Z_{d_{i,j}} \leftarrow C_{d_{i,j}}$
    fim se
  fim para
  $Z \leftarrow \sum Z_{d_{i,j}}$
  enquanto $Z \geq M$ **faça**
    $Z \leftarrow Z - M$
  fim enquanto
  retorne Z ▷ $Z \equiv ABr^{-n/2} \bmod M$
fim procedimento

todos os restos parciais, são necessárias algumas subtrações para reduzir completamente o resultado.

As complexidades dos algoritmos paralelos foram analisadas em (GIORGI; IMBERT; IZARD, 2013), e são apresentadas na Tabela 2. Nestes algoritmos, os operandos A, B são divididos em θ_1, θ_2 partes respectivamente. São calculadas $\theta_1\theta_2$ multiplicações em paralelo. O tempo necessário para realizar uma multiplicação com operandos de tamanho m e n é dado por $M(m, n)$ tal que $m \neq n$, ou $M(n)$ se $m = n$.

Tabela 2: Complexidade paralela e custo de sincronismo por algoritmo de multiplicação modular paralelo. $(*)$ Quando $\theta_1\theta_2 = 2$, o algoritmo Bipartite possui apenas 2 sincronismos.

Algoritmo	Complexidade ($\theta_1\theta_2$ núcleos)	# sinc.
Montgomery/Barrett (Algoritmo 8)	$3M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	6
Bipartite (Algoritmo 10)	$2.5M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	$(*)$ 2 ou 6
k-ary Multipartite v1 (Algoritmo 14)	$(2.5 + \frac{9}{4\theta_1\theta_2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	3
k-ary Multipartite v2 (Algoritmo 15)	$(1.5 + \frac{9}{4\theta_1\theta_2} + \frac{\theta_1\theta_2}{2})M\left(\frac{n}{\theta_1}, \frac{n}{\theta_2}\right)$	2

Nota: $(*)$ O algoritmo Bipartite possui apenas 2 sincronismos quando $\theta_1\theta_2 = 2$, e 6 sincronismos quando $\theta_1\theta_2 > 2$.

Pode-se observar na Tabela 2 que dentre os algoritmos relacionados, a multiplicação modular Bipartite (2 *threads*) e k-ary Multipartipartite v2 possuem o menor *overhead* de sincronismo. O k-ary Multipartipartite v1 possui *overhead* de sincronismo maior que o k-ary Multipartipartite v2, no entanto, a complexidade do k-ary Multipartite v2 é superior, caso $(\theta_1\theta_2)/2 > 1$.

2.6 Trabalhos relacionados

Existem diversos trabalhos na literatura que avaliam algoritmos paralelos nos vários níveis do ECC (Figura 1) em *hardware* (normalmente FPGA) ou *software*

Alguns trabalhos exploraram o paralelismo da multiplicação modular na operação de multiplicação de inteiros (em sua forma binária), que permite o cálculo independente das partições em paralelo. O algoritmo de redução de Montgomery é bastante utilizado em tais implementações, pois permite que a redução modular seja realizada eficientemente. As operações de adição e subtração modulares são mais rápidas que a multiplicação modular, de modo que seu paralelismo não é prioritário (LAUE; HUSS, 2008).

Chen et al. (CHEN; SCHAUMONT, 2011) propuseram o paralelismo em *software* do algoritmo de multiplicação modular Montgomery SOS, em uma arquitetura de protótipos

de processadores com memória local distribuída e comunicação por passagem de mensagem. Foi proposto um modelo de balanceamento de tarefas, no qual a multiplicação é calculada pelos processos em paralelo e, as operações agrupadas em blocos de tarefas, cada qual associada a um processador, de modo circular. Experimentos foram realizados com chaves de 512, 1024 e 2048 bits em FPGA, com 2, 4, 8 núcleos de processamento. Nos experimentos com chave de 2048 bits, este algoritmo obteve aceleração de 1,97, 3,68 e 6,13 vezes em 2, 4 e 8 núcleos, respectivamente.

No trabalho de (BAKTIR; SAVAS, 2013), experimentos foram realizados em processadores de propósito geral com 1, 2, 4, 6 núcleos, e comparados com as implementações em apenas um núcleo do algoritmo proposto e da multiplicação de Montgomery CIOS. Neste trabalho, os autores obtiveram até 81% de aceleração na execução do algoritmo para operandos de pelo menos 4096 bits com 2 núcleos. Houve otimização no desempenho da implementação em 4 e 6 núcleos.

As versões 1 e 2 do algoritmo k-ary Multipartite (GIORGI; IMBERT; IZARD, 2013) foram comparadadas com a multiplicação de Montgomery, Barrett, Bipartite e Tripartite para 1, 3, 4, 6, 8 *threads* paralelas e operandos com tamanho entre 1024 e 16384 *bits*. Nos experimentos realizados, o algoritmo bipartite foi o mais rápido para chaves maiores, pois possui menor complexidade paralela. Devido a seu baixo *overhead*, o Multipartite apresentou-se mais adequado para implementações com chaves menores ou com o acréscimo no número de *threads*.

O paralelismo na aritmética de pontos foi explorado nos sistemas de coordenadas projetivas por Al-Haija et al. (AL-HAIJA; ALKHATIB; JAAFAR, 2011), por possuírem mais operações aritméticas modulares que no sistema de coordenadas Afim. Devido à alta dependência entre as operações, o número de *threads* paralelas é limitado, tal que em geral, poucas *threads* são utilizadas eficientemente em paralelo.

Al-Somani et al. (AL-SOMANI; IBRAHIM, 2009) propuseram o paralelismo da multiplicação escalar *Double-and-Add*. O algoritmo proposto divide o escalar k em partições que podem ser processadas em paralelo. A divisão foi implementada com o balanceamento dos custos das operações de multiplicação modular necessárias para calcular a adição e dobro de ponto. Al-Somani et al. analisaram o desempenho de sua proposta, por meio da análise de complexidade em função da quantidade de operações de adição e dobro de ponto, estendendo em (AL-SOMANI, 2010) os experimentos para cálculo entre operandos de 128, 160, 200 e 256 bits.

Al-Somani et al. (AL-SOMANI; FAYOUMI; IBRAHIM, 2014) propuseram outro algoritmo de multiplicação escalar paralelo com pós-computação. Este algoritmo, no entanto, realiza a divisão do escalar k em partições de mesmo tamanho e calcula o produto parcial de cada segmento pelo ponto P em paralelo, seguido da pós-computação de dobros remanescentes. Este algoritmo se apresentou melhor que o proposto em (AL-SOMANI;

IBRAHIM, 2009), quando utilizado no cálculo de duas ou mais multiplicações escalares consecutivas. Experimentos foram realizados com chaves de 128, 160, 200 e 256 bits.

Basu (BASU, 2012) explorou o paralelismo da multiplicação escalar em *software*, utilizando uma variação do método NAF e pré-computação de pontos. Neste algoritmo, o cálculo de segmentos do produto kP pode ser realizado por mais de um núcleo. No entanto, o algoritmo demonstrou ser eficiente para janelas grandes. Foi implementada a soma em paralelo do total dos produtos parciais calculados. Experimentos foram realizados com janelas de diferentes tamanhos e níveis de paralelismo em arquiteturas *multi-core* com até 8 núcleos. Estudos de simulação demonstraram que a multiplicação escalar atingiu aceleração de aproximadamente *N-1*, para N núcleos. Basu avaliou sua proposta apenas com operandos de 1024 bits, utilizando as leis de Amdahl e Gustafson.

Anagreh et al. (ANAGREH; SAMSUDIN; OMAR, 2014) implementaram o algoritmo Addition-Subtraction utilizando a representação MOF. Este algoritmo foi paralelizado, tal que as operações de dobro e adição de ponto foram calculadas em paralelo por *threads* distintas. Foi utilizado um *buffer* circular para armazenar os valores de dobro calculados temporariamente, necessários para o cálculo da adição de ponto. Neste trabalho, os autores obtiveram *speedup* mínimo de 1,41 nas operações com 384 bits, e máximo de 1,90 nas operações com operandos de 160 bits.

2.7 Experimentos

As bibliotecas MARIA (*Modular ARIthmetic Algorithms*) (GIORGI; IMBERT; IZARD, 2013) e RELIC (RELIC is an Efficient LIbrary for Cryptography) (ARANHA; GOUVÊA,) versão 0.3.5 foram avaliadas em *software* na placa Sabre Lite IMX6Quad com sistema operacional Ubuntu Linaro 12.09 (32 bits). As bibliotecas foram compiladas com o compilador GCC e G++, respectivamente.

2.7.1 Plataforma

Neste estudo, foi utilizada a placa de desenvolvimento SABRE Lite (Figura 6) IMX6Quad para a execução de *benchmarks* das bibliotecas MARIA e RELIC.

Esta placa possui apenas 1 MB de memória, processador com frequência de 1 GHz e trabalha com um conjunto limitado de instruções disponíveis pelo processador ARMv7, caracterizando as restrições de um *smartphone* básico. A placa possui um processador IMX 6 Quad ARM Cortex-A9 da *Freescale*, *quad-core* com até 1GHz de processamento, 1 MB de cache L2 e 1 GB de memória DDR-3.

O sistema operacional Ubuntu Linaro 12.09 possui os modos de CPU *interactive*, *conservative*, *ondemand*, *userspace*, *powersave* e *performance*. Nesta configuração de sistema

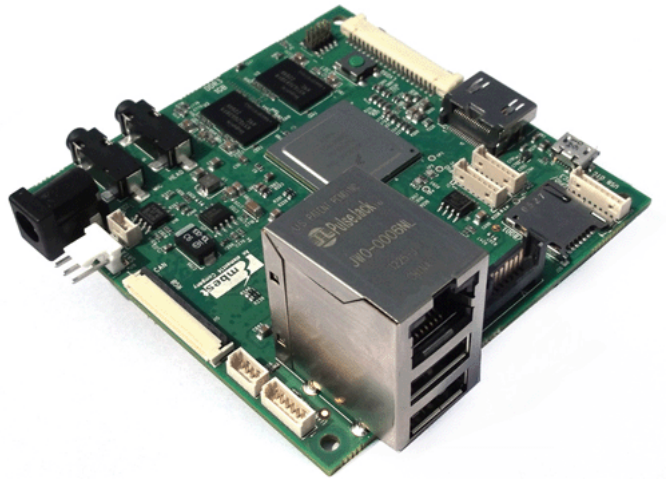

Figura 6: Placa de desenvolvimento Sabre Lite

operacional, o modo *ondemand* é o padrão, a frequência mais baixa disponível é $0,4$ GHz, e a mais alta é $0,8$ GHz. Três modos de CPU que divergem significativamente foram avaliados neste trabalho: *powersave* ($0,4$ GHz), *ondemand* (alterna entre $0,4$ e $0,8$ GHz) e *performance* ($0,8$ GHz).

Tabela 3: Modos de CPU disponíveis em *kernel* Linux

Modo de CPU	Descrição
ondemand	*Alterna as CPU's imediatamente para frequência mais alta, se a carga atingir 95%
performance	*CPU trabalha na frequência máxima
conservative	Alterna as CPUs pouco a pouco para frequência mais alta, se a carga atingir 75%
powersave	CPU trabalha na frequência mínima
userspace	CPU trabalha em frequência especificada pelo usuário

Fonte: https://wiki.archlinux.org/index.php/CPU_frequency_scaling

Anotações: * da mesma forma, estes modos de CPU têm comportamento contrário, alternando para a frequência mais baixa quando a CPU passa a ficar ociosa.

Os modos de CPU *powersave*, *performance* e *userspace* definem uma frequência fixa para a CPU, enquanto o *conservative* e *ondemand* são dinâmicos, tal que a CPU alterna entre a baixa e alta frequência. Experimentos foram realizados nos modos *ondemand*, *powersave* e *performance* e são apresentados no capítulo 2.

2.7.2 Análise de desempenho

Os algoritmos da biblioteca MARIA foram avaliados neste trabalho, a partir de várias execuções de cada multiplicação modular com a mesma entrada, para cada tamanho de operando. A multiplicação escalar foi posteriormente avaliada na biblioteca RELIC

para os mesmos algoritmos, a partir de uma única execução de cada multiplicação escalar com a mesma entrada, para cada tamanho de chave.

Segundo Pacheco (PACHECO, 2011), é improvável que o tempo de execução de um algoritmo seja reduzido devido a uma interferência externa, e portanto, para testes com mesma entrada, deve-se comparar os algoritmos considerando o menor tempo entre suas execuções.

Duas medidas são importantes para a comparação entre os algoritmos paralelos e sequenciais, denominadas *Speedup* e Eficiência.

2.7.3 *Speedup*

O *speedup* (Equação 2.22) é utilizado para identificar quanto houve de redução no tempo do algoritmo paralelo (T_p) com relação ao sequencial (T_s) (PACHECO, 2011).

$$S = T_s/T_p \tag{2.22}$$

É improvável a obtenção de *speedup* linear, devido ao *overhead* associado aos custos de sincronismo (PACHECO, 2011). A criação de *threads* e as operações em memória implicam em *overhead* de processamento, afetando o *speedup* do programa paralelo (GIORGI; IMBERT; IZARD, 2013).

2.7.4 Eficiência

A eficiência (Equação 2.23) permite identificar o quão eficiente os núcleos do processador podem ser utilizados (PACHECO, 2011). Se a eficiência for 1, o programa paralelo possui *Speedup* linear.

$$E = S/p \tag{2.23}$$

2.8 Resumo

Neste capítulo foram abordados os conceitos básicos necessários para a compreensão da aplicação de curvas elípticas em protocolos criptográficos. Algoritmos de multiplicação modular paralelos foram definidos, bem como trabalhos relevantes que exploram o paralelismo nos demais níveis do ECC foram discutidos. Também foi descrita a plataforma adotada para os experimentos, equivalente à de um dispositivo móvel *multicore* atual.

3 Avaliação da biblioteca MARIA

A biblioteca MARIA (Modular ARIthmetic Algorithms) (GIORGI; IMBERT; IZARD, 2013) implementa um conjunto de *benchmarks* para vários algoritmos de multiplicação modular (sequenciais e paralelos). Esta biblioteca utiliza a API (*Application Programming Interface*) OpenMP para realizar o paralelismo em arquitetura *multicore*, e a biblioteca GMP (*The GNU Multiple Precision Arithmetic Library*) para realizar os cálculos aritméticos de multi precisão.

Em seu trabalho, Giorgi et al. (GIORGI; IMBERT; IZARD, 2013) avaliaram o desempenho de alguns algoritmos de multiplicação modular sequenciais e paralelos em uma arquitetura com dois processadores *Intel Xeon X5650 Westmere* com 6 núcleos cada, e operandos na faixa de 1024 a 16384 bits. Arruda et al. (ARRUDA; VENTURINI; SAKATA, 2014) estenderam estes experimentos com os mesmos algoritmos em uma placa SabreLite IMX6Quad, abrangendo operandos com tamanho próximo aos utilizados nas operações com curvas elípticas, padronizadas pelo NIST (entre 128 e 4096 bits). Este capítulo apresenta um estudo aprofundado dos experimentos realizados por Arruda et al., através da elaboração de alguns cenários e realização de diversos experimentos em 1, 2, 3 e 4 *threads*. Segue abaixo a descrição geral dos experimentos realizados.

3.1 Descrição dos experimentos

Foram avaliadas as implementações de multiplicação modular sequenciais (GMP, Barrett, Montgomery), paralelos em 2 *threads* (Montgomery, Bipartite), 3 *threads* (Montgomery, 2-ary Multipartite v1, 2-ary Multipartite v2), e 4 *threads* (Montgomery, Bipartite, 4-ary Multipartite v1, 4-ary Multipartite v2). A descrição destes algoritmos (exceto o GMP) encontra-se na Seção 2.5.

O módulo primo M foi gerado aleatoriamente com tamanho h. Os operandos A e B também foram gerados aleatoriamente, menores que M. A, B e M foram gerados pelo gerador de números pseudo-aleatórios *Mersenne twister* (MATSUMOTO; NISHIMURA, 1998) implementado no GMP, e fixados para cada tamanho de operando. O menor tempo (em μs) foi coletado.

Os sistemas operacionais comumente operam por padrão no modo *ondemand*. No entanto, para a análise de desempenho, é necessário garantir que os algoritmos sejam executados na mesma frequência de CPU, o que pode ser feito utilizando o modo *powersave* ou *performance*. A fim de comparar também os tempos dos algoritmos em um cenário real, bem como analisar os algoritmos para um determinado tamanho de operando na

plataforma em estudo, foram avaliados 2 cenários de teste nos modos de CPU *powersave*, *performance* e *ondemand* (Seção 2.7.1).

Cenário 1 (por *thread*): Foi criado um *benchmark* para cada grupo de algoritmos que executam em um mesmo número de *threads*. Cada grupo (representado por Bench. 1, 2, 3, 4 na figura) é executado independentemente (não há interferência entre os grupos), tal que todos os algoritmos pertencentes a um determinado grupo são executados (na tabela, de cima para baixo) para cada tamanho de operando, dos operandos menores aos maiores, conforme apresentado na Figura 7. Para cada teste (algoritmo por tamanho de operando) foi criado um processo, e estes foram executados em sequência dentro de cada grupo.

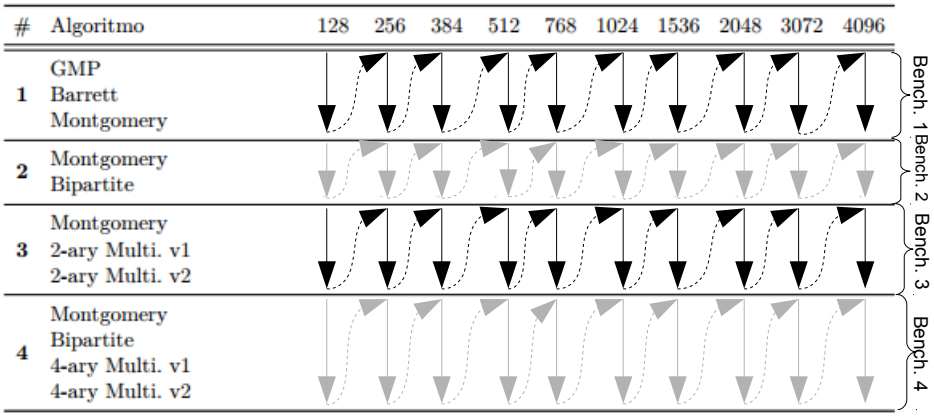

Figura 7: Ilustração do cenário 1 (por *thread*).

Cenário 2 (todas as *threads*): Foi criado um único *benchmark* com todos os algoritmos, na ordem em que aparecem na tabela (de cima para baixo). Todos os algoritmos são executados (de cima para baixo da tabela) para cada tamanho de operando, dos operandos menores aos maiores, conforme apresentado na Figura 8. Um único processo foi criado para o teste dos algoritmos.

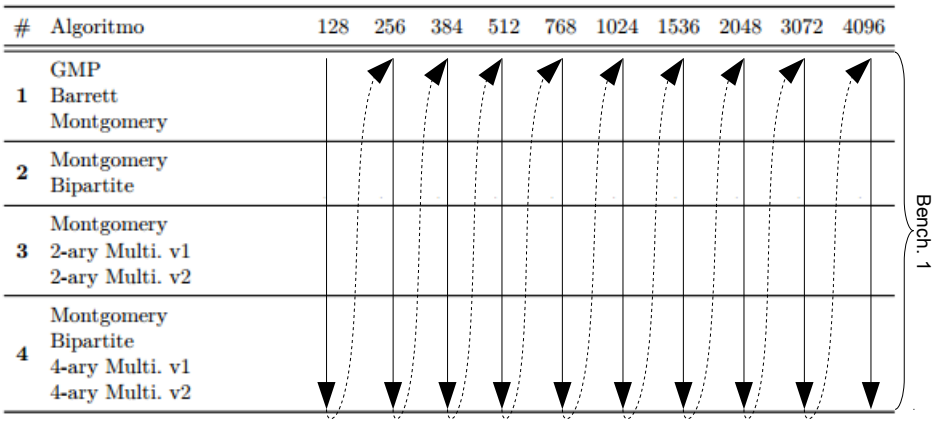

Figura 8: Ilustração do cenário 2 (todas as *threads*).

3.2 Modo de CPU: *powersave*

Esta seção apresenta os resultados de desempenho dos algoritmos de multiplicação modular do MARIA no modo de CPU *powersave*. Neste experimento, foi coletado o menor tempo em 10^3 execuções.

A Tabela 4 apresenta os tempos obtidos no cenário 1. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Nota-se que para operandos de 128 a 512 bits, o algoritmo sequencial GMP foi o que obteve melhor desempenho. A partir de 768 bits, alguns algoritmos paralelos passaram a ter desempenho melhor que os sequenciais, sendo que o 2-ary Multi. v2 foi o melhor para 768 bits, o 4-ary Multi. v2 foi o melhor para 1024 e 1536 bits, e o Bipartite (4 *threads*) foi o melhor para 2048, 3072, 4096 bits. Conforme esperado, os resultados obtidos nos cenários 1 e 2 [1] foram equivalentes, com diferença abaixo de 1,5% para 80% dos casos de teste.

[1] A Tabela 27 - APÊNDICE A apresenta os tempos obtidos no cenário 2.

Tabela 4: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *powersave*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**1,23**	**1,86**	**2,80**	**4,04**	7,41	11,95	23,36	38,76	78,24	129,82
	Barrett	1,39	2,23	3,53	5,97	10,14	16,76	30,30	50,56	92,17	144,23
	Montgomery	1,39	2,20	3,53	6,21	10,04	16,61	30,02	50,06	91,30	142,94
2	Montgomery	6,47	6,44	7,58	9,01	11,88	15,38	25,27	39,21	70,70	108,94
	Bipartite	3,00	3,73	4,30	5,24	7,00	10,12	17,88	30,42	54,03	84,77
3	Montgomery	5,97	6,35	6,80	7,48	9,41	11,65	17,20	25,42	40,52	62,79
	2-ary Multi. v1	4,94	5,09	5,73	6,04	7,46	10,74	15,64	23,09	38,70	60,01
	2-ary Multi. v2	3,50	4,01	4,17	5,26	**6,68**	9,71	15,77	25,50	45,44	71,35
4	Montgomery	6,15	6,48	6,59	7,18	9,53	12,23	17,36	25,09	40,36	62,62
	Bipartite	7,27	7,98	8,27	8,85	10,00	12,44	16,51	**22,38**	**35,33**	**54,50**
	4-ary Multi. v1	5,06	5,42	6,53	7,15	8,27	10,89	15,86	24,07	38,91	59,99
	4-ary Multi. v2	3,74	4,01	4,79	5,45	6,88	**8,89**	**14,38**	25,14	42,83	69,79

3.3 Modo de CPU: *performance*

Esta seção apresenta os resultados de desempenho dos algoritmos de multiplicação modular do MARIA no modo de CPU *performance*. A Tabela 5 mostra o menor tempo em 10^3 execuções. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Nota-se na Tabela 5 que para operandos de 128 a 512 bits, o algoritmo sequencial GMP foi o que obteve melhor desempenho. A partir de 768 bits, alguns algoritmos paralelos passaram a ter melhor desempenho, sendo que o 2-ary Multi. v2 foi o melhor para 768 bits, o 4-ary Multi. v2 foi o melhor para 1024 e 1536 bits, e o Bipartite (4 *threads*) foi o melhor para 2048, 3072, 4096 bits. As mesmas conclusões resultantes dos experimentos no modo *powersave* podem ser observadas no modo *performance*. Os resultados obtidos nos cenários 1 e 2 [2] foram equivalentes, no entanto, com diferença abaixo de 1,5% para 84% dos casos de teste. Isto é, os tempos em ambos os cenários são muito próximos.

Os tempos obtidos no modo *powersave* foram aproximadamente o dobro para os mesmos algoritmos, quando avaliados no modo *performance*, sendo que para 95% dos casos de teste, a proporção entre os tempos do modo *powersave* e *performance* ficou entre 1,90 e 2,1. As variações superiores a 2,1 foram eventuais, e não representam atrasos fixos para quaisquer algoritmo ou tamanho de chave. Além disso, pode-se observar que os algoritmos com menor tempo por operando foram os mesmos em ambos os modos de CPU (Tabelas 4 e 5).

Na Tabela 6 é apresentado o *speedup* dos algoritmos paralelos do MARIA no modo

[2] A Tabela 28 - APÊNDICE A apresenta os tempos obtidos no cenário 2.

Tabela 5: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *performance*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**0,65**	**0,97**	**1,44**	**2,06**	4,00	6,01	11,67	19,41	39,17	64,95
	Barrett	0,73	1,15	1,80	3,03	5,11	8,41	15,20	25,32	46,09	72,41
	Montgomery	0,73	1,14	1,79	3,14	5,06	8,35	15,05	25,08	45,65	71,41
2	Montgomery	3,32	3,65	4,09	4,41	5,96	7,80	12,70	19,59	35,42	54,56
	Bipartite	1,56	1,91	2,21	2,65	3,56	5,12	9,00	15,27	27,14	42,41
3	Montgomery	2,83	3,21	3,42	3,76	4,82	6,01	8,68	12,79	20,33	31,45
	2-ary Multi. v1	2,52	2,59	2,89	3,08	3,77	5,39	7,83	11,58	19,33	30,11
	2-ary Multi. v2	1,79	2,06	2,11	2,67	**3,39**	4,91	7,96	12,79	22,76	35,73
4	Montgomery	3,11	3,27	3,33	3,67	4,82	6,39	8,74	12,59	20,26	31,35
	Bipartite	3,68	4,03	4,20	4,47	4,86	6,27	8,29	**11,20**	**17,88**	**27,30**
	4-ary Multi. v1	2,57	2,76	3,27	3,62	4,27	5,49	7,94	12,06	19,48	30,03
	4-ary Multi. v2	1,89	2,06	2,45	2,77	3,49	**4,50**	**7,21**	12,59	21,47	34,97

performance (Tabela 5), com relação ao menor tempo sequencial para o tamanho de operando. Os menores *speedups* por chave estão em negrito, e os *speedups* maiores que 1 estão com fundo cinza.

Tabela 6: *Speedup* (s) dos algoritmos implementados no cenário 1 do MARIA (Tabela 5). Os maiores *speedups* por chave estão em negrito, e os *speedups* maiores ou iguais a 1 estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
2	Montgomery	0,20	0,27	0,35	0,47	0,67	0,77	0,92	0,99	1,11	1,19
	Bipartite	0,42	0,51	0,65	0,78	1,12	1,17	1,30	1,27	1,44	1,53
3	Montgomery	0,23	0,30	0,42	0,55	0,83	1,00	1,34	1,52	1,93	2,06
	2-ary Multi. v1	0,26	0,37	0,50	0,67	1,06	1,12	1,49	1,68	2,03	2,16
	2-ary Multi. v2	0,36	0,47	0,68	0,77	**1,18**	1,23	1,47	1,52	1,72	1,82
4	Montgomery	0,21	0,30	0,43	0,56	0,83	0,94	1,33	1,54	1,93	2,07
	Bipartite	0,18	0,24	0,34	0,46	0,82	0,96	1,41	**1,73**	**2,19**	**2,38**
	4-ary Multi. v1	0,25	0,35	0,44	0,57	0,94	1,10	1,47	1,61	2,01	2,16
	4-ary Multi. v2	0,34	0,47	0,59	0,74	1,15	**1,34**	**1,62**	1,54	1,82	1,86

Pode-se observar na Tabela 6 que alguns algoritmos paralelos obtiveram *speedup* superior a 1, com operandos a partir de 768 bits. Para operandos de 768 bits, o maior *speedup* foi obtido pelo algoritmo 2-ary Multi. v2. O algoritmo 4-ary Multi. v2 teve maior *speedup* para operandos de 1024 e 1536 bits. Para operandos de 2048, 3072 e 4096 bits, o maior *speedup* foi obtido pelo algoritmo Bipartite (4 *threads*). A partir de 3072 bits, todos os algoritmos paralelos obtiveram *speedup* superior a 1.

3.4 Modo de CPU: *ondemand*

Em estudos anteriores, experimentos foram realizados na placa de desenvolvimento SabreLite IMX6Quad com 10^3 execuções no cenário 1 e modo de CPU *ondemand* (ARRUDA; VENTURINI; SAKATA, 2014). Nesta seção, esse estudo foi ampliado a fim de analisar o tempo dos algoritmos de multiplicação modular apresentados em (GIORGI; IMBERT; IZARD, 2013) no modo de CPU *ondemand*. Este modo é normalmente o padrão, configurado nos sistemas operacionais dos dispositivos móveis atuais para economia de energia. O modo *ondemand* coloca a CPU na frequência do modo *powersave*, quando a carga atinge um limiar inferior a 95%, e eleva para uma frequência equivalente à do modo *performance* quando a carga de CPU atinge um limiar superior a 95% (Seção 2.7.1).

Foram realizados alguns experimentos no modo *ondemand*, a fim de analisar o impacto do aumento no tamanho do operando, número de *threads* e número de execuções no tempo de execução dos algoritmos, considerando que a frequência de CPU pode variar dinamicamente.

Experimento 1: Experimento com operandos entre 128 e 4096 bits (em ordem crescente) nos cenários 1 e 2, em 10^3 execuções;

Experimento 2: Experimento com operandos entre 4096 e 128 bits (em ordem decrescente) nos cenários 1 e 2, em 10^3 execuções;

Experimento 3: Experimento apenas com operandos de 384 bits e *benchmark* independente para cada algoritmo, em 10^3 e 10^5 execuções. Neste caso, não é adequado o uso dos cenários 1 e 2;

Experimento 4: Experimento com operandos entre 128 e 4096 bits (em ordem crescente) nos cenários 1 e 2, em 10^5 execuções.

3.4.1 Experimento 1

O Experimento 1 descreve os menores tempos em 10^3 execuções dos algoritmos de multiplicação modular avaliados em (ARRUDA; VENTURINI; SAKATA, 2014). Este experimento foi realizado a fim de verificar, de que forma o tamanho do operando e o número de *threads* podem influenciar nos resultados dos algoritmos. Assim, esta seção compara os menores tempos obtidos pelos algoritmos nos cenários 1 e 2, com modo de CPU *ondemand*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Na Figura 9 é apresentado um gráfico com os tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand*, apresentados na Tabela 7.

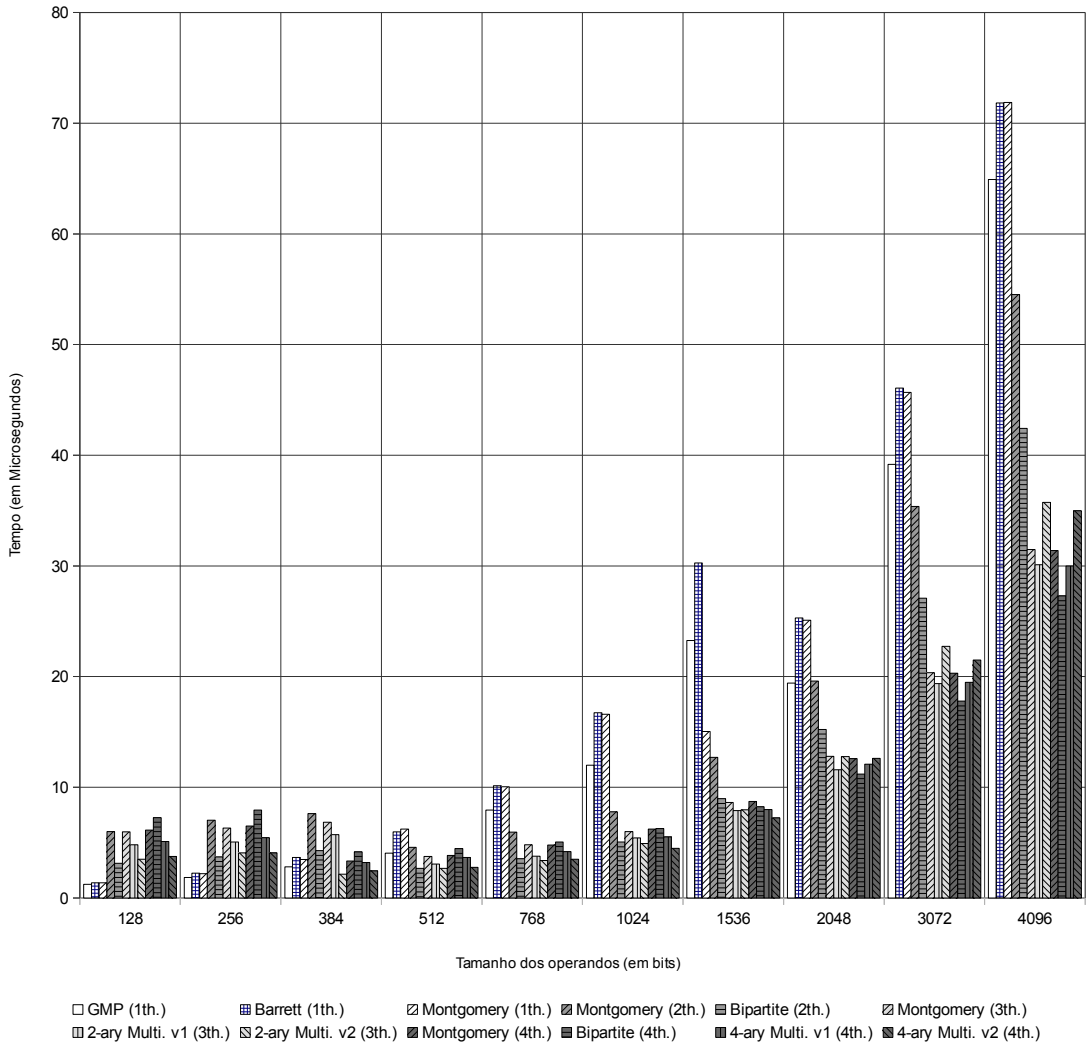

Figura 9: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand*. A relação com os tempos de execução é apresentada na Tabela 7.

Na Tabela 7 são apresentados os tempos de *benchmark* ao executar os experimentos no modo *ondemand* no cenário 1 (todas as *threads*). Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Pode-se observar que, ao executar os *benchmarks* no cenário 1 (por *thread*) com modo de CPU *ondemand* (Tabela 7), os algoritmos que obtiveram melhor tempo (em negrito) para operandos com tamanho menor ou igual a 256 bits e maior ou igual a 768 bits, foram os mesmos, quando comparado aos tempos no modo *powersave* (Tabela 4) e

Tabela 7: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza. O sublinhado mostra o único ponto na linha onde ocorreu uma diminuição no tempo de execução.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**1,23**	**1,85**	2,80	4,05	7,94	11,98	23,26	19,41	39,17	64,93
	Barrett	1,38	2,24	3,67	5,98	10,14	16,73	30,27	25,30	46,08	71,83
	Montgomery	1,38	2,20	3,48	6,21	10,04	16,61	15,03	25,09	45,68	71,88
2	Montgomery	6,00	7,03	7,62	4,58	5,95	7,79	12,71	19,59	35,38	54,51
	Bipartite	3,12	3,71	4,27	**2,67**	3,54	5,04	8,97	15,23	27,09	42,42
3	Montgomery	5,97	6,32	6,83	3,76	4,80	6,00	8,64	12,80	20,35	31,48
	2-ary Multi. v1	4,80	5,06	5,73	3,08	3,77	5,41	7,89	11,59	19,36	30,11
	2-ary Multi. v2	3,50	4,06	**2,14**	2,68	**3,39**	4,92	7,97	12,77	22,74	35,74
4	Montgomery	6,14	6,50	3,33	3,83	4,79	6,21	8,71	12,59	20,30	31,39
	Bipartite	7,26	7,94	4,17	4,45	5,06	6,27	8,24	**11,20**	**17,77**	**27,29**
	4-ary Multi. v1	5,11	5,45	3,20	3,65	4,18	5,51	7,98	12,07	19,49	30,00
	4-ary Multi. v2	3,76	4,08	2,45	2,77	3,50	**4,48**	**7,23**	12,61	21,50	34,98

performance (Tabela 5), respectivamente. No entanto, para operandos de 384 e 512 bits, os algoritmos paralelos 2-ary Multi. v2 e Bipartite (2 *threads*) obtiveram menor tempo no modo *ondemand*, respectivamente, com tempos menores que os dos algoritmos sequenciais a partir de 384 bits.

Na Tabela 8 são apresentados os tempos de *benchmark* ao executar os experimentos no modo *ondemand* no cenário 2 (todas as *threads*). Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Tabela 8: Tempos (em μs) obtidos no cenário 2 (todas as *threads*) dos algoritmos modulares no modo de CPU *ondemand*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza. O sublinhado mostra o único ponto na linha onde ocorreu uma diminuição no tempo de execução.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**1,23**	**1,85**	**1,44**	**2,06**	3,73	6,04	11,76	19,41	39,20	64,95
	Barrett	1,36	2,21	1,82	3,03	5,11	8,41	15,18	25,32	46,04	72,79
	Montgomery	1,39	2,20	1,79	3,14	5,06	8,35	15,06	25,08	45,68	71,12
2	Montgomery	6,23	7,15	4,09	4,39	6,03	7,83	12,76	20,74	35,58	54,53
	Bipartite	2,88	3,70	2,04	2,71	3,57	5,14	9,03	15,30	27,14	42,45
3	Montgomery	5,92	3,26	3,45	3,86	4,83	6,04	8,71	12,68	20,33	31,61
	2-ary Multi. v1	5,00	2,64	2,98	3,12	3,85	5,45	7,91	11,59	19,44	30,14
	2-ary Multi. v2	3,48	2,12	2,21	2,68	**3,44**	4,95	7,97	12,80	22,76	35,85
4	Montgomery	6,18	3,26	3,35	3,83	4,80	6,35	8,73	12,62	20,32	31,36
	Bipartite	7,00	4,02	4,18	4,50	5,14	6,27	8,27	**11,20**	**17,74**	**27,33**
	4-ary Multi. v1	5,03	2,73	3,08	3,64	4,15	5,48	8,01	12,11	19,49	30,09
	4-ary Multi. v2	3,77	2,09	2,47	2,77	3,50	**4,50**	**7,24**	12,62	21,51	35,00

Pode-se observar na Tabela 8 que, os algoritmos que apresentam melhor tempo para o cenário 2 foram os mesmos comparados aos resultados nos modos *powersave* e *performance*, indicando melhor desempenho dos algoritmos paralelos nas operações com operandos de tamanho igual ou superior a 768 bits.

Na Figura 10 é apresentado um gráfico com os tempos (em μs) obtidos no cenário 2 (todas as *threads*) dos algoritmos modulares no modo de CPU *ondemand*, apresentados na Tabela 8.

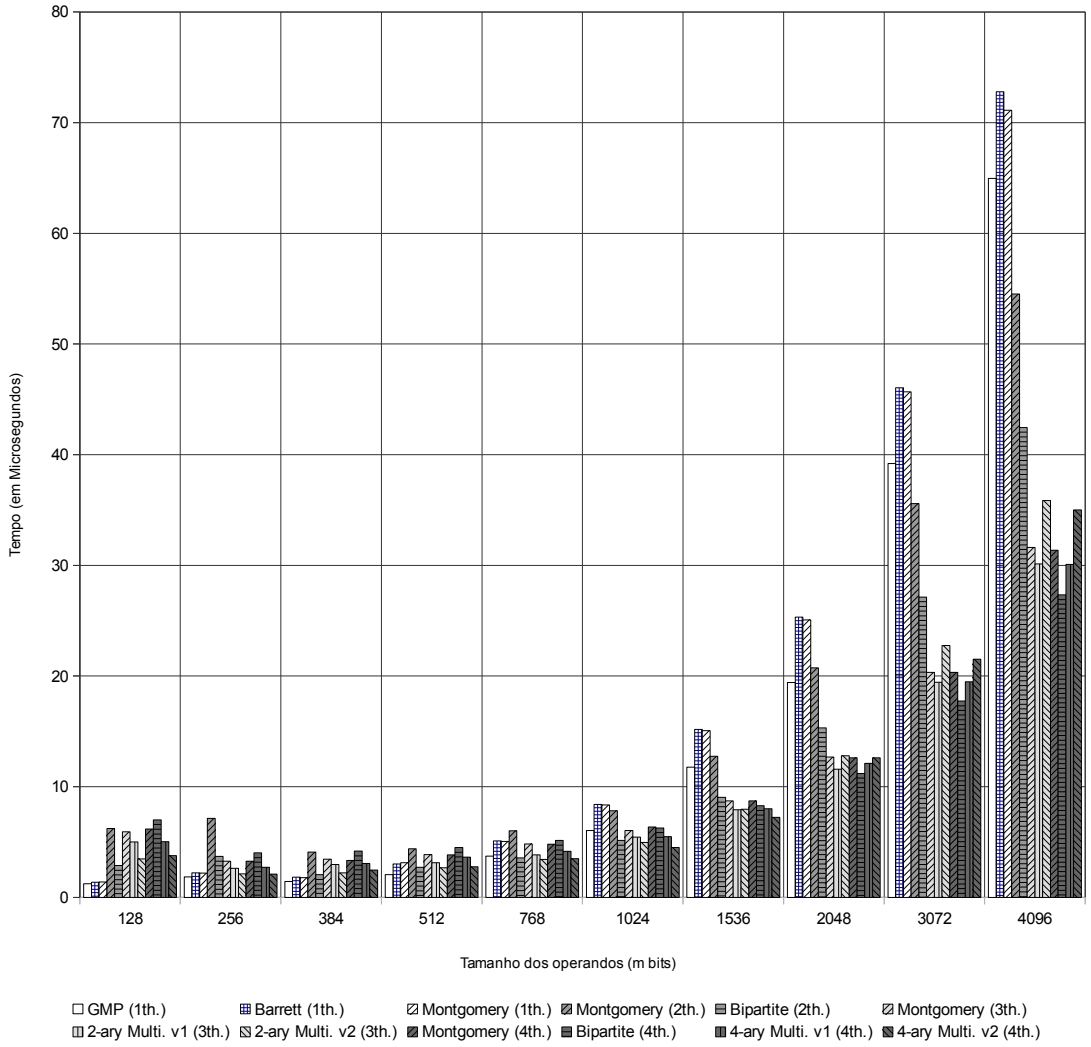

Figura 10: Tempos (em μs) obtidos no cenário 2 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand*. A relação com os tempos de execução é apresentada na Tabela 8.

Como pode ser observado na Figura 10, embora os tempos do GMP no cenário 2 com operandos de 128, 256 e acima de 2048 bits sejam próximos aos do cenário 1 (Figura 9), eles são significativamente diferentes para os demais tamanhos de operando. Ao aumentar

o tamanho dos operandos para um determinado algoritmo sem modificar o número de *threads*, espera-se que o tempo de execução também aumente. Pode-se observar ainda nas Tabelas 7 e 8 que para cada algoritmo, existe um ponto em que ocorre uma variação decrescente no tempo de execução ao aumentar o tamanho dos operandos. Os tempos relacionados foram sublinhados nas tabelas.

Com o objetivo de analisar esta variação decrescente nos tempos de execução, foi elaborada uma tabela comparativa (Tabela 9), que demonstra o grau de proximidade entre os tempos obtidos nos cenários 1 (por *thread*) e 2 (todas as *threads*). Os tempos que diferiram significativamente foram destacados com fundo cinza.

Tabela 9: Relação entre os tempos obtidos no cenário 1 (por *thread*) (Tabela 7) e cenário 2 (todas as *threads*) (Tabela 8). Os tempos que diferiram significativamente foram destacados com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**1,00**	**1,00**	1,95	1,96	2,13	1,98	1,98	1,00	1,00	1,00
	Barrett	**1,01**	**1,01**	2,02	1,97	1,99	1,99	1,99	1,00	1,00	0,99
	Montgomery	**0,99**	**1,00**	1,95	1,98	1,99	1,99	1,00	1,00	1,00	1,01
2	Montgomery	**0,96**	**0,98**	1,86	1,04	0,99	0,99	1,00	0,94	0,99	1,00
	Bipartite	**1,08**	**1,00**	2,09	0,98	0,99	0,98	0,99	1,00	1,00	1,00
3	Montgomery	**1,01**	1,94	1,98	0,97	0,99	0,99	0,99	1,01	1,00	1,00
	2-ary Multi. v1	**0,96**	1,92	1,92	0,99	0,98	0,99	1,00	1,00	1,00	1,00
	2-ary Multi. v2	**1,00**	1,91	0,97	1,00	0,99	0,99	1,00	1,00	1,00	1,00
4	Montgomery	**0,99**	2,00	1,00	1,00	1,00	0,98	1,00	1,00	1,00	1,00
	Bipartite	**1,04**	1,98	1,00	0,99	0,99	1,00	1,00	1,00	1,00	1,00
	4-ary Multi. v1	**1,01**	2,00	1,04	1,00	1,01	1,01	1,00	1,00	1,00	1,00
	4-ary Multi. v2	**1,00**	1,95	0,99	1,00	1,00	1,00	1,00	1,00	1,00	1,00

Mediante a comparação apresentada na Tabela 9, pode-se observar a relação dos tempos coletados no modo de baixa frequência (Tabela 4) com o de alta frequência (Tabela 5). Seja T_1 o tempo de cada algoritmo por tamanho de operando na Tabela 7 e T_2 o tempo do mesmo algoritmo e operando na Tabela 8. Os tempos em negrito referem-se ao momento em que a CPU se encontrava no modo de baixa frequência em ambos os cenários, tal que $T_1/T_2 \approx 1$. Os tempos com fundo cinza, referem-se ao momento em que a CPU se encontrava no modo de baixa frequência no cenário 1 (por *thread*) e alta frequência no cenário 2 (todas as *threads*) ($T_1/T_2 \approx 2$). Os tempos com fundo branco e não negritados, referem-se aos momentos em que a CPU se encontrava no modo alta frequência em ambos os cenários, tal que $T_1/T_2 \approx 1$.

Pode-se identificar que a variação decrescente dos tempos de execução observada nas Tabelas 7 e 8 são decorrentes da mudança do modo de CPU da baixa para alta frequência. Pode-se observar que a carga de CPU atingiu 95% em momentos distintos nos 2 cenários, dependendo dos algoritmos, número de *threads* e tamanho dos operandos.

Na Tabela 9, a região com fundo cinza separa a região de teste em baixa frequência

(esquerda) da região de teste em alta frequência (direita), em que ambos os cenários executaram na mesma frequência. Pode-se observar que os algoritmos que executam em um número maior de *threads* elevam a frequência da CPU para operandos cada vez menores. Nota-se nos experimentos realizados que, uma vez que a frequência foi elevada, esta permanece até o término dos experimentos.

3.4.2 Experimento 2

O Experimento 2 descreve os menores tempos em 10^3 execuções dos algoritmos de multiplicação modular. Este experimento foi realizado com a finalidade de analisar a influência do tamanho dos operandos nos tempos do cenário 1 (por *thread*) com modo de CPU *ondemand*. O modo de execução utilizado é semelhante ao cenário 1 (por *thread*), mas desta vez em ordem decrescente de operandos, conforme apresentado na Figura 11.

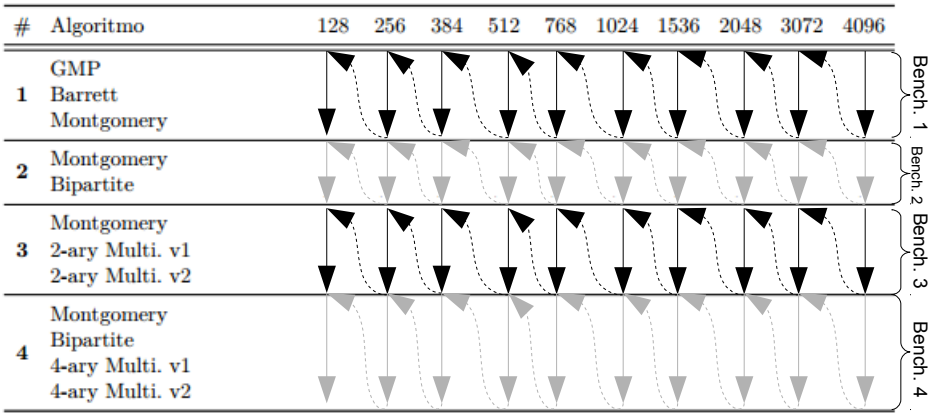

Figura 11: Ilustração do cenário 1, em ordem decrescente.

A Tabela 10 apresenta os tempos (em μs) dos experimentos com operandos entre 128 e 4096 bits. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

Tabela 10: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares com operandos em ordem decrescente no modo de CPU *ondemand*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**0,65**	**0,97**	**1,42**	**2,05**	3,74	6,01	11,51	19,41	39,15	64,95
	Barrett	0,74	1,15	1,83	3,03	5,12	8,41	15,20	25,32	46,09	71,62
	Montgomery	0,74	1,15	1,80	3,15	5,06	8,35	15,04	25,08	45,71	71,61
2	Montgomery	3,14	3,49	4,08	4,43	5,98	7,82	12,76	19,67	35,36	54,55
	Bipartite	1,61	1,86	2,00	2,64	3,53	5,09	8,94	15,21	27,20	42,41
3	Montgomery	2,86	3,03	3,53	4,06	4,85	6,04	8,67	12,65	20,35	31,48
	2-ary Multi. v1	2,55	2,61	2,91	3,09	3,82	5,42	7,88	11,61	19,39	30,08
	2-ary Multi. v2	1,74	1,98	2,03	2,59	**3,35**	4,85	7,86	12,67	22,55	35,65
4	Montgomery	3,15	3,32	3,33	3,82	4,74	6,33	8,59	12,50	20,15	62,17
	Bipartite	3,61	3,92	4,09	4,29	4,92	6,15	8,15	**11,11**	**17,53**	**27,11**
	4-ary Multi. v1	2,57	2,71	2,99	3,59	4,17	5,47	7,91	12,00	19,32	29,89
	4-ary Multi. v2	1,92	2,06	2,44	2,77	3,50	**4,48**	**7,18**	12,59	21,47	34,91

Pode-se observar na Tabela 10 que ao executar os *benchmarks* no cenário 1 (decrescente) (por *thread*) com modo de CPU *ondemand* e operandos em ordem decrescente,

o algoritmo sequencial GMP obteve menor tempo de execução para operandos de 128 a 512 bits. Os algoritmos paralelos passaram a executar em menor tempo com operandos a partir de 768 bits, com o algoritmo 2-ary Multi. v2. O 4-ary Multi. v2 foi o mais rápido para 1024 e 1536 bits. O algoritmo Bipartite (4 *threads*) foi o mais rápido para operandos entre 2048 e 4096 bits.

Na Figura 12 é apresentado um gráfico com os tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares com operandos em ordem decrescente no modo de CPU *ondemand*, apresentados na Tabela 10.

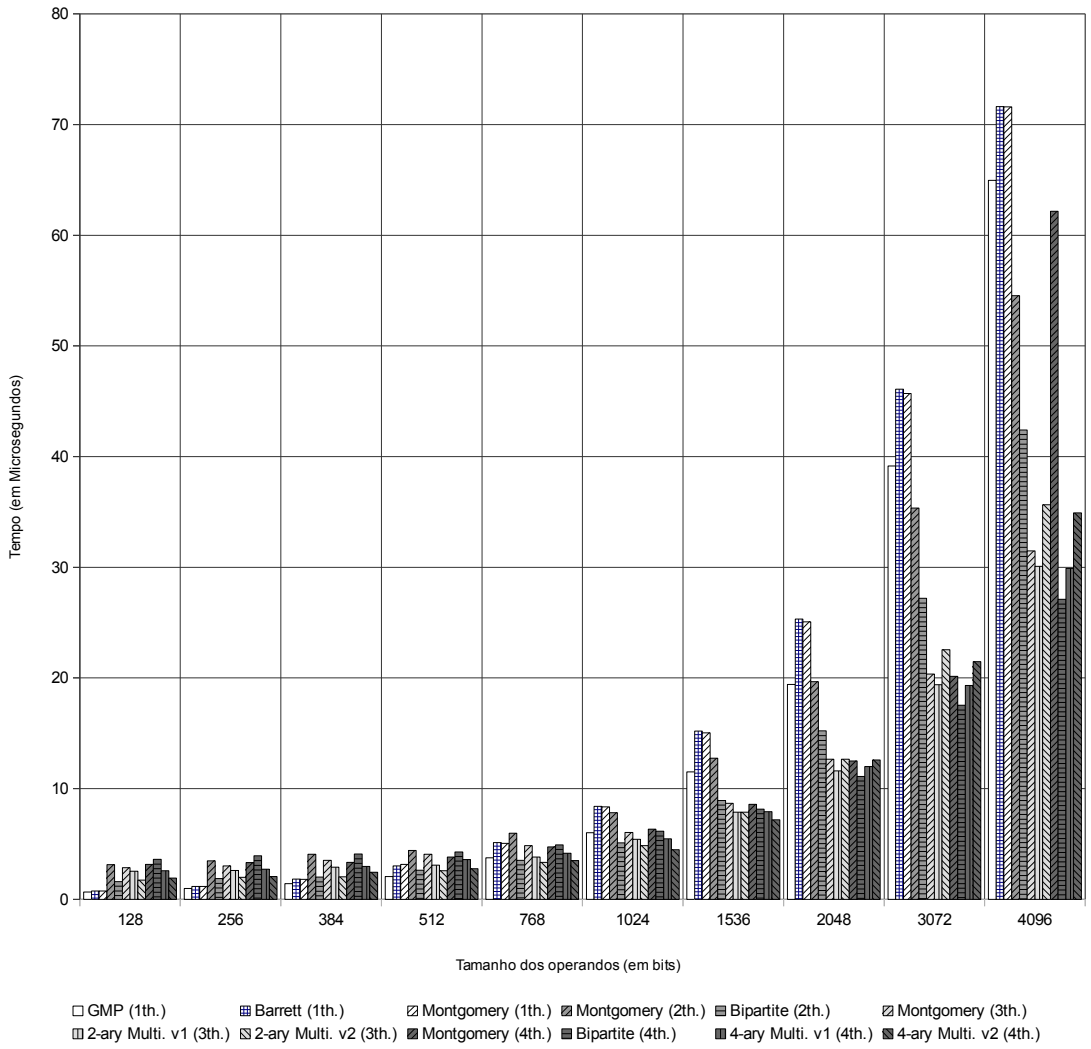

Figura 12: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares com operandos em ordem decrescente no modo de CPU *ondemand*. A relação com os tempos de execução é apresentada na Tabela 10.

Com o objetivo de identificar o comportamento do sistema operacional ao realizar este experimento, foi elaborada uma tabela comparativa (Tabela 11), que demonstra o

grau de proximidade entre os tempos obtidos no cenário 1 (decrescente) (por *thread*) do Experimento 2 (Tabela 10), e no cenário 1 do modo *performance* (Tabela 5). Pode-se observar na Tabela 11 que, os menores tempos por operando da Tabela 10 são próximos aos do mesmo cenário no modo *performance* (Tabela 5). A única relação com tempo divergente foi destacada com fundo cinza. Este resultado é bem diferente do apresentado na Tabela 7, onde os operandos foram avaliados no mesmo cenário em ordem crescente.

Tabela 11: Relação entre os tempos obtidos no cenário 1 (decrescente) (por *thread*) a partir dos operandos maiores no modo de CPU *ondemand* (Tabela 10) e os obtidos no modo de CPU *performance* (Tabela 5). A única relação com tempo divergente foi destacada com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	1,00	1,00	0,99	0,99	0,94	1,00	0,99	1,00	1,00	1,00
	Barrett	1,02	1,00	1,02	1,00	1,00	1,00	1,00	1,00	1,00	0,99
	Montgomery	1,02	1,01	1,01	1,00	1,00	1,00	1,00	1,00	1,00	1,00
2	Montgomery	0,95	0,95	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
	Bipartite	1,03	0,98	0,90	0,99	0,99	0,99	0,99	1,00	1,00	1,00
3	Montgomery	1,01	0,94	1,03	1,08	1,01	1,00	1,00	0,99	1,00	1,00
	2-ary Multi. v1	1,01	1,01	1,01	1,00	1,01	1,01	1,01	1,00	1,00	1,00
	2-ary Multi. v2	0,97	0,96	0,96	0,97	0,99	0,99	0,99	0,99	0,99	1,00
4	Montgomery	1,01	1,01	1,00	1,04	0,98	0,99	0,98	0,99	0,99	1,98
	Bipartite	0,98	0,97	0,97	0,96	1,01	0,98	0,98	0,99	0,98	0,99
	4-ary Multi. v1	1,00	0,98	0,91	0,99	0,97	1,00	1,00	0,99	0,99	1,00
	4-ary Multi. v2	1,02	1,00	0,99	1,00	1,00	1,00	1,00	1,00	1,00	1,00

Pode-se observar a partir da Tabela 11, que o tempo de execução do algoritmo de Montgomery em 4 *threads* na Tabela 10 foi maior que no modo *performance* (Tabela 5), apenas para operandos de 4096 bits. Este *benchmark* iniciou no modo de baixa frequência nas 10^3 avaliações da multiplicação de Montgomery (4 *threads*), e logo em seguida passou para alta frequência com a multiplicação modular Bipartite, não retornando para baixa frequência até o final dos testes.

No Experimento 1, cada *benchmark* do cenário 1 foi executado no modo de alta frequência, a partir de tamanhos diferentes de operandos. No entanto, no Experimento 2, o único algoritmo executado em baixa frequência foi o Montgomery (4 *threads*). Isto ocorreu pois, como o custo para execução das operações com operandos de 4096 bits é bem maior que o das operações com operandos de 128 bits, o algoritmo de Montgomery em 4 *threads* e operandos de 4096 bits levou a CPU a alternar no início do *benchmark* 4 para o modo de alta frequência, não retornando para baixa frequência nas operações entre operandos menores.

3.4.3 Experimento 3

O Experimento 3 descreve os menores tempos em 10^3 e 10^5 execuções dos algoritmos de multiplicação modular. Este experimento foi realizado com a finalidade de identificar o impacto do aumento na quantidade de execuções nos tempos dos algoritmos, ao utilizar o modo de CPU *ondemand*.

A Tabela 12 apresenta o tempo (em μs) dos experimentos com operando de 384 bits. Cada algoritmo foi executado independente dos demais, e o menor tempo em 10^3 e 10^5 execuções foi coletado. Os menores tempos em 10^3 e 10^5 execuções estão em negrito, e os menores tempos por *thread* estão com fundo cinza. Esta tabela também apresenta as relações entre os tempos obtidos no cenário 1 em 10^3 e 10^5 execuções no modo *ondemand*, 10^3 execuções no modo *ondemand* e 10^3 no modo *powersave* (Tabela 4), 10^5 execuções no modo *ondemand* e 10^3 no modo *performance* (Tabela 5).

Tabela 12: Tempos (em μs) dos algoritmos modulares com operandos de 384 *bits* no modo de CPU *ondemand* em 10^3 e 10^5 execuções (10^3*ond.*, 10^5*ond.* respect.). Os menores tempos em 10^3 e 10^5 execuções estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	10^3*ond.*	10^5*ond.*	10^3*ond.*/10^5*ond.*	10^3*ond.*/10^3*powers.*	10^5*ond.*/10^3*perf.*
1	GMP	**2,79**	**1,42**	1,96	0,99	0,99
	Barrett	3,56	1,82	1,96	1,01	1,01
	Montgomery	3,48	1,77	1,97	0,99	0,99
2	Montgomery	7,98	3,97	2,01	1,05	0,97
	Bipartite	4,08	1,92	2,12	0,95	0,87
3	Montgomery	6,41	3,41	1,88	0,94	1,00
	2-ary Multi. v1	5,71	2,85	2,01	1,00	0,98
	2-ary Multi. v2	4,06	2,02	2,02	0,97	0,96
4	Montgomery	6,55	3,24	2,02	0,99	0,97
	Bipartite	8,02	3,95	2,03	0,97	0,94
	4-ary Multi. v1	6,51	2,95	2,21	1,00	0,90
	4-ary Multi. v2	4,79	2,42	1,98	1,00	0,99

O algoritmo GMP foi o mais rápido em 10^3 e 10^5 execuções no modo *ondemand*. Mediante a comparação (10^3*ond.*/10^5*ond.*) apresentada na Tabela 12, pode-se identificar uma relação de aproximadamente 2 vezes entre os tempos obtidos em 10^3 e 10^5 execuções no modo *ondemand*. Isto ocorre por que todos os casos de teste realizados em 10^3 execuções ocorreram em baixa frequência, enquanto os testes realizados em 10^5 execuções ocorreram em alta frequência. A relação dos tempos obtidos em 10^3 e 10^5 execuções nos modos de CPU *powersave* e *performance*, respectivamente pode ser observada na Tabela 12.

O experimento mostra que mesmo com operandos pequenos, se a quantidade de operações for significativa, a CPU pode entrar no modo de alta frequência na execução do primeiro algoritmo do *benchmark*.

3.4.4 Experimento 4

O Experimento 4 descreve os menores tempos em 10^5 execuções dos algoritmos de multiplicação modular no modo de CPU *ondemand*. Este experimento foi realizado a fim de comparar os tempos obtidos no modo de CPU *ondemand* em 10^3 e 10^5 execuções para todos os algoritmos.

A Tabela 13 apresenta os tempos (em μs) dos experimentos com operandos entre 128 e 4096 bits no cenário 1 (por *thread*). Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza. Os resultados obtidos no cenário 2 (todas as *threads*) foram equivalentes e encontram-se na Tabela 29 - APÊNDICE A.

Tabela 13: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand* em 10^5 execuções. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**0,65**	**0,97**	**1,42**	**2,04**	3,98	6,01	11,73	19,39	39,15	64,88
	Barrett	0,73	1,15	1,79	3,01	5,11	8,41	15,17	25,29	46,03	70,79
	Montgomery	0,73	1,14	1,77	3,14	5,04	8,33	15,03	25,04	45,67	70,39
2	Montgomery	3,33	3,50	3,77	4,53	6,00	7,71	12,71	19,61	35,46	54,48
	Bipartite	1,45	1,86	1,98	2,62	3,50	5,06	8,94	15,20	26,92	42,41
3	Montgomery	2,80	3,04	3,47	3,82	4,85	6,03	8,68	12,70	20,33	31,50
	2-ary Multi. v1	2,38	2,57	2,86	3,06	3,76	5,38	7,88	11,55	19,32	30,17
	2-ary Multi. v2	1,82	2,08	2,12	2,68	**3,41**	4,94	7,94	12,82	22,76	35,79
4	Montgomery	3,06	3,09	3,26	3,68	4,68	6,29	8,64	12,47	20,14	31,21
	Bipartite	3,67	4,00	4,17	4,42	5,08	6,27	8,29	**11,20**	**17,64**	**27,26**
	4-ary Multi. v1	2,47	2,65	2,89	3,55	4,04	5,38	7,86	11,95	19,32	29,82
	4-ary Multi. v2	1,91	2,06	2,42	2,77	3,47	**4,47**	**7,20**	12,59	21,48	34,95

Pode-se observar na Tabela 13 que, devido o grande número de execuções, os tempos dos cenários foram comparáveis aos obtidos no modo *performance*, indicando que a CPU entrou no modo de alta frequência a partir do *benchmark* do primeiro algoritmo (GMP) com operandos de 128 bits.

Na Figura 13 é apresentado um gráfico com os tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand* em 10^5 execuções, apresentados na Tabela 13.

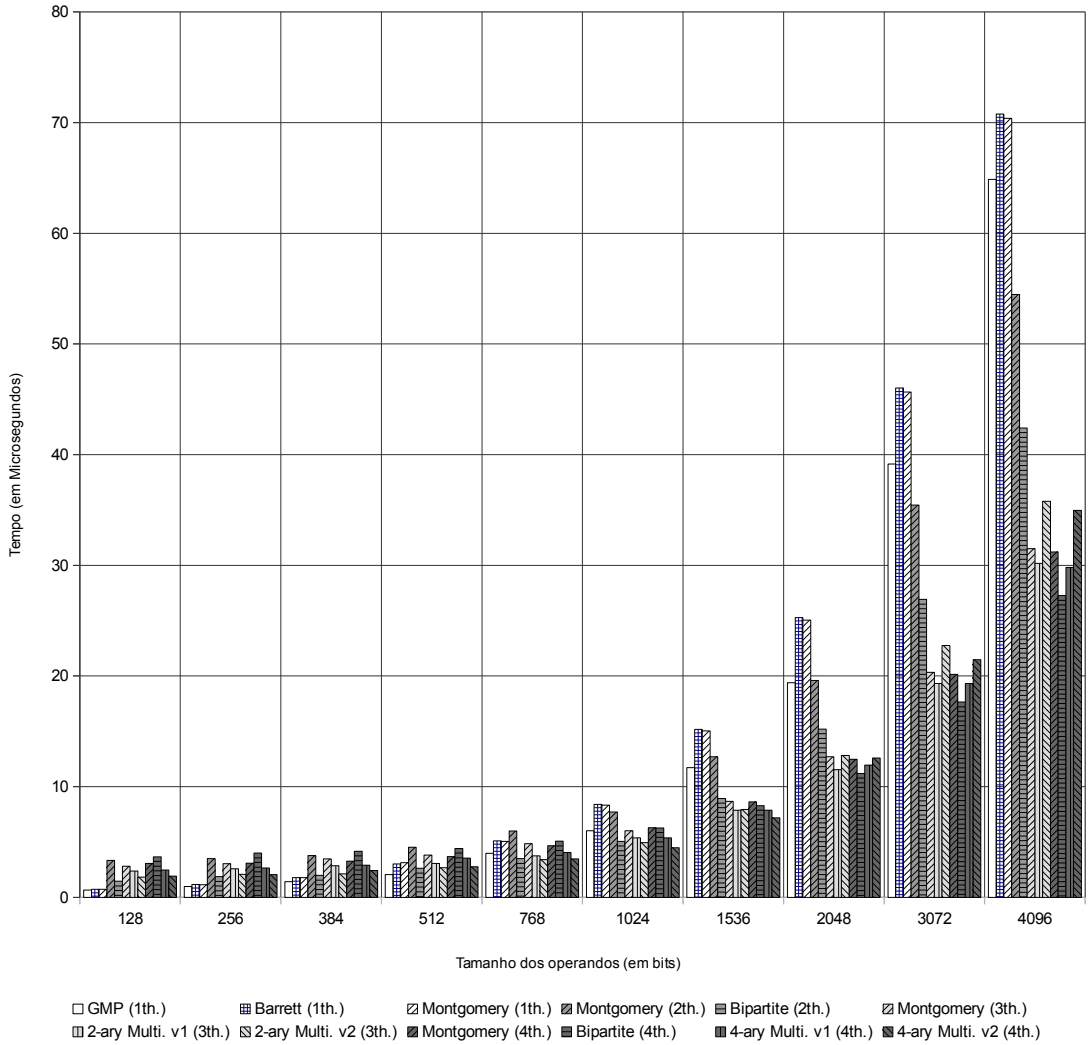

Figura 13: Tempos (em μs) obtidos no cenário 1 (por *thread*) dos algoritmos modulares no modo de CPU *ondemand* em 10^5 execuções. A relação com os tempos de execução é apresentada na Tabela 13.

Pode-se observar na Tabela 13 que o GMP foi o algoritmo mais rápido para operandos entre 128 e 512 bits. Os algoritmos paralelos obtiveram resultados melhores que os sequenciais nas operações com operandos a partir de 768 bits.

3.5 Resumo

Neste capítulo foram apresentados os tempos dos experimentos realizados com os algoritmos de multiplicação modular avaliados em Giorgi et al. (GIORGI; IMBERT; IZARD, 2013), estendidos posteriormente em (ARRUDA; VENTURINI; SAKATA, 2014), com o objetivo de analisar os tempos destes algoritmos para operandos de tamanho

equivalente aos utilizados em protocolos de criptografia basados em curvas elípticas. Para isso, experimentos foram realizados com operandos entre 128 e 4096 bits, nos modos de CPU *powersave*, *performance* e *ondemand*. Dois cenários foram avaliados alterando a ordem dos testes dos *benchmarks* (por *thread* e todas as *threads*). Os resultados nos modos *powersave* e *performance* apresentaram tempos proporcionais entre os cenários, e a relação entre os tempos destes modos foi de aproximadamente 2. Em ambos os cenários dos modos *powersave* e *performance*, os algoritmos paralelos foram mais rápidos que os sequenciais para operandos com pelo menos 768 bits. No cenário 1 com uso do modo de CPU *ondemand*, no entanto, os algoritmos paralelos foram mais rápidos a partir de 384 bits. Isto ocorre devido à mudança da CPU do modo de baixa para o de alta frequência ocorrer para operandos menores quando algoritmos paralelos são utilizados, devido à carga imposta às CPUs.

Embora os modos de CPU *powersave* e *performance* sejam os adequados para comparação entre algoritmos, os experimentos mostraram que em um cenário real, quando o modo *ondemand* é utilizado, um algoritmo mais rápido que impõe menor carga à CPU (ex. GMP com operandos de 384 bits) pode ser executado em maior tempo que outro algoritmo mais lento (ex. 2-ary Multi. v2 com operandos de 384 bits) mas que impõe carga suficiente à CPU, a ponto de fazer com que ela alterne da baixa para alta frequência.

Pôde-se observar também que a multiplicação modular, quando executada com a CPU no modo *ondemand*, e operandos grandes (ex. 4096 bits) com 10^3 execuções, ou operandos pequenos (ex. 128 bits) com 10^5 execuções, é suficiente para elevar a CPU para alta frequência, que é equivalente à frequência do modo *performance*.

4 Experimento da multiplicação escalar em ECC com algoritmos paralelos

A biblioteca de criptografia RELIC (RELIC is an Efficient LIbrary for Cryptography) (ARANHA; GOUVÊA,) versão 0.3.5 possui a implementação de rotinas de testes e *benchmarks* de vários protocolos criptográficos baseados em curvas elípticas: ECDH (*Elliptic curve Diffie–Hellman*), ECMQV (*Elliptic Curve Menezes-Qu-Vanstone*), ECDSA (*Elliptic Curve Digital Signature Algorithm*), ECSS (*Elliptic Curve Schnorr Signature*). O protocolo ECDH é utilizado na troca de chaves simétricas em um canal inseguro.

O protocolo ECDH foi escolhido para a realização dos experimentos descritos neste capítulo. Este protocolo é composto por três operações principais: (1) geração da chave pública através da multiplicação do ponto base pela chave privada (escalar k), (2) produto de uma chave pública por outra privada e (3) derivação da chave simétrica, a partir da coordenada x resultante através de uma função *hash*. A multiplicação escalar (Seção 2.2) é a operação principal e mais custosa deste protocolo, sendo realizada nos passos (1) e (2) da descrição acima.

Este capítulo avalia o comportamento da operação de multiplicação escalar, presente em todos os protocolos criptográficos baseados em curvas elípticas mencionados. Para isso, os algoritmos de multiplicação modular apresentados no Capítulo 3, inclusos na biblioteca MARIA, foram integrados no RELIC.

Experimentos foram conduzidos na placa de desenvolvimento SabreLite IMX6Quad, com o objetivo de analisar o custo da operação de multiplicação escalar, quando implementada em conjunto com a multiplicação modular paralela em *software*. O uso da placa SabreLite IMX6Quad permite que seja considerada a limitação computacional de um conjunto de dispositivos móveis *multi-core* de mesma configuração.

Como identificado no Capítulo 3, o modo de CPU pode interferir diretamente nos tempos dos algoritmos de multiplicação modular avaliados. Portanto, o desempenho do RELIC com os algoritmos integrados foi avaliado nos três modos de CPU: *powersave*, *ondemand*, *performance*. Os modos de CPU *powersave* e *performance* são fixos, e permitem que os experimentos sejam executados utilizando a CPU no modo de baixa ou alta frequência, respectivamente. O modo de CPU *ondemand*, como já foi explicado no Capítulo 2, alterna entre os modos de baixa e alta frequência dinamicamente. O modo de CPU padrão do sistema operacional Ubuntu Linaro 12.09 é *ondemand*. Este modo representa o comportamento real do sistema operacional nesta arquitetura.

Ao integrar os algoritmos de multiplicação modular no RELIC, modificações na

implementação foram necessárias para suportar curvas padronizadas pelo NIST para uso em ECC. Tais modificações são adaptações nos algoritmos para suportar operandos cujo número de bits não é múltiplo do tamanho da palavra do processador, ou do número de *threads*.

Neste capítulo, os tempos da multiplicação escalar com cada algoritmo de multiplicação modular foram analisados, e comparados com os obtidos nos experimentos individuais realizados no MARIA.

4.1 Configuração do RELIC para avaliação da multiplicação escalar do ECC

A biblioteca RELIC é composta por um conjunto de protocolos baseados em curvas elípticas. Estes protocolos realizam o cálculo da multiplicação escalar, que é em geral a operação mais custosa, como visto no Capítulo 2. Na Figura 14 são apresentados os algoritmos utilizados no cálculo da multiplicação escalar dos experimentos realizados nas seções a seguir.

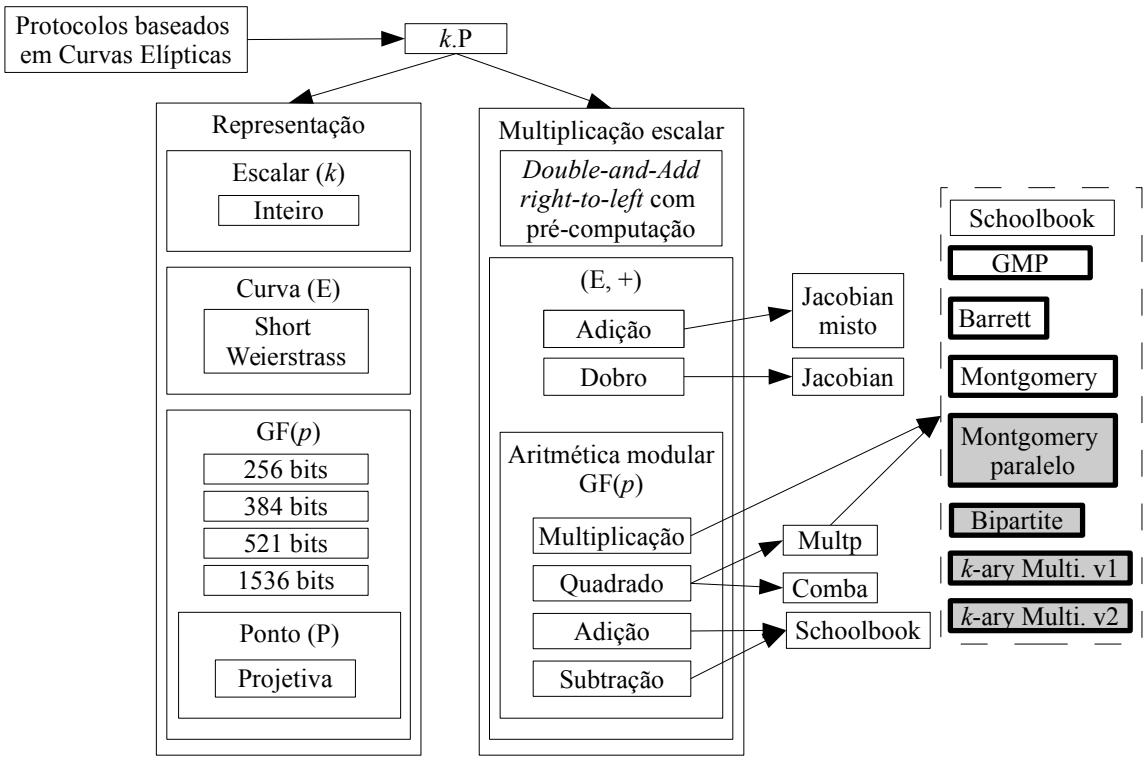

Figura 14: Algoritmos avaliados no RELIC. Os algoritmos integrados (oriundos do MARIA) estão destacados com borda larga, e os paralelos com fundo cinza.

Pode-se observar na Figura 14 que foi utilizada uma variação da multiplicação escalar *Double-and-Add right-to-left*, que realiza a pré-computação de uma tabela com

múltiplos do ponto gerador (Algoritmo 4). O escalar não foi recodificado, tal que foi mantido em sua forma binária. O cálculo da tabela pré-computada é realizado através da operação de dobro de ponto, na fase de pré-computação da biblioteca que não compõe os tempos de *benchmark*. Apenas a operação de adição de ponto é calculada e avaliada dentro da multiplicação escalar.

A multiplicação escalar é composta por operações de aritmética de ponto em $E(GF(p))$. A aritmética de ponto, por sua vez, é composta por operações modulares. A escolha do sistema de coordenadas implica na definição do conjunto de operações aritméticas modulares que são realizadas pelas operações de adição e dobro de ponto.

A inversão modular é uma operação muito custosa, mesmo para os sistemas computacionais atuais. Os sistemas de coordenadas projetivas permitem que nenhuma inversão modular seja necessária nas operações de ponto, ao custo de um acréscimo na quantidade das demais operações modulares. O sistema de coordenadas projetivas *Jacobian short Weierstrass* foi utilizado nos experimentos, a fim de reduzir o custo em relação ao sistema de coordenadas Afim. Na Tabela 14 são apresentados os custos das operações de adição, dobro e adição mista de ponto utilizando este sistema de coordenadas. A adição com coordenadas mistas foi utilizada, por realizar menos operações de multiplicação e quadrado modulares que o de coordenadas não-mistas.

Tabela 14: Custos da aritmética de ponto do sistema de coordenadas projetivas *Jacobian short Weierstrass*. Os custos apresentados são de otimizações propostas, em relação às implementações apresentadas na Tabela 1. Os algoritmos utilizados estão com fundo cinza.

Operação	a	Custo
Adição	-3	$11M + 5S$
Adição mista	-3	$7M + 4S$
Dobro	-3	$3M + 5S$
	0	$2M + 5S$
	-	$1M + 8S$

Fonte: hyperelliptic.org (<http://www.hyperelliptic.org/>)

Neste trabalho, os algoritmos de multiplicação modular paralelos Montgomery paralelo (Algoritmo 8), Bipartite (Algoritmo 10), Multipartite v1 (Algoritmo 14) e Multipartite v2 (Algoritmo 15) foram integrados à versão 0.3.5 da biblioteca RELIC, juntamente com os sequenciais GMP, Montgomery (Algoritmo 6) e Barrett (Algoritmo 7) sequencial. Estes algoritmos foram avaliados, tanto no cálculo da multiplicação, quanto do quadrado modular. A biblioteca RELIC permite escolher algoritmos diferentes para a realização do cálculo da multiplicação e quadrado modulares. O algoritmo Comba (COMBA, 1990) é utilizado por RELIC para o cálculo do quadrado, permitindo dois modos de *benchmark*.

Modo Comba cálculo do quadrado modular pelo algoritmo Comba (COMBA, 1990) (nativo do RELIC), e da multiplicação modular pelo algoritmo avaliado.

Modo Multp cálculo do quadrado modular pelo mesmo algoritmo definido para a multiplicação modular.

Os experimentos realizados envolvem o *benchmark* com 1 execução da multiplicação escalar no RELIC, para cada algoritmo de multiplicação modular e curva utilizados. Foram avaliados os tempos de execução para as operações com as curvas NIST de 256, 384 e 521 bits, e de 1536 bits, onde $a = -3$. Todas as curvas vieram previamente configuradas nesta versão da biblioteca RELIC.

As chaves privadas (k) utilizadas no cálculo da multiplicação escalar foram previamente fixadas, a fim de dar consistência às comparações dos tempos coletados. A fim de auxiliar na reprodução dos experimentos realizados nesta seção, a relação do peso de *hamming* das chaves utilizadas, o número de operações de adição de ponto $P + Q$ e $P + O$ (O é o ponto no infinito) com as respectivas curvas, são apresentados na Tabela 15. Conforme apresentado no Algoritmo 4, o peso de *Hamming* reflete a quantidade de adições de ponto a serem calculadas pela multiplicação escalar. A operação de adição de ponto $Q = P + O$ é implementada no RELIC como $Q = P$.

Tabela 15: Peso de *Hamming* das curvas utilizadas (sobre $GF(p)$) e a quantidade de adições de ponto $\#(P + Q)$ e $\#(P + O)$ realizadas.

Curva	*Hamming (bits)*	$\#(P + Q)$	$\#(P + O)$
NIST de 256 bits	152	151	1
NIST de 384 bits	217	216	1
NIST de 521 bits	277	276	1
1536 bits (RELIC)	718	71	647

Pode-se observar na Tabela 15, que as operações de multiplicação escalar com curvas NIST realizaram apenas 1 operação de adição pelo ponto no infinito (O), enquanto, com a curva de 1536 bits avaliada, foram realizadas 647 operações de adição pelo ponto no infinito. As operações de adição pelo ponto no infinito (elemento identidade) de uma curva $(E, +)$ não possuem nenhum custo, uma vez que não é necessário o cálculo de nenhuma operação aritmética modular.

4.2 Integração dos algoritmos paralelos de multiplicação modular (MARIA) no RELIC

Os algoritmos oriundos do MARIA que foram integrados no RELIC são apresentados na Figura 14 em caixas com borda larga. Os paralelos estão com fundo cinza. Os algoritmos implementados no RELIC utilizam a biblioteca GMP para realizar os cálculos aritméticos de multi precisão. As operações realizadas na biblioteca envolvem a manipulação de palavras, de acordo com a arquitetura do processador utilizado. Os algoritmos do MARIA foram implementados na linguagem C++, com uma função de *benchmark* que desconsidera os custos de pré-alocação de variáveis. As operações de inicialização e pré-alocação são realizadas na instanciação da classe na qual os métodos dos algoritmos estão presentes. A biblioteca RELIC utilizada neste trabalho foi implementada na linguagem de programação C. Devido à impossibilidade de integração direta, a codificação dos algoritmos extraídos do MARIA foi convertida para a linguagem C. A alocação de variáveis foi implementada na etapa de pré-computação da biblioteca, juntamente com o cálculo das constantes de Barret e Montgomery, tal que esta operação passou a ser realizada apenas uma vez antes da execução dos *benchmarks*, e portanto, seu tempo não é contabilizado nos experimentos realizados.

Procurou-se manter as implementações o mais fiel possível às originais durante a integração. No entanto, várias adaptações foram necessárias, uma vez que o MARIA foi desenvolvido para a realização de *benchmarks*, e não para uso em um protocolo de criptografia, como é a proposta deste trabalho. Acredita-se que tais modificações contribuíram para o aumento no custo computacional obtido nos experimentos realizados. Segue abaixo uma descrição detalhada das modificações que foram realizadas nos algoritmos, dentro do processo de integração mencionado.

Multiplicação modular de Barrett

Sejam M o módulo primo e $r = 2^w$, tal que w é o tamanho da palavra e n é a quantidade de palavras de M. A multiplicação modular de Barrett (Seção 2.4.2) utiliza uma constante pré-computada $\nu = \lfloor r^{2n}/M \rfloor$, de modo que não é realizada nenhuma inversão na redução modular. A implementação deste algoritmo na biblioteca MARIA assume que o tamanho h do módulo M é igual a nw, e realiza a divisão por potências de r através de deslocamentos de palavras à direita. Ao integrar no RELIC com chave e operandos de $h = 521$ bits, foi necessário realizar o deslocamento de $\lfloor h/w \rfloor$ palavras e $nw - w\lfloor h/w \rfloor$ bits. O código fonte (na linguagem C) deste algoritmo após sua adaptação encontra-se disponível no APÊNDICE D.2 deste trabalho.

Multiplicação modular de Montgomery

Assim como no algoritmo de multiplicação modular de Barrett, a multiplicação

modular de Montgomery (Seção 2.4.1) realiza a divisão por potências de r, fazendo uso de uma constante pré-computada $\mu = -1/M \bmod r^n$.Nos casos em que $h = nw$, os algoritmos não apresentaram nenhum erro. No entanto, nas operações em que $h \neq nw$, como por exemplo $h = 521$ bits, os algoritmos ficam inconsistentes. O cálculo da constante de Montgomery foi modificado, para ser realizado com raiz r^n, onde $n = \left\lceil \frac{h}{w} \right\rceil$. O código fonte (na linguagem C) deste algoritmo após sua adaptação encontra-se disponível no APÊNDICE D.1 deste trabalho.

Multiplicação modular de Montgomery paralela

A multiplicação modular de Montgomery paralela (Seção 2.5.1) possui operações de multiplicação em sua composição, que são paralelizadas. Este algoritmo realiza o particionamento do(s) operandos em θ blocos. A implementação presente no MARIA, assume que θ é múltiplo do número de palavras dos operandos, tal que se $\theta = 2$, os operandos devem obrigatoriamente possuir um número de palavras múltiplo de 2. O particionamento presente neste algoritmo foi generalizado para funcionar com todos os tamanhos de chave em estudo neste trabalho, em 2, 3 e 4 *threads*. Estas modificações podem ser aplicadas às implementações paralelas de Barrett presentes no MARIA. Segue nas Figuras 15, 16 e 17 o modelo de particionamento da multiplicação de inteiros proposto em 2, 3 e 4 *threads*, respectivamente:

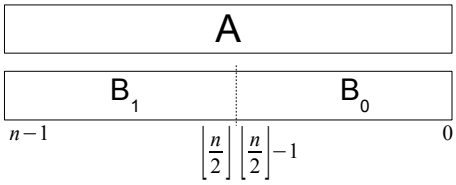

Figura 15: Multiplicação paralela de inteiros em 2 *threads*

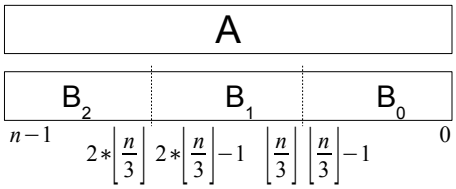

Figura 16: Multiplicação paralela de inteiros em 3 *threads*

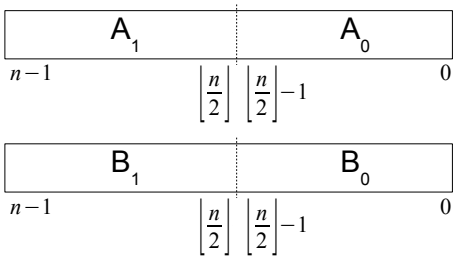

Figura 17: Multiplicação paralela de inteiros em 4 *threads*

O código fonte (na linguagem C) deste algoritmo (2, 3 e 4 *th.* respectivamente) após sua adaptação encontra-se disponível nos APÊNDICES D.5, D.6 e D.7 deste trabalho.

Multiplicação modular Bipartite

A multiplicação modular Bipartite (Seção 2.5.2), realiza o particionamento do multiplicador em duas partes B_0 e B_1. A implementação deste algoritmo no MARIA assume que o multiplicador possui uma quantidade par de palavras. Conforme (KAIHARA; TAKAGI, 2008), pode-se calcular a raiz de Montgomery como $r^{\lceil n/2 \rceil}$, onde $r = 2^w$ e n é a quantidade de palavras de tamanho w do primo M. Foi necessário modificar o particionamento dos operandos, para ser realizado da seguinte forma: B_0 com $\lceil n/2 \rceil$ palavras e B_1 com $\lfloor n/2 \rfloor$ palavras. A equação da multiplicação modular Bipartite implementada é apresentada na Equação 4.1.

$$(AB_0 r^{-\lceil n/2 \rceil} \bmod M + AB_1 \bmod M) \bmod M \tag{4.1}$$

Uma vez que o tamanho de AB_1 foi alterado para $n + \lfloor n/2 \rfloor$ palavras, foi necessário reformular a Redução Parcial de Barret (RPB) (Equação 2.20), apresentada em (GIORGI; IMBERT; IZARD, 2013). Sejam $m = \lfloor n/2 \rfloor + n$ o tamanho de C, $t = \lfloor n/2 \rfloor$ e a constante de Barrett $\nu = \left\lfloor \frac{r^{n+t}}{M} \right\rfloor$. A nova fórmula é apresentada na Equação 4.2.

$$Q = \left\lfloor \frac{\left\lfloor \frac{C}{r^{m-t}} \right\rfloor \nu}{r^t} \right\rfloor r^{m-n-t} \tag{4.2}$$

O código fonte (na linguagem C) deste algoritmo após sua adaptação encontra-se disponível no APÊNDICE D.3 deste trabalho.

Multiplicação modular Multipartite

A multiplicação modular Multipartite (Seção 2.5.4) realiza o particionamento de A e B em k partes. Analogamente à modificação realizada para o algoritmo Bipartite, foi necessário generalizar a fórmula da multiplicação modular 2-ary Multipartite v2 para prever o particionamento consistente dos operandos com quantidade ímpar de palavras, utilizando raiz $r^{\lceil n/2 \rceil}$. Seja $k = 2$, o particionamento foi realizado da seguinte forma: A_0, B_0 com $\lceil n/2 \rceil$ palavras e A_1, B_1 com $\lfloor n/2 \rfloor$ palavras, onde n é a quantidade de palavras de tamanho w do primo M. A nova fórmula da multiplicação modular 2-ary Multipartite v2 é apresentada na Equação 4.3.

$$(A_0 B_0 r^{-\lceil n/2 \rceil} \bmod M + A_0 B_1 + A_1 B_0 + A_1 B_1 r^{\lceil n/2 \rceil} \bmod M) \bmod M \tag{4.3}$$

O código fonte (na linguagem C) deste algoritmo após sua adaptação encontra-se disponível no APÊNDICE D.4 deste trabalho.

A Tabela 16 apresenta a relação dos algoritmos integrados no RELIC, juntamente com os tamanhos das chaves utilizadas nos experimentos. Os algoritmos paralelos Bipartite (4 *threads*), 2-ary Multi. v1 e 4-ary Multi. v1 não foram adaptados para chaves de 521 bits, e serão, juntamente com o algoritmo 4-ary Multi. v2, tratados em trabalhos futuros.

O Schoolbook utilizado nos experimentos é uma implementação nativa do RELIC com redução de Montgomery, e não faz uso da biblioteca de aritmética de multi precisão GMP.

Tabela 16: Algoritmos integrados no RELIC para as chaves em estudo. Os algoritmos não integrados foram marcados com hífen.

#	Algoritmo	256	384	521	1536
1	GMP	X	X	X	X
	Barret	X	X	X	X
	Montgomery	X	X	X	X
2	Montgomery	X	X	X	X
	Bipartite	X	X	X	X
3	Montgomery	X	X	X	X
	2-ary Multi. v1	X	X	-	X
	2-ary Multi. v2	X	X	X	X
4	Montgomery	X	X	X	X
	Bipartite	X	X	-	X
	4-ary Multi. v1	X	X	-	X
	4-ary Multi. v2	-	-	-	-

4.3 Avaliação dos tempos da multiplicação escalar do RELIC

Esta seção analisa os tempos (em μs) obtidos na execução do RELIC em 1, 2, 3 e 4 *threads* na plataforma SabreLite IMX6Quad, para os algoritmos de multiplicação modular utilizados. Seja k um inteiro fixo que representa a chave privada. Foram coletados os tempos da multiplicação do escalar k pelo ponto base G da curva $E(GF(p))$.

Os tempos da multiplicação escalar utilizando os algoritmos integrados no RELIC e o Schoolbook (com redução de Montgomery) no modo Comba são apresentados na Tabela 17, e ilustrados na Figura 18. Pode-se observar que ao utilizar o modo Comba, a multiplicação modular paralela 2-ary Multi. v2 seguida do Bipartite (2 *threads*) foram as mais rápidas para a curva NIST-521, mostrando que a partir deste tamanho de chave, alguns algoritmos paralelos passam a ser mais vantajosos que os sequenciais. A implementação com multiplicação modular sequencial de Montgomery foi a mais rápida para curvas NIST-256 e NIST-384. A multiplicação modular paralela 2-ary Multi. v2 obteve o menor tempo ao utilizar a curva de 1536 bits. Nota-se também que para 1536 bits, todos os algoritmos paralelos obtiveram desempenho superior ao melhor algoritmo sequencial.

Tabela 17: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Comba e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	7776,33	21277,39	49404,76	96349,29
	GMP	8363,21	20755,20	42474,61	66398,79
	Barrett	9077,58	21748,98	43509,50	69068,15
	Montgomery	**4763,83**	**12246,70**	26902,18	51884,21
2	Montgomery	8081,06	16803,09	31588,33	50379,93
	Bipartite	8176,95	17072,06	25079,33	45712,50
3	Montgomery	7200,62	15149,27	29169,18	48794,85
	2-ary Multi. v1	9277,30	19040,67	-	44613,20
	2-ary Multi. v2	8411,52	17256,82	**25063,12**	44742,56
4	Montgomery	7089,74	15019,54	28907,76	48036,95
	Bipartite	10686,49	20347,17	-	44747,50
	4-ary Multi. v1	9479,45	18395,56	-	**44598,24**

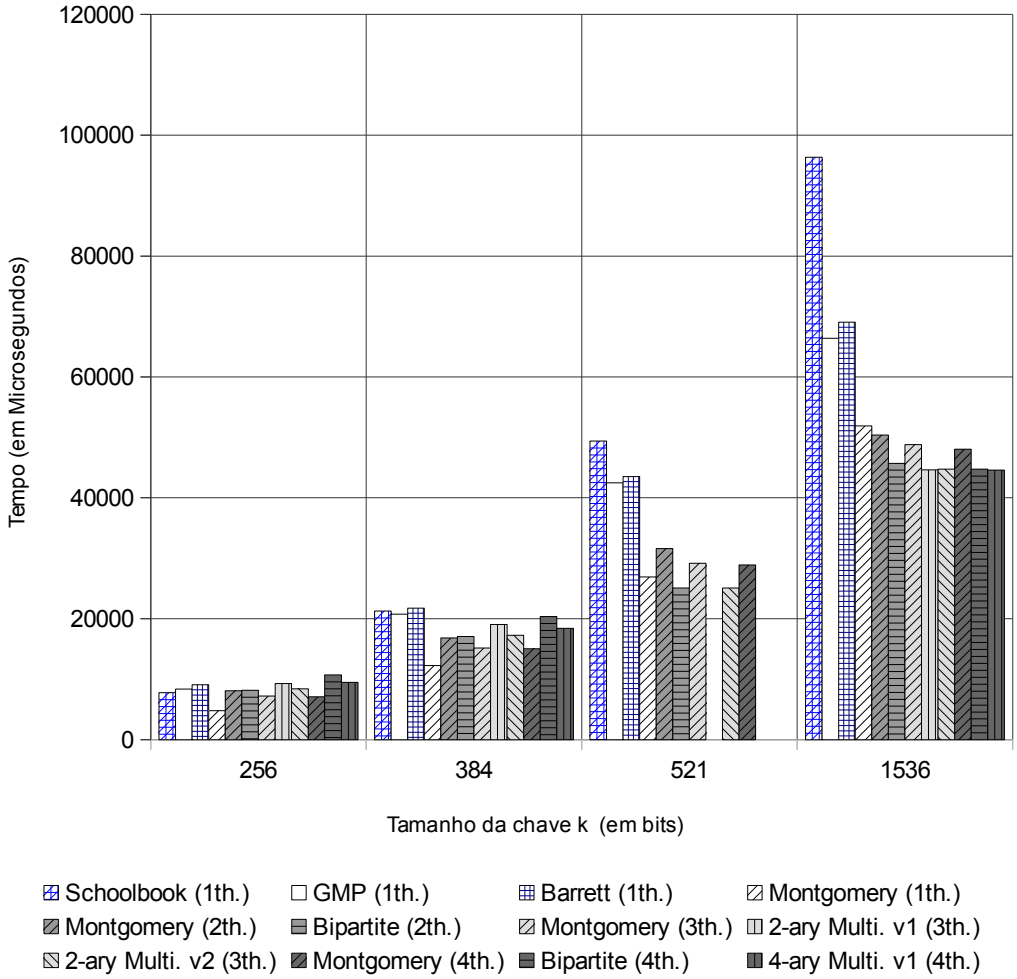

Figura 18: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Comba e modo de CPU *performance*. A relação com os tempos de execução é apresentada na Tabela 17.

Na Tabela 18 é apresentado o *speedup* dos algoritmos implementados no RELIC no modo Comba e modo de CPU *performance*, com relação ao menor tempo sequencial para o tamanho de operando. Os maiores *speedups* por chave estão em negrito, e os *speedups* maiores que 1 estão com fundo cinza.

Tabela 18: *Speedup* (s) dos algoritmos implementados no RELIC (tempos na Tabela 17, ao utilizar o modo Comba. Os maiores *speedups* por chave estão em negrito, e os *speedups* maiores que 1 estão com fundo cinza.

#	Algoritmo	256	384	521	1536
2	Montgomery	0,59	0,73	0,85	1,03
	Bipartite	0,58	0,72	1,07	1,14
3	Montgomery	0,66	0,81	0,92	1,06
	2-ary Multi. v1	0,51	0,64	-	**1,16**
	2-ary Multi. v2	0,57	0,71	1,07	**1,16**
4	Montgomery	0,67	0,82	0,93	1,08
	Bipartite	0,45	0,60	-	**1,16**
	4-ary Multi. v1	0,50	0,67	-	**1,16**

Pode-se observar na Tabela 18 que os algoritmos paralelos Bipartite e 2-ary Multi. v2 obtiveram *speedup* superior a 1 para chave de 521 bits, e todos os paralelos obtiveram *speedup* superior a 1 para chave de 1536 bits.

Os tempos da multiplicação escalar utilizando os algoritmos integrados no RELIC e o Schoolbook (com redução de Montgomery) no modo Multp são apresentados na Tabela 19, e ilustrados na Figura 19.

Tabela 19: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Multp e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	7776,65	21537,62	50738,30	97859,83
	GMP	**2651,32**	**5437,42**	**11075,87**	19700,15
	Barrett	3653,94	7400,58	13253,79	22717,97
	Montgomery	3043,97	6390,00	12420,33	22412,85
2	Montgomery	8465,44	13554,42	20188,24	20835,24
	Bipartite	4458,83	7643,15	12415,02	17766,59
3	Montgomery	6827,15	10926,20	16070,18	35223,82
	2-ary Multi. v1	5665,82	9090,06	-	16709,30
	2-ary Multi. v2	4593,38	7649,74	12239,27	16926,12
4	Montgomery	6679,00	10479,39	15383,35	17030,17
	Bipartite	8295,18	13159,88	-	17218,50
	4-ary Multi. v1	6033,67	9041,88	-	**16548,46**

Pode-se observar na Figura 19 que os tempos do GMP foram inferiores aos demais nos experimentos com modo Multp e curvas NIST-256, NIST-384 e NIST-521. Analisando os algoritmos paralelos para chaves de 521 bits, nota-se que o comportamento foi semelhante ao modo Comba. Isto é, em ambos os modos o 2-ary Multi. v2 foi o que obteve melhor desempenho, seguido do Bipartite (2 *threads*) e Montgomery 4, 3, 2 (*threads*). Para chaves de 1536 bits, o 4-ary Multi. v1 foi o que obteve melhor desempenho (como ocorreu no

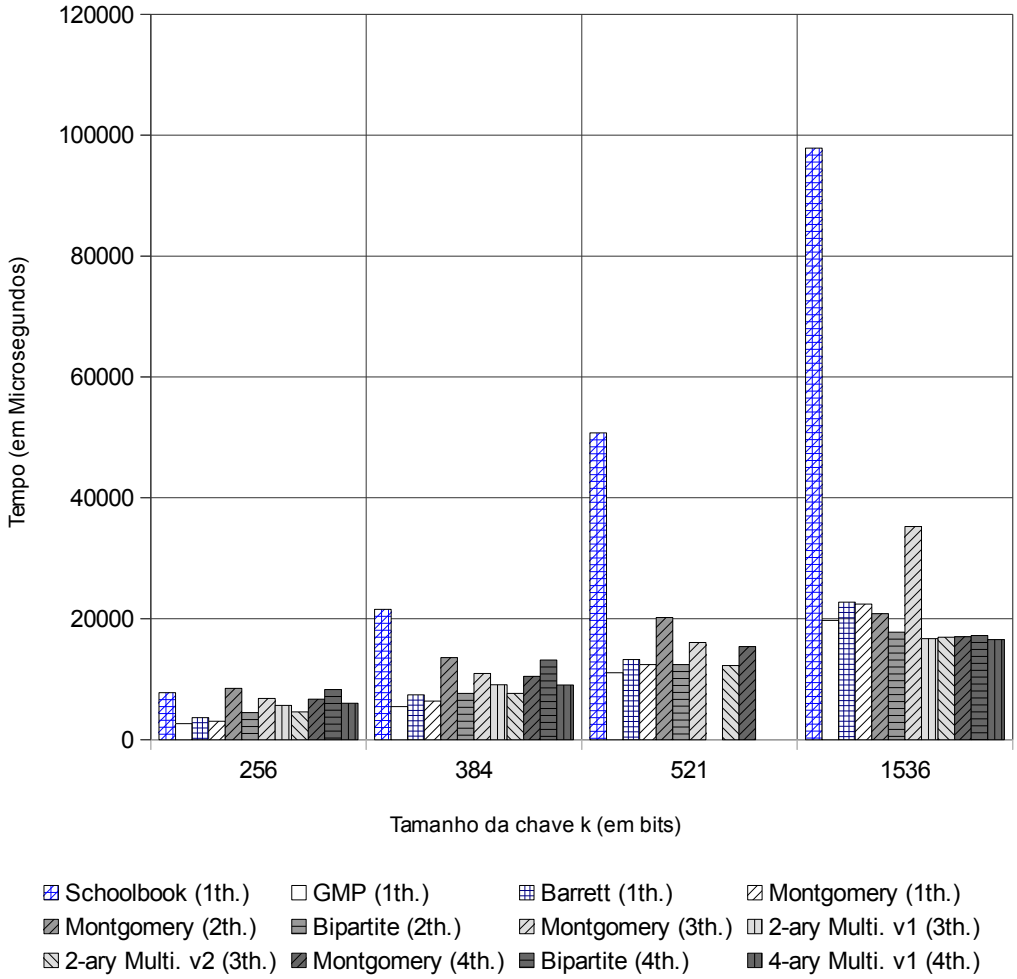

Figura 19: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Multp e modo de CPU *performance*. A relação com os tempos de execução é apresentada na Tabela 19.

modo Comba), e a maioria dos algoritmos paralelos obtiveram desempenho superior ao melhor algoritmo sequencial.

Na Tabela 20 é apresentado o *speedup* dos algoritmos implementados no RELIC no modo Multp e modo de CPU *performance*, com relação ao menor tempo sequencial para o tamanho de operando. Os maiores *speedups* por chave estão em negrito, e os *speedups* maiores que 1 estão com fundo cinza.

Tabela 20: *Speedup* (s) dos algoritmos implementados no RELIC (tempos na Tabela 19), ao utilizar o modo Multp. Os maiores *speedups* por chave estão em negrito, e os *speedups* maiores que 1 estão com fundo cinza.

#	Algoritmo	256	384	521	1536
2	Montgomery	0,31	0,40	0,55	0,95
	Bipartite	0,59	0,71	0,89	1,11
3	Montgomery	0,39	0,50	0,69	0,56
	2-ary Multi. v1	0,47	0,60	-	1,18
	2-ary Multi. v2	0,58	0,71	0,90	1,16
4	Montgomery	0,40	0,52	0,72	1,16
	Bipartite	0,32	0,41	-	1,14
	4-ary Multi. v1	0,44	0,60	-	**1,19**

Pode-se observar na Tabela 20 que alguns algoritmos paralelos obtiveram *speedup* superior a 1 apenas para chave de 1536 bits, dentre os quais o maior *speedup* foi obtido pelo 4-ary Multi. v1.

A Tabela 21 descreve a relação Comba/Multp dos resultados obtidos nos modos Comba (Tabela 17) e Multp (Tabela 19).

Tabela 21: Relação entre os tempos obtidos na multiplicação escalar utilizando quadrado Comba (Tabela 17) e Multp (Tabela 19). As relações com tempos mais divergentes estão em negrito.

#	Algoritmo	256	384	521	1536
1	Schoolbook	1,00	0,99	0,97	0,98
	GMP	**3,15**	**3,82**	**3,83**	**3,37**
	Barrett	2,48	2,94	3,28	3,04
	Montgomery	1,57	1,92	2,17	2,31
2	Montgomery	0,95	1,24	1,56	2,42
	Bipartite	1,83	2,23	2,02	2,57
3	Montgomery	1,05	1,39	1,82	1,39
	2-ary Multi. v1	1,64	2,09	-	2,67
	2-ary Multi. v2	1,83	2,26	2,05	2,64
4	Montgomery	1,06	1,43	1,88	2,82
	Bipartite	1,29	1,55	-	2,60
	4-ary Multi. v1	1,57	2,03	-	2,70

Pode-se observar na Tabela 21 que o algoritmo Schoolbook obteve no modo Multp tempo comparável ao modo Comba, tal que a relação entre eles se manteve próxima a 1. Isso se deve ao fato de que o algoritmo Schoolbook não utiliza otimizações externas senão àquelas implementadas no próprio RELIC.

Além disso, nota-se que ao aumentar o tamanho da chave, houve uma tendência em aumentar a razão Comba/Multp. Isto pode estar relacionado ao aumento no custo do quadrado comba em função do tamanho h dos operandos, sendo que ao aumentar o

tamanho dos operandos no modo Multp, o aumento foi menor, por serem algoritmos mais rápidos.

A maior relação ocorreu nos algoritmos GMP e Barret. Os experimentos com os estes algoritmos foram realizados com os operandos em sua forma decimal, diferentemente dos demais, que realizam as operações entre operandos na forma de Montgomery. Esta diferença implica diretamente nos modos Comba e Multp, pois o modo Comba realiza a redução do quadrado comba com divisão para os algoritmos na forma decimal, e Montgomery para os algoritmos na forma de Montgomery.

É importante ressaltar que a biblioteca GMP utiliza um conjunto de heurísticas para definir qual algoritmo utilizar para realizar cada operação e otimiza o conjunto de instruções utilizado de acordo com a plataforma (GRANLUND et al., 2014). Todos os algoritmos integrados (o que não inclui o Schoolbook e o Comba) utilizam o GMP para realizar os cálculos aritméticos de baixo nível, o que pode explicar esta grande diferença nos tempos de execução.

Os tempos obtidos nos experimentos do RELIC no modo *ondemand* (APÊNDICE B-Tabelas 30 e 31) estão muito próximos aos tempos obtidos com a CPU no modo *performance*. A multiplicação escalar, que é a operação cujos tempos estão sendo analisados, é executada após uma série de operações de inicialização da biblioteca. Através de testes empíricos, observou-se que, devido ao alto custo de inicialização, o ponto de coleta de tempo do *benchmark* foi executado após alterar para o modo de alta frequência com tempos equiparáveis aos do modo *performance*. Ao isolar o *benchmark* da inicialização, aguardando um tempo t (suficiente para retornar ao modo de baixa frequência), o tempo de execução do *benchmark* se aproximou ao obtido no modo *powersave*.

Conforme esperado, os tempos dos experimentos com modo de CPU *powersave* foram aproximadamente o dobro dos obtidos no modo *performance*, nos modos Comba e Multp (APÊNDICE B-Tabelas 32 e 33).

4.4 Avaliação dos tempos dos algoritmos de multiplicação modular integrados no RELIC

Esta seção analisa os tempos dos algoritmos de multiplicação modular após integração na biblioteca RELIC, nos modos Comba e Multp. A multiplicação escalar apresentada na seção anterior, inclui a aritmética de ponto, de acordo com o sistema de coordenadas utilizado no RELIC, que por sua vez envolve o cálculo de um conjunto de operações modulares, cada qual com operandos distintos.

O tempo de cada multiplicação modular foi coletado, a fim de compreender a composição dos tempos obtidos nas operações de multiplicação escalar apresentadas na Seção 4.3. Para o conjunto de tempos obtidos em cada algoritmo, foram calculados os valores de média, mediana, desvio padrão e o coeficiente de variação. As Tabelas 22 e 23 apresentam a média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv)[1] (em %) dos tempos decorrentes das operações de multiplicação e quadrado modulares, respectivamente, presentes na aritmética de ponto da multiplicação escalar com chave de 521 bits. Os cálculos equivalentes para as demais chaves encontram-se no APÊNDICE C.

Tabela 22: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) da multiplicação modular no RELIC com chaves de 521 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	15,96	15,98	0,21	1,32	15,92	15,86	0,28	1,74
	GMP	**3,50**	**3,46**	0,21	5,93	**3,34**	**3,32**	0,20	5,87
	Barrett	4,11	4,08	0,20	4,87	3,99	3,96	0,20	5,11
	Montgomery	3,81	3,77	0,14	3,69	3,70	3,68	0,15	3,93
2	Montgomery	6,78	6,71	0,31	4,54	6,70	6,68	0,14	2,08
	Bipartite	4,85	4,68	0,75	15,39	4,33	4,32	0,14	3,15
3	Montgomery	5,59	5,53	0,26	4,61	5,37	5,35	0,21	3,96
	2-ary Multi. v2	4,98	4,68	0,78	15,61	4,34	4,32	0,19	4,47
4	Montgomery	5,43	5,36	0,25	4,59	5,29	5,27	0,24	4,57

Nota-se comparando as Tabelas 22 e 23, que no modo Comba, apenas o algoritmo Schoolbook obteve média e mediana da multiplicação aproximados ao quadrado modular. Isto ocorreu por que o Schoolbook utiliza o algoritmo de multiplicação com custo equivalente ao quadrado Comba, e ambos calculam a redução de Montgomery.

No modo Multp, no entanto, todos os algoritmos obtiveram média e mediana da multiplicação aproximados aos do quadrado modular, pois ambos utilizam os mesmos algoritmos, variando apenas os operandos.

[1] O cv indica a % que o desvio padrão está para a média.

Tabela 23: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) do quadrado modular no RELIC com chaves de 521 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	15,69	15,65	0,44	2,80	15,92	15,86	0,35	2,18
	GMP	31,54	31,51	0,69	2,17	**3,28**	**3,24**	0,14	4,26
	Barrett	31,55	31,49	0,78	2,48	4,02	4,00	0,14	3,57
	Montgomery	16,89	16,86	0,19	1,14	3,73	3,70	0,14	3,79
2	Montgomery	16,90	16,86	0,30	1,78	6,78	6,70	0,36	5,28
	Bipartite	15,25	15,20	0,52	3,39	4,41	4,35	0,26	5,90
3	Montgomery	16,92	16,88	0,44	2,60	5,44	5,38	0,20	3,73
	2-ary Multi. v2	**15,22**	**15,14**	0,66	4,33	4,41	4,36	0,20	4,63
4	Montgomery	16,92	16,86	0,51	2,99	5,36	5,29	0,25	4,67

A multiplicação modular é a operação mais custosa presente no cálculo da multiplicação escalar. Com o objetivo de identificar esta afirmação após a integração no RELIC, quando adaptações/conversões de tipos de estruturas foram necessárias, foi calculada uma estimativa do tempo gasto com a multiplicação modular, e comparada com o tempo obtido na Seção 4.3.

Os cálculos da estimativa foram realizados utilizando o produto do número de operações pela média dos tempos das operações. Como apresentado na Tabela 15, a operação de multiplicação escalar (com pré-computação) com curvas NIST-521 para a chave selecionada, realizam 276 operações de adição de ponto, tal que $P, Q \neq O$. A quantidade de multiplicações (M) e quadrados (S) modulares realizados por determinada operação de ponto no sistema de coordenadas utilizado é $7M + 4S$ para adição, e $3M + 5S$ para o dobro de ponto (Tabela 14). Como a operação de dobro de ponto pertence à etapa de pré-computação da biblioteca, seu tempo de operação não compõe os tempos de multiplicação escalar apresentados, e não será considerada no cálculo.

Seja a quantidade de adições de ponto apresentados acima, tem-se: $276(7M + 4S) = (1932M + 1104S)$. A primeira adição de ponto, no entanto, realiza o cálculo de $(3M + 2S)$, portanto são $1929M + 1102S$ operações. A quantidade de quadrados e multiplicações modulares é representada por $q = 1102$ e $m = 1929$, respectivamente. A Tabela 24 apresenta o tempo estimado para realizar estes números de operações no modo Multp. A estimativa no modo Comba é apresentada na Tabela 34 - APÊNDICE C. As variáveis $\bar{t_q}$ e $\bar{t_m}$ referem-se à média dos tempos de quadrado e multiplicação modulares (Tabela 22), respectivamente.

Tabela 24: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Multp, chave de 521 bits, $q = 1102$ e $m = 1929$. Os menores tempos por coluna estão em negrito. A diferença do tempo estimado em relação ao tempo total da multiplicação escalar obtida no experimento é apresentada na coluna (%).

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)	%
1	Schoolbook	17545,28	30707,20	48252,48	5
	GMP	**3614,28**	**6437,67**	**10051,95**	9
	Barrett	4426,68	7695,69	12122,37	9
	Montgomery	4115,83	7140,67	11256,50	9
2	Montgomery	7468,99	12916,35	20385,34	-1
	Bipartite	4855,26	8349,90	13205,16	-6
3	Montgomery	5993,13	10359,65	16352,78	-2
	2-ary Multi. v2	4858,54	8369,76	13228,30	-8
4	Montgomery	5908,30	10206,98	16115,28	-5

Obs.: A precisão da média calculada é superior ao apresentado nas Tabelas 22 e 23.

A Tabela 24 mostra que, para os algoritmos sequenciais, o cálculo do tempo estimado para as operações modulares, numa multiplicação escalar, ficou no máximo 10% inferior ao valor obtido nos experimentos (Tabela 19). Já para os algoritmos paralelos, os resultados mostraram que o tempo estimado foi maior que o tempo obtido nos experimentos de multiplicação escalar. Este fato se mantém mesmo quando os tempos experimentais para a multiplicação escalar são obtidos da média de 100 testes. Uma vez que não há diferença nas linhas de código entre os dois experimentos, apenas o ponto em que os tempos são coletados, não foi possível até o momento compreender o motivo do tempo coletado sobre a multiplicação escalar ser inferior.

4.5 Comparativo da multiplicação escalar do RELIC com a multiplicação modular do MARIA

As Tabelas 23 e 24 mostram que o tempo médio das operações modulares obtidos nos experimentos com a biblioteca RELIC foram bem superiores aos obtidos nos testes antes da integração (Capítulo 3). Uma comparação direta destes valores não seria possível, pois nos testes no MARIA foram coletados os tempos mínimos para operandos fixos, com o objetivo de obter os tempos com menor influência do sistema. Porém este teste não fazia sentido no RELIC, pois uma multiplicação escalar utiliza operandos modulares distintos para cada soma sucessiva de ponto. Outros testes foram então realizados com o objetivo de verificar se houve impacto no tempo das operações modulares após a integração das bibliotecas.

Tabela 25: Tempo mínimo (em μs) dos algoritmos de multiplicação modular em 10^3 execuções no RELIC e MARIA (cenário 1) com os mesmos operandos, no modo *performance*. Os menores tempos por operando estão em negrito.

#	Algoritmo	RELIC			MARIA		
		256	384	1536	256	384	1536
1	GMP	**0,98**	**1,45**	11,83	**0,97**	**1,44**	11,67
	Barrett	1,26	1,94	15,50	1,15	1,80	15,20
	Montgomery	1,11	1,73	15,07	1,14	1,79	15,05
2	Montgomery	4,21	4,48	12,68	3,65	4,09	12,70
	Bipartite	1,89	2,14	8,99	1,91	2,21	9,00
3	Montgomery	3,26	3,50	9,36	3,21	3,42	8,68
	2-ary Multi. v1	2,64	2,92	7,68	2,59	2,89	**7,83**
	2-ary Multi. v2	1,95	2,18	7,73	2,06	2,11	7,96
4	Montgomery	3,21	3,32	8,06	3,27	3,33	8,74
	Bipartite	3,68	3,85	7,74	4,03	4,20	8,29
	4-ary Multi. v1	2,62	2,91	**7,33**	2,76	3,27	7,94

A Tabela 25 mostra o tempo obtido com o RELIC para operandos fixos e iguais aos utilizados nos testes com o MARIA. Não foram realizados testes com operandos de 521 bits pois não são suportados pelo MARIA.

Tabela 26: Relação do menor tempo da multiplicação modular no RELIC com a do MARIA em 10^3 execuções, apresentados na Tabela 25. Os menores tempos e as menores relações por operando estão em negrito.

#	Algoritmo	RELIC			MARIA			RELIC/MARIA		
		256	384	1536	256	384	1536	256	384	1536
1	GMP	**0,98**	**1,45**	11,83	**0,97**	**1,44**	11,67	1,01	1,01	1,01
	Barrett	1,26	1,94	15,50	1,15	1,80	15,20	1,09	1,08	1,02
	Montgomery	1,11	1,73	15,07	1,14	1,79	15,05	0,97	0,96	1,00
2	Montgomery	4,21	4,48	12,68	3,65	4,09	12,70	1,15	1,10	1,00
	Bipartite	1,89	2,14	8,99	1,91	2,21	9,00	0,99	0,97	1,00
3	Montgomery	3,26	3,50	9,36	3,21	3,42	8,68	1,01	1,02	1,08
	2-ary Multi. v1	2,64	2,92	7,68	2,59	2,89	**7,83**	1,02	1,01	0,98
	2-ary Multi. v2	1,95	2,18	7,73	2,06	2,11	7,96	0,95	1,03	0,97
4	Montgomery	3,21	3,32	8,06	3,27	3,33	8,74	0,98	1,00	**0,92**
	Bipartite	3,68	3,85	7,74	4,03	4,20	8,29	**0,91**	0,92	0,93
	4-ary Multi. v1	2,62	2,91	**7,33**	2,76	3,27	7,94	0,95	**0,89**	**0,92**

Obs.: A precisão da mediana calculada é superior ao apresentado nas colunas RELIC e MARIA.

Uma comparação entre os tempo obtidos com as duas bibliotecas é apresentada na Tabela 26. Como pode ser observado, há variação de tempo de até 5% para a maioria dos casos, aproximadamente 2/3. Para 9 dos 32 casos testados a variação fica entre 5 e 10%,

ultrapassando este limite para apenas 2 casos. Este comparativo indica que, apesar das alterações necessárias para integrar o código C++ numa biblioteca C e das adaptações realizadas para ajustar os algoritmos para os operandos usados em ECC, a integração não inseriu atrasos significativos nos tempos dos algoritmos.

Como esperado, pode-se observar que os algoritmos de multiplicação modular paralelos influenciaram diretamente nos tempos da multiplicação escalar avaliada na biblioteca RELIC. O uso dos algoritmos paralelos no cálculo do quadrado modular influenciou na melhora dos tempos desta operação, ou seja, no tempo final da multiplicação escalar. A integração apresentada, paraleliza apenas a aritmética modular, tal que a aritmética de pontos (adição e dobro de ponto) se mantém sequencial.

4.6 Experimentos adicionais

A multiplicação escalar utilizada permite que as operações de adição de ponto sejam paralelizadas, uma vez que fazem uso de uma tabela de pontos pré-computada. Experimentos não foram realizados neste modelo, por que algumas funções do RELIC não podem ser consideradas *thread-safe* (PACHECO, 2011), o que em estudos preliminares afetou a integridade dos resultados, quando executado em 2 ou mais *threads*.

Experimentos foram realizados no nível da aritmética de pontos sem resultados positivos, devido à alta dependência entre as operações aritméticas modulares do sistema de coordenadas avaliado. Trabalhos futuros podem ser realizados no sentido de avaliar se uma composição de implementações paralelas nos diversos níveis de operação do ECC, com reuso de *threads*, pode obter melhores resultados.

4.7 Resumo

Neste capítulo foram apresentadas as adaptações realizadas nos algoritmos de multiplicação modular sequenciais e paralelos integrados no RELIC, a fim de garantir sua consistência ao realizar operações com curvas NIST de 256, 384, 521 bits e uma curva de 1536 bits disponível na biblioteca RELIC. Experimentos foram realizados em uma arquitetura de 32 bits, com os algoritmos de multiplicação modular paralelos nos modos Comba e Multp, e modos de CPU *powersave*, *performance* e *ondemand*.

No modo Comba da multiplicação escalar, alguns algoritmos paralelos foram mais rápidos que os sequenciais, para chaves a partir de 521 bits. Este comportamento divergiu do modo Multp, no qual os paralelos foram mais rápidos para chave de 1536 bits, mas não de 521 bits. Ao comparar os resultados e estimar a composição dos tempos da multiplicação escalar no modo Comba, foi possível identificar que, neste modo, o quadrado modular influenciou diretamente no resultado final dos experimentos com a multiplicação escalar.

O alto custo da operação de redução baseada em divisão e quadrado Comba contribuíram para que o modo Comba obtivesse tempos muito superiores aos do modo Multp.

Nas estimativas realizadas para operandos de 521 bits no modo Multp (apenas com a multiplicação e quadrado modulares), a diferença do tempo estivado variou em no máximo 10% para mais ou menos, em relação ao tempo médio de 100 execuções da operação de multiplicação escalar no mesmo cenário.

Embora nos experimentos adicionais, questões como o modo *thread-safe* de algumas funções implementadas no RELIC não tenham favorecido mais experimentos, as mesmas podem de fato, ser modificadas a fim de solucionar esta limitação.

Nos experimentos realizados implementando o sistema de coordenadas Jacobian *short Weierstrass*, os custos de criação e sincronismo das *threads* foram superiores ao ganho obtido no paralelismo das operações modulares, devido à alta dependência entre as operações do sistema de coordenadas avaliado.

Conclusão

Esta dissertação apresentou uma revisão dos conceitos básicos e da estrutura do ECC e de trabalhos relacionados que se propuseram a paralelizar alguma das operações que o compõem. Foi avaliado um conjunto de algoritmos de multiplicação modular paralelos, posteriormente integrados na operação de multiplicação escalar com curvas NIST de 256, 384 e 521 bits, e uma não padronizada de 1536 bits previamente configuradas na biblioteca RELIC. Experimentos foram realizadas em uma placa de desenvolvimento SabreLite IMX6Quad, com processador ARM7.

O modo de CPU *ondemand* é o padrão de alguns sistemas operacionais, inclusive do sistema operacional Ubuntu Linaro utilizado nos experimentos. No Capítulo 3, os experimentos foram realizados considerando 2 ordens para a execução dos algoritmos de multiplicação modular: uma que executa todos os algoritmos com o mesmo número de *threads* e outra que executa todos os algoritmos com o mesmo operando. Assim, ao considerar 10^3 execuções de cada algoritmo, para cada tamanho de operando (128, 256, 384, 512, 768, 1024, 1536, 2048, 3072 e 4096), foi possível identificar que a CPU alternava do modo de baixa para alta frequência em momentos distintos, dependendo da ordem em que os algoritmos foram executados. Isso ocorreu pois a mudança de baixa para alta frequência no modo *ondemand* depende não somente do número de vezes que a multiplicação modular é executada, mas também do tamanho da chave. Ao aumentar o número de execuções para 10^5, todos os tempos foram próximos aos obtidos no modo *performance* (alta frequência). Como os tempos reportados foram os mínimos, os tempos obtidos foram equivalentes, ora ao de baixa, ora ao de alta frequência.

Os algoritmos apresentaram comportamento similar nos modos de CPU *powersave* e *performance*. Foi possível observar que alguns algoritmos paralelos obtiveram melhor tempo para operandos com pelo menos 768 bits em ambos os modos. Como, segundo os experimentos realizados, estes modos de CPU são realizados em modos de frequência não-variáveis, estes tempos permitem comparar os algoritmos adequadamente.

Para a realização dos experimentos de multiplicação escalar, foi necessário integrar algoritmos sequenciais e paralelos, disponíveis na biblioteca MARIA, avaliados no Capítulo 3. Foi necessário adaptar algumas implementações para o contexto de ECC com curvas de 256, 384, 521 e 1536 bits.

Os experimentos realizados no Capítulo 4 mostraram que o tempo de execução da multiplicação escalar não pode ser melhorado levando-se em consideração apenas a otimização da multiplicação modular, ignorando a do quadrado modular (como ocorre no modo Comba). Isto pôde ser observado nos custos dos algoritmos GMP e Barrett

apresentados, onde os quadrados modulares utilizaram redução baseada em divisão (por não se tratar de Montgomery), o que elevou significativamente seus custos, em relação às demais implementações de multiplicação escalar.

Ao executar a multiplicação escalar, foi possível observar que os tempos obtidos no modo de CPU *ondemand* foram próximos aos obtidos no modo *performance*. Isto ocorre devido aos custos de inicialização da biblioteca RELIC.

Nos experimentos realizados na multiplicação escalar com modo Multp, foi possível observar que alguns algoritmos paralelos foram mais rápidos que os sequenciais nas operações com chaves de 1536 bits. No entanto, os tempos paralelos podem começar a ser mais rápidos nas operações com chaves acima de 521 bits, o que não pôde ser avaliado devido às limitações de curvas utilizadas nos experimentos.

Pode-se observar que os algoritmos de multiplicação escalar (paralelos) apresentados, são mais rápidos na plataforma SabreLite IMX6Quad ao se trabalhar com chaves atualmente não padronizadas para uso em ECC. Acredita-se, porém que tamanhos de chave maiores poderão ser adotados futuramente em padrões criptográficos para protocolos baseados em curvas elípticas.

Em trabalhos futuros, os resultados da proposta deste trabalho poderão ser comparados com os obtidos nos trabalhos que paralelizaram outros níveis do ECC, tais como Basu (BASU, 2012) e Al-Somani et. al (AL-SOMANI; IBRAHIM, 2009), (AL-SOMANI; IBRAHIM, 2014). Experimentos mais completos podem ser feitos com a adaptação de todos os algoritmos paralelos estudados, para outras curvas não avaliadas.

O reuso de *threads* implementado pelo GCC pode ser avaliado, juntamente com uma tentativa de reuso explícito de *threads*, a fim de reduzir os tempos de criação e destruição de *threads* decorrentes das implementações paralelas, quando utilizadas no cálculo da multiplicação escalar.

Publicações

Durante o desenvolvimento deste trabalho, foi publicado o artigo (ARRUDA; VENTURINI; SAKATA, 2014), o qual apresenta um estudo teórico e a avaliação dos algoritmos de multiplicação modular paralelos presentes na biblioteca MARIA nas plataformas SabreLite IMX6Quad e Core I7. Neste trabalho, algumas implementações apresentaram *speedup* maior que 1 nas operações entre operandos de 384 bits, na arquitetura IMX6Quad. Discussões sobre os resultados obtidos, bem como o aprofundamento no estudo dos tempos coletados estão no Capítulo 3 deste trabalho. Segue abaixo o *abstract* do artigo mencionado:

“A multiplicação modular é a operação principal da Criptografia de Curvas Elípticas (ECC) sobre corpos finitos primos. Além de ser executada muitas vezes, ela é custosa, pois

requer uma redução modular e um método de multi precisão. Assim, seu custo aumenta proporcionalmente ao tamanho da chave escolhida para o ECC. Esse fator pode ser considerado um problema em dispositivos móveis que, apesar de recentemente possuírem melhores recursos, ainda têm desempenho limitado. Além disso, há uma tendência de processadores *multi-core* em tais dispositivos. Assim, este trabalho tem como objetivo investigar o desempenho da operação de multiplicação modular em nível de software em uma plataforma móvel atual. Estudos recentes mostraram que a multiplicação modular paralela é eficaz apenas em operações com chaves de tamanho grande, como as utilizadas pelo RSA, mas nenhum deles tem foco em plataforma de dispositivos móveis. O resultado deste artigo mostra que, mesmo para chaves menores, como as utilizadas pelo ECC, a multiplicação modular paralela garante uma melhora de desempenho em dispositivos móveis.".

Referências

AFREEN, R.; MEHROTRA, S. C. A review on Elliptic Curve Cryptography for embedded systems. *CoRR*, abs/1107.3631, 2011. Acesso em: 11 ago. 2013. Disponível em: <arxiv.org/pdf/1107.3631>. Citado na página 49.

AHMADI, O.; HANKERSON, D.; RODRÍGUEZ-HENRÍQUEZ, F. Parallel formulations of scalar multiplication on Koblitz curves. *Journal of Universal Computer Science*, [S.l.], v. 14, n. 3, p. 481–504, February 2008. Acesso em: 11 ago. 2013. Disponível em: <http://www.jucs.org/jucs_14_3/parallel_formulations_of_scalar/>. Citado 2 vezes nas páginas 36 e 37.

AL-HAIJA, Q. A.; ALKHATIB, M.; JAAFAR, A. B. Choices on designing gf(p) elliptic curve coprocessor benefiting from mapping homogeneous curves in parallel multiplications. *International Journal on Computer Science and Engineering*, v. 3, n. 2, p. 467–480, 2011. Acesso em: 11 ago. 2013. Disponível em: <http://www.enggjournals.com/ijcse/doc/IJCSE11-03-02-056.pdf>. Citado na página 58.

AL-SOMANI, T.; IBRAHIM, M. *Method for generic-point parallel elliptic curve scalar multiplication*. Google Patents, 2014. US Patent App. 12/963,524. Acesso em: 11 ago. 2013. Disponível em: <https://www.google.com.ar/patents/US20140105381>. Citado na página 102.

AL-SOMANI, T. F. Performance analysis of the postcomputation-based generic-point parallel scalar multiplication method. *Global Journal of Computer Science and Technology*, [S.l.], v. 10, n. 11, p. 32–45, 2010. Citado na página 58.

AL-SOMANI, T. F.; FAYOUMI, A. G.; IBRAHIM, M. K. An efficient and scalable postcomputation-based generic-point parallel scalar multiplication method. *IEICE Electronics Express*, [S.l.], v. 11, n. 19, p. 20140356–20140356, 2014. Acesso em: 01 out. 2014. Disponível em: <http://ci.nii.ac.jp/naid/130004687529>. Citado 2 vezes nas páginas 28 e 58.

AL-SOMANI, T. F.; IBRAHIM, M. K. Generic-point parallel scalar multiplication without precomputations. *IEICE Electronics Express*, [S.l.], v. 6, n. 24, p. 1732–1736, 2009. Acesso em: 11 ago. 2013. Disponível em: <http://ci.nii.ac.jp/naid/130000140900/en/>. Citado 5 vezes nas páginas 28, 38, 58, 59 e 102.

ANAGREH, M.; SAMSUDIN, A.; OMAR, M. A. Parallel method for computing elliptic curve scalar multiplication based on MOF. *International Arab Journal of Information Technology*, [S.l.], v. 11, n. 6, p. 521–525, 2014. Acesso em: 01/10/2014. Disponível em: <http://ccis2k.org/iajit/PDF/vol.11,no.6/5155.pdf>. Citado 2 vezes nas páginas 28 e 59.

ANSARI, B.; HASAN, M. A. High-performance architecture of elliptic curve scalar multiplication. *IEEE Transactions on Computers*, Los Alamitos, v. 57, n. 11, p. 1443–1453, 2008. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/TC.2008.133>. Citado na página 42.

ANTÃO, S.; BAJARD, J.-C.; SOUSA, L. RNS-based elliptic curve point multiplication for massive parallel architectures. *Computer Journal*, London, 2011. Acesso em: 11 ago. 2013. Disponível em: <http://comjnl.oxfordjournals.org/content/early/2011/11/30/comjnl.bxr119.abstract>. Citado na página 51.

ARANHA, D. F.; GOUVÊA, C. P. L. *RELIC is an Efficient LIbrary for Cryptography*. <http://code.google.com/p/relic-toolkit/>. Acesso em: 11 ago. 2013. Citado 2 vezes nas páginas 59 e 81.

ARRUDA, T. V.; VENTURINI, Y. R.; SAKATA, T. C. Performance analysis of parallel modular multiplication algorithms for ECC in mobile devices. *Revista de Sistemas de Informação da FSMA*, [S.l.], v. 1, n. 13, p. 57–67, 2014. Acesso em: 01/08/2014. Disponível em: <http://www.fsma.edu.br/si/edicao13/FSMA_SI_2014_1_Estudantil_4_en.html>. Citado 4 vezes nas páginas 63, 68, 79 e 102.

BAKTIR, S.; SAVAS, E. Highly-parallel Montgomery multiplication for multi-core general-purpose microprocessors. In: GELENBE, E.; LENT, R. (Ed.). *Computer and Information Sciences III*. London: Springer, 2013. p. 467–476. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/978-1-4471-4594-3_48>. Citado 4 vezes nas páginas 28, 50, 51 e 58.

BARRETT, P. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: ODLYZKO, A. (Ed.). *Advances in Cryptology: CRYPTO' 86*. [S.l.]: Springer Berlin Heidelberg, 1987. p. 311–323. (Lecture Notes in Computer Science, v. 263). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/3-540-47721-7_24>. Citado na página 50.

BASU, S. A new parallel Window-based implementation of the Elliptic Curve Point Multiplication in multi-core architectures. *International Journal of Network Security*, v. 14, n. 2, p. 101–108, 2012. Acesso em: 11 ago. 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/ijnsec/ijnsec14.html#Basu12>. Citado 3 vezes nas páginas 28, 59 e 102.

BOOTH, A. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics*, Oxford, v. 4, n. 2, p. 236–240, June 1951. Citado na página 35.

BROWN, M. et al. Software implementation of the NIST elliptic curves over prime fields. In: *TOPICS IN CRYPTOLOGY – CT-RSA 2001*. [S.l.]: Springer, 2001. p. 250–265. (VOLUME 2020 OF LNCS). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/3-540-45353-9_19>. Citado na página 50.

CHABRIER, T.; TISSERAND, A. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In: *SYMPOSIUM ON COMPUTER ARITHMETIC 21., 2013, Austin*. Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic. Washington: IEEE Computer Society, 2013. p. 219–228. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/ARITH.2013.17>. Citado na página 36.

CHEN, Z.; SCHAUMONT, P. A parallel implementation of Montgomery multiplication on multicore systems: Algorithm, analysis, and prototype. *IEEE Transactions on Computers*, [S.l.], v. 60, n. 12, p. 1692–1703, 2011. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/TC.2010.256>. Citado 2 vezes nas páginas 28 e 57.

COHEN, H. et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. 2nd. ed. [S.l.]: Chapman & Hall/CRC, 2006. Citado 6 vezes nas páginas 42, 43, 46, 47, 48 e 139.

COMBA, P. G. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, Armonk, v. 29, n. 4, p. 526–538, 1990. Citado 2 vezes nas páginas 83 e 84.

DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. *IEEE Transactions on Information Theory*, [S.l.], v. 22, n. 6, p. 644–654, November 1976. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/TIT.1976.1055638>. Citado 2 vezes nas páginas 27 e 31.

DOMINGUEZ-OVIEDO, A.; HASAN, M. Algorithm-level error detection for Montgomery ladder-based ECSM. *Journal of Cryptographic Engineering*, [S.l.], v. 1, n. 1, p. 57–69, 2011. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/s13389-011-0003-1>. Citado na página 39.

GALLAGHER, P.; FURLANI, C. *FIPS PUB 186-3 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS)*. [S.l.: s.n.], 2009. Citado na página 40.

GAMAL, T. E. A public key cryptosystem and a signature scheme based on discrete logarithms. In: *Advances in cryptology*. New York: Springer-Verlag New York, 1985. p. 10–18. (Proceedings of CRYPTO 84 on Advances in Cryptology). Acesso em: 11 ago. 2013. Disponível em: <http://dl.acm.org/citation.cfm?id=19478.19480>. Citado na página 32.

GIORGI, P.; IMBERT, L.; IZARD, T. Parallel modular multiplication on multi-core processors. In: NANNARELLI, A.; SEIDEL, P.-M.; TANG, P. T. P. (Ed.). *IEEE Symposium on Computer Arithmetic*. [S.l.]: IEEE Computer Society, 2013. p. 135–142. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/ARITH.2013.20>. Citado 15 vezes nas páginas 28, 49, 50, 51, 52, 54, 55, 57, 58, 59, 61, 63, 68, 79 e 87.

GRANLUND, T. et al. GMP, the GNU Multiple Precision arithmetic library. 2014. Acesso em: 01 out. 2014. Disponível em: <https://gmplib.org/gmp-man-6.0.0a.pdf>. Citado na página 94.

GUPTA, K.; SILAKARI, S. ECC over RSA for asymmetric encryption: A review. *International Journal of Computer Science Issues*, [S.l.], v. 8, n. 3, p. 370–375, 2012. Acesso em: 11 ago. 2013. Disponível em: <http://ijcsi.org/papers/IJCSI-8-3-2-370-375.pdf>. Citado na página 27.

HANKERSON, D.; MENEZES, A. J.; VANSTONE, S. *Guide to Elliptic Curve Cryptography*. Secaucus: Springer-Verlag New York, 2003. Citado 11 vezes nas páginas 31, 32, 41, 42, 43, 44, 45, 46, 47, 136 e 139.

JOHNSON, D.; MENEZES, A.; VANSTONE, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, [S.l.], v. 1, n. 1, p. 36–63, 2001. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/s102070100002>. Citado na página 31.

KAIHARA, M.; TAKAGI, N. Bipartite modular multiplication method. *IEEE Transactions on Computers*, Washington, v. 57, n. 2, p. 157–164, fev. 2008. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/TC.2007.70793>. Citado 5 vezes nas páginas 28, 50, 52, 53 e 87.

KARTHIKEYAN, E. Survey of Elliptic Curve Scalar Mutiplication algorithms. *International Journal of Advanced Networking and Applications*, v. 4, n. 2, p. 1581–1590, 2012. Acesso em: 11 ago. 2013. Disponível em: <http://www.ijana.in/papers/V4I2-8.pdf>. Citado 4 vezes nas páginas 35, 36, 37 e 38.

KNUDSEN, E. Elliptic Scalar Multiplication using Point Halving. In: LAM, K.-Y.; OKAMOTO, E.; XING, C. (Ed.). *Advances in Cryptology - ASIACRYPT'99*. [S.l.]: Springer Berlin Heidelberg, 1999. p. 135–149. (Lecture Notes in Computer Science, v. 1716). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/978-3-540-48000-6_12>. Citado na página 39.

KOBLITZ, N. Elliptic Curve Cryptosystems. *Mathematics of Computation*, Providence, v. 48, n. 177, p. 203–209, 1987. Acesso em: 11 ago. 2013. Disponível em: <http://www.jstor.org/stable/2007884>. Citado na página 27.

KOC, C. K. Montgomery reduction with even modulus. *IEE Proceedings - Computers and Digital Techniques*, [S.l.], v. 141, n. 2, p. 314–316, September 1994. Acesso em: 11 ago. 2013. Disponível em: <http://digital-library.theiet.org/content/journals/10.1049/ip-cdt_19941291>. Citado na página 49.

KOVALENKO, I.; KOCHUBINSKII, A. Asymmetric cryptographic algorithms. *Cybernetics and Systems Analysis*, New York, v. 39, n. 4, p. 549–554, 2003. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1023/B:CASA.0000003504.91987.d9>. Citado na página 27.

KOYAMA, K.; TSURUOKA, Y. Speeding up Elliptic Cryptosystems by using a Signed Binary Window Method. In: BRICKELL, E. (Ed.). *Advances in Cryptology: CRYPTO' 92*. [S.l.]: Springer Berlin Heidelberg, 1993. p. 345–357. (Lecture Notes in Computer Science, v. 740). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/3-540-48071-4_25>. Citado na página 36.

LAUE, R.; HUSS, S. Parallel memory architecture for Elliptic Curve Cryptography over $GF(p)$ aimed at efficient FPGA implementation. *Journal of Signal Processing Systems*, [S.l.], v. 51, n. 1, p. 39–55, 2008. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/s11265-007-0135-9>. Citado na página 57.

LAUTER, K. The advantages of Elliptic Curve Cyptography for wireless security. *Wireless Communications*, [S.l.], v. 11, n. 1, p. 62–67, 2004. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1109/MWC.2004.1269719>. Citado na página 27.

LAW, L. et al. An efficient protocol for authenticated key agreement. *Designs, Codes, and Cryptography*, Norwell, v. 28, n. 2, p. 119–134, March 2003. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1023/A:1022595222606>. Citado na página 31.

LÓPEZ, J.; DAHAB, R. High-speed software multiplication in $\mathbb{F}_{2m}$. In: ROY, B. K.; OKAMOTO, E. (Ed.). *INDOCRYPT*. [S.l.]: Springer, 2000. p. 203–212. (Lecture Notes in Computer Science v. 1977). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/3-540-44495-5_18>. Citado na página 136.

MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, New York, v. 8, n. 1, p. 3–30, January 1998. Acesso

em: 11 ago. 2013. Disponível em: <http://doi.acm.org/10.1145/272991.272995>. Citado na página 63.

MILLER, V. S. Use of elliptic curves in cryptography. In: WILLIAMS, H. (Ed.). *Advances in Cryptology: CRYPTO '85 Proceedings*. [S.l.]: Springer Berlin Heidelberg, 1986. p. 417–426. (Lecture Notes in Computer Science, v. 218). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/3-540-39799-X_31>. Citado na página 27.

MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of Computation*, Providence, v. 44, n. 170, p. 519–521, 1985. Citado 2 vezes nas páginas 49 e 51.

MONTGOMERY, P. L. Speeding the Pollard and Elliptic Curve methods of factorization. *Mathematics of Computation*, Providence, v. 48, n. 177, p. 243–264, 1987. Acesso em: 11 ago. 2013. Disponível em: <http://www.jstor.org/stable/2007888>. Citado na página 39.

OKEYA, K. et al. Signed Binary Representations Revisited. In: *Advances in Cryptology: CRYPTO 2004*. [S.l.]: Springer, 2004. p. 123–139. (Lecture Notes in Computer Science 3152). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/978-3-540-28628-8_8>. Citado 2 vezes nas páginas 36 e 37.

PACHECO, P. *An Introduction to Parallel Programming*. San Francisco: Morgan Kaufmann Publishers, 2011. Citado 2 vezes nas páginas 61 e 99.

REITWIESNER, G. W. Binary arithmetic. *Advances in Computers*, New York, v. 1, p. 231–308, 1960. Acesso em: 11 ago. 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/ac/ac1.html#Reitwiesner60>. Citado na página 35.

RIVEST, R. L. et al. Responses to NIST's proposal. *Communications of the ACM*, New York, v. 35, n. 7, p. 41–54, July 1992. Acesso em: 11 ago. 2013. Disponível em: <http://doi.acm.org/10.1145/129902.129905>. Citado na página 31.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, New York, v. 21, n. 2, p. 120–126, February 1978. Acesso em: 11 ago. 2013. Disponível em: <http://doi.acm.org/10.1145/359340.359342>. Citado na página 27.

SAKIYAMA, K. et al. Tripartite modular multiplication. *Integration*, [S.l.], v. 44, n. 4, p. 259–269, 2011. Acesso em: 11 ago. 2013. Disponível em: <http://dblp.uni-trier.de/db/journals/integration/integration44.html\#SakiyamaKFPV11>. Citado 3 vezes nas páginas 28, 50 e 54.

SCHNORR, C. P. Efficient identification and signatures for smart cards. In: *Advances in Cryptology: CRYPTO '89*. [S.l.]: Springer, 1989. p. 239–252. (Lecture Notes in Computer Science, v. 435). Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/0-387-34805-0_22>. Citado na página 31.

SCHROEPPEL, R. Elliptic Curves: twice as fast. 2000. (Presentation at the Crypto, v. 34). Citado na página 39.

STALLINGS, W. *Cryptography and network security: principles and practice*. 5th. ed. Upper Saddle River: Prentice Hall Press, 2010. Citado 4 vezes nas páginas 35, 40, 135 e 137.

TAVERNE, J. et al. Software implementation of binary Elliptic Curves: impact of the carry-less multiplier on Scalar Multiplication. In: *CRYPTOGRAPHIC Hardware and Embedded Systems: CHES 2011*. [S.l.]: Springer, 2011. p. 108–123. Acesso em: 11 ago. 2013. Disponível em: <http://dx.doi.org/10.1007/978-3-642-23951-9_8>. Citado na página 39.

APÊNDICE A – Experimentos MARIA

Tabela 27: Tempos (em μs) obtidos no cenário 2 (todas as *threads*) dos algoritmos modulares no modo de CPU *powersave*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**1,23**	**1,86**	**2,82**	**4,36**	7,92	11,97	23,41	38,74	78,27	129,79
	Barrett	1,38	2,23	3,53	5,98	10,15	16,74	30,29	50,53	92,05	143,12
	Montgomery	1,38	2,20	3,50	6,20	10,04	16,61	30,03	50,08	91,29	143,11
2	Montgomery	6,51	6,77	7,91	8,80	11,88	15,49	25,53	41,42	71,02	109,23
	Bipartite	2,88	3,68	4,01	5,32	6,97	10,20	18,01	30,55	54,24	84,83
3	Montgomery	5,98	6,41	6,42	7,80	9,56	11,88	17,36	25,47	40,59	63,23
	2-ary Multi. v1	4,99	5,15	5,85	6,17	7,53	10,67	15,71	23,08	38,79	60,18
	2-ary Multi. v2	3,50	4,18	4,27	5,27	**6,77**	9,82	15,91	25,47	45,44	71,59
4	Montgomery	6,20	6,47	6,51	7,58	9,56	12,71	17,36	25,18	40,44	62,64
	Bipartite	7,21	8,01	8,33	8,88	10,15	12,52	16,58	**22,44**	**35,56**	**54,65**
	4-ary Multi. v1	5,00	5,30	6,41	7,17	8,23	10,82	15,86	24,17	38,89	60,06
	4-ary Multi. v2	3,76	4,08	4,85	5,47	6,95	**8,92**	**14,39**	25,18	42,99	69,88

Tabela 28: Tempos (em μs) obtidos no cenário 2 (todas as *threads*) dos algoritmos modulares no modo de CPU *performance*. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**0,65**	**0,97**	**1,44**	**2,04**	4,00	6,03	11,74	19,41	39,17	64,88
	Barrett	0,73	1,15	1,80	3,03	5,12	8,39	15,20	25,32	46,08	71,77
	Montgomery	0,73	1,14	1,79	3,14	5,06	8,33	15,03	25,08	45,70	71,54
2	Montgomery	3,30	3,62	4,11	4,41	6,01	7,79	12,77	20,74	35,53	54,61
	Bipartite	1,59	1,92	2,20	2,73	3,50	5,15	9,04	15,35	27,18	42,39
3	Montgomery	3,03	3,24	3,32	3,86	4,86	6,06	8,71	12,68	20,33	31,59
	2-ary Multi. v1	2,53	2,61	2,97	3,12	3,80	5,41	7,94	11,61	19,46	30,15
	2-ary Multi. v2	1,77	2,14	2,17	2,70	**3,41**	4,91	7,96	12,79	22,71	35,79
4	Montgomery	3,14	3,27	3,32	3,85	4,80	6,42	8,71	12,67	20,27	31,36
	Bipartite	3,71	4,08	4,23	4,17	5,00	6,30	8,30	**11,26**	**17,80**	**27,39**
	4-ary Multi. v1	2,58	2,77	3,09	3,67	4,15	5,48	8,02	12,12	19,48	30,05
	4-ary Multi. v2	1,92	2,07	2,45	2,79	3,48	**4,48**	**7,24**	12,64	21,50	34,94

Tabela 29: Tempos (em μs) obtidos no cenário 2 (todas as *threads*) dos algoritmos modulares no modo de CPU *ondemand* em 10^5 execuções. Os menores tempos por operando estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	128	256	384	512	768	1024	1536	2048	3072	4096
1	GMP	**0,65**	**0,97**	**1,42**	**2,05**	3,73	6,03	11,73	19,39	39,15	64,86
	Barrett	0,73	1,14	1,80	3,01	5,11	8,39	15,17	25,29	46,05	70,67
	Montgomery	0,71	1,14	1,77	3,12	5,06	8,33	15,04	25,05	45,67	70,05
2	Montgomery	3,30	3,44	4,09	4,74	6,00	7,74	12,77	20,71	35,50	54,57
	Bipartite	1,44	1,86	1,99	2,68	3,50	5,03	8,98	15,26	26,92	42,42
3	Montgomery	2,79	3,02	3,27	3,59	4,91	6,08	8,80	12,73	20,36	31,64
	2-ary Multi. v1	2,42	2,59	2,91	3,07	3,77	5,39	7,95	11,56	19,44	30,12
	2-ary Multi. v2	1,82	2,12	2,18	2,71	**3,44**	4,95	7,95	12,77	22,77	35,86
4	Montgomery	2,95	3,23	3,27	3,73	4,71	6,24	8,67	12,56	20,12	31,17
	Bipartite	3,67	4,02	3,92	4,45	5,11	6,15	8,27	**11,09**	**17,62**	**27,27**
	4-ary Multi. v1	2,49	2,65	2,91	3,56	4,04	5,38	7,91	11,97	19,32	29,92
	4-ary Multi. v2	1,92	2,06	2,44	2,79	3,48	**4,50**	**7,21**	12,64	21,48	34,98

APÊNDICE B – Experimentos RELIC

Tabela 30: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Comba e modo de CPU *ondemand*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	7780,35	21138,56	49358,27	96021,58
	GMP	8364,56	20361,32	42210,15	66118,32
	Barrett	9072,74	21735,88	43598,86	68447,32
	Montgomery	**4786,47**	**12249,91**	27001,77	51556,18
2	Montgomery	8319,56	16763,44	31716,61	50509,30
	Bipartite	8297,29	17037,06	**24704,38**	45298,68
3	Montgomery	7161,38	15097,18	29333,09	48888,24
	2-ary Multi. v1	9227,95	18123,58	-	44739,85
	2-ary Multi. v2	8416,91	17194,08	25164,32	44729,76
4	Montgomery	7096,11	14828,65	29928,94	48231,71
	Bipartite	10709,38	20318,65	-	44876,74
	4-ary Multi. v1	9464,47	18448,24	-	**44567,65**

Tabela 31: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Multp e modo de CPU *ondemand*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	7760,99	21398,67	50154,86	97095,92
	GMP	**2655,14**	**5421,91**	**11038,14**	19295,30
	Barrett	3630,29	7411,50	13309,71	22616,55
	Montgomery	3023,46	6367,88	12340,82	22087,55
2	Montgomery	8446,88	13531,24	20415,08	20804,53
	Bipartite	9076,30	7569,52	12418,91	17788,97
3	Montgomery	6996,03	10899,82	15937,44	18121,77
	2-ary Multi. v1	5744,82	9058,97	-	16653,21
	2-ary Multi. v2	4604,71	7673,48	12316,10	16802,18
4	Montgomery	6757,62	10462,54	15403,12	17151,59
	Bipartite	8337,98	12827,67	-	16944,68
	4-ary Multi. v1	6032,74	9117,27	-	**16449,67**

Tabela 32: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Comba e modo de CPU *powersave*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	15557,91	42549,20	99479,86	194169,70
	GMP	16883,59	41058,88	85194,53	132199,39
	Barrett	18958,95	43653,45	87327,06	136604,14
	Montgomery	**9590,86**	**24702,88**	54087,61	103742,47
2	Montgomery	18300,35	33580,73	63755,29	102374,30
	Bipartite	16619,96	34173,29	**49682,18**	91827,27
3	Montgomery	14391,50	31046,06	58654,33	97778,85
	2-ary Multi. v1	18431,17	36430,33	-	**89329,17**
	2-ary Multi. v2	16883,27	34607,99	50708,00	89389,18
4	Montgomery	14094,12	30034,70	57844,73	97192,74
	Bipartite	22137,77	40613,83	-	89627,45
	4-ary Multi. v1	19168,06	36983,71	-	89439,12

Tabela 33: Tempos (em μs) da multiplicação escalar no RELIC, ao utilizar o modo Multp e modo de CPU *powersave*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	256	384	521	1536
1	Schoolbook	15517,82	43545,83	101650,23	194505,05
	GMP	**5288,94**	**11021,17**	**22120,42**	38561,89
	Barrett	7317,32	14966,29	26469,12	46111,83
	Montgomery	6075,52	12756,95	24812,07	44170,21
2	Montgomery	17784,47	27795,86	40791,51	42223,76
	Bipartite	8979,98	15058,35	24817,95	36163,41
3	Montgomery	13688,49	22009,48	31930,68	36223,47
	2-ary Multi. v1	11586,38	18290,30	-	**33331,59**
	2-ary Multi. v2	9299,94	15318,38	24733,15	33648,27
4	Montgomery	13515,61	21616,88	31630,98	34754,50
	Bipartite	16645,91	25587,50	-	34495,65
	4-ary Multi. v1	12387,73	18306,48	-	33709,03

Obs.: A precisão da mediana calculada é superior ao apresentado nas Tabelas 22 e 23.

APÊNDICE C – Aritmética modular RELIC (dados estatísticos)

Tabela 34: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Comba, chave de 521 bits, $q = 1102$ e $m = 1929$. Os menores tempos por coluna estão em negrito.

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)
1	Schoolbook	17293,11	30795,49	48088,60
	GMP	34756,92	**6744,20**	41501,12
	Barrett	34772,28	7923,76	42696,04
	Montgomery	18612,76	7348,86	**25961,62**
2	Montgomery	18620,70	13079,79	31700,49
	Bipartite	16805,83	9351,20	26157,04
3	Montgomery	18642,66	10774,24	29416,90
	2-ary Multi. v2	**16770,39**	9608,83	26379,22
4	Montgomery	18647,34	10465,27	29112,61

Tabela 35: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) da multiplicação modular no RELIC com chaves de 384 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	8,78	8,68	0,43	4,88	8,79	8,70	0,53	5,99
	GMP	**2,47**	**2,43**	0,22	9,03	**2,36**	**2,33**	0,11	4,86
	Barrett	3,65	3,61	0,22	5,90	3,43	3,41	0,13	3,86
	Montgomery	2,67	2,64	0,21	7,76	2,59	2,56	0,12	4,78
2	Montgomery	6,30	6,25	0,46	7,33	6,19	6,17	0,24	3,91
	Bipartite	4,39	4,08	0,78	17,75	3,82	3,79	0,46	12,16
3	Montgomery	5,27	5,23	0,26	5,03	5,14	5,12	0,14	2,77
	2-ary Multi. v1	5,16	4,89	0,73	14,21	4,56	4,55	0,17	3,63
	2-ary Multi. v2	4,45	4,15	0,75	16,90	3,85	3,83	0,25	6,41
4	Montgomery	5,00	4,96	0,21	4,19	4,94	4,92	0,23	4,68
	Bipartite	6,64	6,45	0,72	10,78	6,26	6,23	0,60	9,62
	4-ary Multi. v1	5,29	4,98	0,76	14,31	4,75	4,74	0,23	4,88

Tabela 36: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) do quadrado modular no RELIC com chaves de 384 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	**8,62**	**8,53**	0,48	5,56	8,82	8,73	0,60	6,82
	GMP	19,95	19,91	0,44	2,21	**2,36**	**2,33**	0,10	4,22
	Barrett	20,02	19,97	0,62	3,09	3,52	3,49	0,19	5,38
	Montgomery	9,19	9,15	0,16	1,70	2,66	2,64	0,14	5,13
2	Montgomery	9,23	9,18	0,16	1,71	2,36	6,18	0,38	15,95
	Bipartite	13,95	13,88	0,45	3,20	3,87	3,82	0,32	8,20
3	Montgomery	9,21	9,15	0,39	4,26	5,20	5,14	0,21	3,99
	2-ary Multi. v1	14,07	14,03	0,27	1,93	5,01	4,58	0,22	4,33
	2-ary Multi. v2	14,04	13,94	0,94	6,68	3,90	3,85	0,22	5,55
4	Montgomery	9,20	9,17	0,15	1,66	5,20	4,94	0,21	4,08
	Bipartite	13,93	13,85	0,57	4,12	6,31	6,26	0,23	3,67
	4-ary Multi. v1	14,03	13,98	0,51	3,63	3,90	4,77	0,21	5,46

Tabela 37: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Comba, chave de 384 bits, $q = 862$ e $m = 1509$.

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)
1	Schoolbook	**7432,34**	13246,21	20678,55
	GMP	17199,51	**3734,63**	20934,13
	Barrett	17258,46	5506,70	22765,16
	Montgomery	7921,72	4036,34	**11958,05**
2	Montgomery	7955,35	9508,42	17463,77
	Bipartite	12023,69	6617,41	18641,11
3	Montgomery	7935,28	7946,40	15881,68
	2-ary Multi. v1	12127,76	7787,14	19914,89
	2-ary Multi. v2	12100,66	6719,67	18820,33
4	Montgomery	7927,65	7543,84	15471,49
	Bipartite	12008,87	10017,97	22026,84
	4-ary Multi. v1	12089,78	7985,72	20075,50

Obs.: A precisão da mediana calculada é superior ao apresentado nas Tabelas 35 e 36.

Tabela 38: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Multp, chave de 384 bits, $q = 862$ e $m = 1509$.

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)
1	Schoolbook	7604,62	13266,51	20871,13
	GMP	2032,42	**3553,83**	**5586,25**
	Barrett	3034,10	5172,39	8206,48
	Montgomery	2293,98	3910,75	6204,74
2	Montgomery	**2032,42**	9333,37	11365,79
	Bipartite	3340,18	5763,45	9103,63
3	Montgomery	4479,82	7749,16	12228,97
	2-ary Multi. v1	4321,01	6882,32	11203,33
	2-ary Multi. v2	3358,55	5803,27	9161,83
4	Montgomery	4479,82	7455,37	11935,18
	Bipartite	5439,38	9447,78	14887,16
	4-ary Multi. v1	3358,55	7164,74	10523,29

Obs.: A precisão da mediana calculada é superior ao apresentado nas Tabelas 35 e 36.

Tabela 39: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) da multiplicação modular no RELIC com chaves de 1536 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	111,77	111,39	1,17	1,05	111,74	111,41	0,98	0,88
	GMP	13,35	13,30	0,23	1,74	13,22	13,20	0,18	1,33
	Barrett	18,19	18,11	0,64	3,52	18,06	18,02	0,27	1,50
	Montgomery	16,57	16,50	0,38	2,27	16,48	16,42	0,43	2,63
2	Montgomery	15,08	14,99	0,73	4,87	14,81	14,76	0,41	2,76
	Bipartite	11,68	11,30	1,10	9,46	11,02	10,98	0,78	7,08
3	Montgomery	11,34	11,29	0,24	2,08	11,29	11,21	0,60	5,30
	2-ary Multi. v1	10,48	10,09	1,29	12,35	9,75	9,73	0,21	2,13
	2-ary Multi. v2	10,62	10,20	1,27	11,93	9,95	9,93	0,20	2,01
4	Montgomery	**10,05**	**9,94**	0,85	8,48	9,94	9,91	0,17	1,71
	Bipartite	11,15	10,79	1,15	10,33	10,32	10,30	0,22	2,17
	4-ary Multi. v1	10,38	10,00	1,13	10,90	**9,57**	**9,56**	0,21	2,23

Tabela 40: Média ($\bar{t_m}$), mediana ($\tilde{t_m}$), desvio padrão ($\sigma(t_m)$) e coeficiente de variação (cv) do quadrado modular no RELIC com chaves de 1536 bits e modo de CPU *performance*. Os menores tempos por chave estão em negrito, e os menores tempos por *thread* estão com fundo cinza.

#	Algoritmo	Comba				Multp			
		$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)	$\bar{t_m}$	$\tilde{t_m}$	$\sigma(t_m)$	cv (%)
1	Schoolbook	108,69	108,33	1,21	1,12	15,94	15,85	0,60	2,18
	GMP	182,49	182,18	1,67	0,92	**3,29**	**3,24**	0,19	4,26
	Barrett	182,83	182,27	3,29	1,80	4,02	4,00	0,13	3,57
	Montgomery	119,71	119,24	1,83	1,53	3,74	3,70	0,13	3,78
2	Montgomery	119,64	119,27	1,21	1,01	6,78	6,71	0,23	5,27
	Bipartite	108,36	108,11	1,52	1,41	4,40	4,35	0,19	5,90
3	Montgomery	119,70	119,27	1,28	1,07	5,44	5,38	0,20	3,73
	2-ary Multi. v1	**107,63**	**107,47**	0,67	0,62	9,79	9,77	0,21	2,18
	2-ary Multi. v2	108,18	107,94	1,26	1,17	10,00	9,97	0,22	2,16
4	Montgomery	119,66	119,27	1,06	0,89	5,38	5,29	0,27	4,67
	Bipartite	108,58	108,26	1,46	1,35	4,42	4,36	0,21	4,63
	4-ary Multi. v1	107,82	107,56	1,37	1,27	9,72	9,62	0,75	7,70

Tabela 41: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Comba, chave de 1536 bits, $q = 282$ e $m = 494$.

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)
1	Schoolbook	30651,63	55212,10	85863,74
	GMP	51461,50	6596,67	58058,17
	Barrett	51559,41	8985,45	60544,85
	Montgomery	33758,13	8187,61	41945,74
2	Montgomery	33739,38	7449,51	41188,89
	Bipartite	30557,06	5768,39	36325,44
3	Montgomery	33754,86	5602,09	39356,95
	2-ary Multi. v1	**30353,07**	5176,08	**35529,15**
	2-ary Multi. v2	30507,50	5245,48	35752,98
4	Montgomery	33743,81	**4964,66**	38708,47
	Bipartite	30620,51	5508,60	36129,11
	4-ary Multi. v1	30405,37	5129,23	35534,60

Obs.: A precisão da mediana calculada é superior ao apresentado nas Tabelas 39 e 40.

Tabela 42: Estimativa (em μs) do tempo total da multiplicação e quadrado modulares presentes na multiplicação escalar com modo Multp, chave de 1536 bits, $q = 282$ e $m = 494$.

#	Algoritmo	Quadrado ($\bar{t_q} * q$)	Multiplicação ($\bar{t_m} * m$)	Total ($\bar{t_q} * q + \bar{t_m} * m$)
1	Schoolbook	4494,39	55199,04	59693,43
	GMP	**927,59**	6532,99	7460,58
	Barrett	1134,40	8923,96	10058,36
	Montgomery	1054,45	8142,31	9196,76
2	Montgomery	1912,38	7314,33	9226,71
	Bipartite	1240,79	5443,94	6684,72
3	Montgomery	1534,03	5574,85	7108,88
	2-ary Multi. v1	2761,09	4814,23	7575,32
	2-ary Multi. v2	2820,15	4914,99	7735,13
4	Montgomery	1516,11	4908,48	6424,59
	Bipartite	1245,04	5096,96	**6342,01**
	4-ary Multi. v1	2739,94	**4729,87**	7469,81

Obs.: A precisão da mediana calculada é superior ao apresentado nas Tabelas 39 e 40.

APÊNDICE D – Algoritmos adaptados

Neste APÊNDICE são apresentadas partes do código fonte (na linguagem C) dos algoritmos paralelos adaptados e integrados na biblioteca RELIC. Nos comentários, as letras gregas μ e ν foram representadas por u e v, respectivamente.

D.1 Montgomery (sequencial)

```
/* MONTGOMERY */

...

/* C = A*B */
mpz_mul(C, A, B);

/* m = (C mod r^n)u mod r^n */
C[0]._mp_size = FP_DIGS;
mpz_mul(_reg1, C, invP);
_reg1[0]._mp_size = FP_DIGS;

/* Z = (C + mM )/r^n */
mpz_mul(_reg2, _reg1, P);
C[0]._mp_d = &C[0]._mp_d[FP_DIGS];
_reg2[0]._mp_d = &_reg2[0]._mp_d[FP_DIGS];
_reg2[0]._mp_size = FP_DIGS;
mpz_add(C, C, _reg2);
mpz_add_ui(C, C, 1);

/* se Z >= M entao Z = Z - M fim se */
if(mpz_cmp(C, P) >= 0)
   mpz_sub(C, C, P);

...
```

D.2 Barrett (sequencial)

```
/* BARRETT */

...

/* C = A*B */
mpz_mul(C, A, B);

/* Q = Cv/r^{n-1} / r^{n+1} */
mpz_tdiv_q_2exp(l, C, FP_BITS);
mpz_mul(_reg1, l, divp);
mpz_tdiv_q_2exp(_reg1, _reg1, FP_BITS);

/* Z = C - QM */
mpz_mul(_reg2, _reg1, P);
mpz_sub(C, C, _reg2);

/* enquanto Z >= M faca Z = Z - M fim enquanto */
while(mpz_cmp(C, P) >= 0)
   mpz_sub(C, C, P);

...
```

D.3 Bipartite (2 *threads*)

```
/* BIPARTITE */

...

#pragma omp parallel num_threads(2)
{
   if (!omp_get_thread_num()) { /* Z[0] = RPM */
      // C = A*B0
      mpn_mul(_r00, a, FP_DIGS, b, FP_DIGS - (FP_DIGS >> 1));

      /* Q = (C mod r^t )u mod r^t */
      mpn_mul_n(_r01, fp_maria_monty_get(), _r00, FP_DIGS - (FP_DIGS >> 1));
```

```
    /* Z = (C + QM )/r^t */
    mpn_mul(_r02, _p, FP_DIGS, _r01, FP_DIGS - (FP_DIGS >> 1));
    C[0]._mp_d[FP_DIGS] = mpn_add_n(C[0]._mp_d, &_r00[_q], &_r02[_q],
        FP_DIGS);
    C[0]._mp_d[FP_DIGS] += mpn_add_1(C[0]._mp_d, C[0]._mp_d, FP_DIGS, 1);

    /* se Z >= max(r^{m-t}, r^n) entao Z = Z - M fim se */
    if (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
       mpn_sub(C[0]._mp_d, C[0]._mp_d, FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 :
           0), _p,
             FP_DIGS);
    }

    C[0]._mp_size = FP_DIGS+1;

  } else { /* Z[1] = RPB */

    /* C = A*B1 */
    mpn_mul(_r03, a, FP_DIGS, &b[_q], FP_DIGS >> 1);

    /* Q = C1v / r^t / r^{m-n-t} */
    q_m = FP_BITS >> bits_per_limb_p;
    q_b = FP_BITS - (q_m << bits_per_limb_p);

    j = &_r03[q_m];

    if (q_b != 0)
       mpn_rshift (k, j, FP_DIGS+(FP_DIGS >> 1)-q_m, q_b);
    else
       k = &j[0];

    mpn_mul(_r04, fp_maria_barret2_get(), (FP_DIGS >> 1)+1, k,(FP_DIGS >>
        1));

    q_m = (FP_BITS >> 1) >> bits_per_limb_p;
    q_b = (FP_BITS >> 1) - (q_m << bits_per_limb_p);

    _r04 = &_r04[q_m];

    if (q_b != 0)
       mpn_rshift (_r04, _r04, FP_DIGS-q_m, q_b);
```

```
      /* Z = C - QM */
      mpn_mul(_r05, _p, FP_DIGS, _r04,(FP_DIGS >> 1));
      mpn_sub(_r04, _r03, FP_DIGS+(FP_DIGS >> 1), _r05, FP_DIGS+(FP_DIGS >>
          1));

      /* enquanto Z >= M faca Z = Z - M fim enquanto */
      while (_r04[FP_DIGS] || mpn_cmp(_r04, _p, FP_DIGS) >= 0) {
         mpn_sub(_r04, _r04, FP_DIGS + (_r04[FP_DIGS] ? 1 : 0), _p,
              FP_DIGS);
      }
   }
}

/* Z = (Z[0] + Z[1] ) */
mpn_add(C[0]._mp_d, C[0]._mp_d, FP_DIGS+1, _r04, FP_DIGS);

/* se Z >= M entao Z = Z - M fim se */
if (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
   mpn_sub(C[0]._mp_d, C[0]._mp_d,
         FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 : 0), _p, FP_DIGS);
}

...
```

D.4 2-ary Multi. v2

```
/* MULTIPARTITEV2 */

...

#pragma omp parallel num_threads(3)
{

   if(!omp_get_thread_num()){ /* Z[0] = RPM */

      /* C = A0*B0 */
      mpn_mul_n(C[0]._mp_d, a, b, _q);
```

```
    /* Q = (C mod r^t )u mod r^t */
    mpn_mul_n(_R01, C[0]._mp_d, fp_maria_monty_get(), _q);

    /* Z = (C + QM )/r^t */
    mpn_mul(_r02, _p, FP_DIGS, _R01, _q);
    _R01[FP_DIGS] = mpn_add(_R01, &_r02[_q], FP_DIGS, &C[0]._mp_d[_q], _q);
    _R01[FP_DIGS] = mpn_add_1(_R01, _R01, FP_DIGS, 1);

    /* se Z >= max(r^{m-t}, r^n) entao Z = Z - M fim se */
    if(_R01[FP_DIGS])
       mpn_sub(_R01, _R01, FP_DIGS+1, _p, FP_DIGS);
    else if( mpn_cmp(_R01, _p, FP_DIGS) > 0)
       mpn_sub_n(_R01, _R01, _p, FP_DIGS);

 }else
    if (omp_get_thread_num()==1) /* Z[1] = A0B1 + A1B0 */
    {
       /* A0B1 + A1B0 */
       mpn_mul(_r10, a, _q, &b[_q], _l);
       mpn_mul(_r11, b, _q, &a[_q], _l);
       _R12[_q + _l] = mpn_add_n(_R12, _r10, _r11, _q + _l);
    }else{ /* Z[2] = RPB */

       /* C = A1B1r^t */
       mpn_mul_n(_r03[0]._mp_d, &a[_q], &b[_q], FP_DIGS >> 1);
       _r03[0]._mp_size = (FP_DIGS >> 1)<<1;
       mpz_mul_2exp(_r03, _r03, (_q)*GMP_LIMB_BITS);

          /* Q = C1v / r^t / r^{m-n-t} */
       q_m = FP_BITS >> bits_per_limb_p;
       q_b = FP_BITS - (q_m << bits_per_limb_p);

       j = &_r03[0]._mp_d[q_m];

       if (q_b != 0)
          mpn_rshift (k, j, (((FP_DIGS >> 1)<<1) + _q)-q_m, q_b);
       else
          k = &j[0];

       mpn_mul(_r04, fp_maria_barret2_get(), (FP_DIGS >> 1)+1, k,(FP_DIGS >>
           1));
```

```
        q_m = (FP_BITS >> 1) >> bits_per_limb_p;
        q_b = (FP_BITS >> 1) - (q_m << bits_per_limb_p);

        _r04 = &_r04[q_m];
        if (q_b != 0)
           mpn_rshift (_r04, _r04, FP_DIGS-q_m, q_b);

           /* Z = C - QM */
        mpn_mul(_r05, _p, FP_DIGS, _r04, (FP_DIGS >> 1));
        mpn_sub(_r05, _r03[0]._mp_d, FP_DIGS+(FP_DIGS >> 1), _r05,
            FP_DIGS+(FP_DIGS >> 1));

        /* enquanto Z >= M faca Z = Z - M fim enquanto */
        while(_r05[FP_DIGS] || mpn_cmp(_r05, _p, FP_DIGS) >= 0) {
           mpn_sub(_r05, _r05, FP_DIGS+(_r05[FP_DIGS] ? 1 : 0), _p, FP_DIGS);
        }

     }
}

/* Z[0] + Z[1] + Z[2] */
C[0]._mp_d[(_q + _l + (_R12[_q + _l]?1:0))] = mpn_add(C[0]._mp_d, _R01,
    (FP_DIGS + (_R01[FP_DIGS]?1:0)), _R12, (_q + _l + (_R12[_q + _l]?1:0)));
C[0]._mp_size = (FP_DIGS + (_R01[FP_DIGS]?1:0)) + (_q + _l + (_R12[_q +
    _l]?1:0)) + (C[0]._mp_d[(_q + _l + (_R12[_q + _l]?1:0))]?1:0);

C[0]._mp_d[C[0]._mp_size] = mpn_add(C[0]._mp_d, C[0]._mp_d, C[0]._mp_size,
    _r05, FP_DIGS);
C[0]._mp_size = C[0]._mp_size + (C[0]._mp_d[C[0]._mp_size]?1:0);

/* enquanto Z >= M faca Z = Z - M fim enquanto */
while (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
   mpn_sub(C[0]._mp_d, C[0]._mp_d, FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 : 0),
       _p, FP_DIGS);
}

...
```

D.5 Montgomery 2 *threads*

```
/* MONTGOMERYPARALELO */

...

#pragma omp parallel num_threads(2)
{

   if(!omp_get_thread_num()) {

       /* C = A*B */
      fp_mult0_parallel2(_r00, a, b, FP_DIGS, _xr00, _xr01);

      /* m = (C mod r^n)u mod r^n */
      fp_mult0_parallel2(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00,
          _xr01);

      /* Z = (C + mM )/r^n */
      fp_mult0_parallel2(_r02, _p, _r01, FP_DIGS, _xr00, _xr01);
      C[0]._mp_d[FP_DIGS] = mpn_add_n(C[0]._mp_d, &_r00[FP_DIGS],
           &_r02[FP_DIGS], FP_DIGS);
      C[0]._mp_d[FP_DIGS] += mpn_add_1(C[0]._mp_d, C[0]._mp_d, FP_DIGS,
           1);

   } else {

      /* C = A*B */
      fp_mult1_parallel2(_r00, a, b, FP_DIGS, _xr00, _xr01);

      /* m = (C mod r^n)u mod r^n */
      fp_mult1_parallel2(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00,
          _xr01);

      /* mM */
      fp_mult1_parallel2(_r02, _p, _r01, FP_DIGS, _xr00, _xr01);

   }
}
```

```
/* se Z >= M entao Z = Z - M fim se */
if (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
   mpn_sub(C[0]._mp_d, C[0]._mp_d,
        FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 : 0), _p, FP_DIGS);
}

...

/* MULTIPLICACAOPARALELA */

void fp_mult0_parallel2(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t* B,
    unsigned int _n, mp_limb_t *_r00, mp_limb_t *_r01) {

   unsigned int _l;
   _l = _n >> 1;

    /* C = A*B0 */
   mpn_mul(_r00, A, _n, B, _l);
#pragma omp barrier

#pragma omp barrier
   mpn_add(&C[_l], _r01, _n + (_n - _l), &_r00[_l], _n);
#pragma omp barrier

}

void fp_mult1_parallel2(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t* B,
    unsigned int _n, mp_limb_t *_r00, mp_limb_t *_r01) {

   unsigned int _l;
   _l = _n >> 1;

   /* C = A*B1 */
   mpn_mul(_r01, A, _n, &B[_l], (_n - _l));
#pragma omp barrier
   mpn_copyi(C, _r00, _l);
#pragma omp barrier

#pragma omp barrier
}
```

D.6 Montgomery 3 *threads*

```
/* MONTGOMERYPARALELO */

...

#pragma omp parallel num_threads(3)
{

   if(!omp_get_thread_num()) {
      /* C = A*B */
      fp_mult0_parallel3(_r00, a, b, FP_DIGS, _xr00);

      /* m = (C mod r^n)u mod r^n */
      fp_mult0_parallel3(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00);

      /* Z = (C + mM )/r^n */
      fp_mult0_parallel3(_r02, _p, _r01, FP_DIGS, _xr00);
      C[0]._mp_d[FP_DIGS] = mpn_add_n(C[0]._mp_d, &_r00[FP_DIGS],
            &_r02[FP_DIGS], FP_DIGS);
      C[0]._mp_d[FP_DIGS] += mpn_add_1(C[0]._mp_d, C[0]._mp_d, FP_DIGS,
            1);

   } else
      if (omp_get_thread_num() == 1) {
         /* C = A*B */
         fp_mult1_parallel3(_r00, a, b, FP_DIGS, _xr00, _xr01, _xr02);

         /* m = (C mod r^n)u mod r^n */
         fp_mult1_parallel3(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00,
             _xr01, _xr02);

         /* mM */
         fp_mult1_parallel3(_r02, _p, _r01, FP_DIGS, _xr00, _xr01, _xr02);

      } else
         if (omp_get_thread_num() == 2) {

         /* C = A*B */
```

```
        fp_mult2_parallel3(_r00, a, b, FP_DIGS, _xr02);

        /* m = (C mod r^n)u mod r^n */
        fp_mult2_parallel3(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr02);
        /* mM */
         fp_mult2_parallel3(_r02, _p, _r01, FP_DIGS, _xr02);
        }

}

/* se Z >= M entao Z = Z - M fim se */
if (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
   mpn_sub(C[0]._mp_d, C[0]._mp_d,
        FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 : 0), _p, FP_DIGS);
}

...

/* MULTIPLICACAOPARALELA */

/* Montgomery 3 parallel modular multiplication */

  void fp_mult0_parallel3(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
     B, unsigned int _n, mp_limb_t *_r00) {

  unsigned int s1 = (_n % 3 ? (_n / 3) : _n/3);

    /* C = A*B0 */
    mpn_mul(_r00, A, _n, B, s1);
#pragma omp barrier
    mpn_copyi(C, _r00, s1);
#pragma omp barrier
  }

  void fp_mult1_parallel3(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
     B, unsigned int _n, mp_limb_t *_r00, mp_limb_t *_r01, mp_limb_t *_r02) {

  unsigned int s1 = (_n % 3 ? (_n / 3) : _n/3);
  unsigned int p2 = s1 << 1;
  unsigned int s2 = (_n % 3 ? _n - p2 : s1);

    /* C = A*B1 */
```

```
    mpn_mul(_r01, A, _n, &B[s1], s1);
#pragma omp barrier

    mpn_add(&C[s1], _r01, _n+s1, &_r00[s1], _n);
    mpn_add(&C[p2], _r02, _n+s2, &C[p2], _n);
#pragma omp barrier
  }

  void fp_mult2_parallel3(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
      B, unsigned int _n, mp_limb_t *_r02) {

  unsigned int s1 = (_n % 3 ? (_n / 3) : _n/3);
  unsigned int p2 = s1 << 1;
  unsigned int s2 = (_n % 3 ? _n - p2 : s1);

    /* C = A*B2 */
    mpn_mul(_r02, A, _n,&B[p2], s2);
#pragma omp barrier

#pragma omp barrier
  }
```

D.7 Montgomery 4 *threads*

```
/* MONTGOMERYPARALELO */

...

#pragma omp parallel num_threads(4)
{

   if(!omp_get_thread_num()) {

      /* C = A*B */
      fp_mult0_parallel4(_r00, a, b, FP_DIGS, _xr00);

      /* m = (C mod r^n)u mod r^n */
      fp_mult0_parallel4(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00);
```

```
    /* Z = (C + mM )/r^n */
    fp_mult0_parallel4(_r02, _p, _r01, FP_DIGS, _xr00);
    C[0]._mp_d[FP_DIGS] = mpn_add_n(C[0]._mp_d, &_r00[FP_DIGS],
         &_r02[FP_DIGS], FP_DIGS);
    C[0]._mp_d[FP_DIGS] += mpn_add_1(C[0]._mp_d, C[0]._mp_d, FP_DIGS,
         1);

} else
    if (omp_get_thread_num() == 1) {
       /* C = A*B */
       fp_mult1_parallel4(_r00, a, b, FP_DIGS, _xr00);

       /* m = (C mod r^n)u mod r^n */
       fp_mult1_parallel4(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00);

       /* mM */
       fp_mult1_parallel4(_r02, _p, _r01, FP_DIGS, _xr00);

    } else
       if (omp_get_thread_num() == 2) {

       /* C = A*B */
       fp_mult2_parallel4(_r00, a, b, FP_DIGS, _xr00, _xr01, _xr02);

       /* m = (C mod r^n)u mod r^n */
       fp_mult2_parallel4(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr00,
           _xr01, _xr02);

       /* mM */
       fp_mult2_parallel4(_r02, _p, _r01, FP_DIGS, _xr00, _xr01, _xr02);

       } else
          if (omp_get_thread_num() == 3) {

       /* C = A*B */
       fp_mult3_parallel4(_r00, a, b, FP_DIGS, _xr02);

          /* m = (C mod r^n)u mod r^n */
       fp_mult3_parallel4(_r01, fp_maria_monty_get(), _r00, FP_DIGS, _xr02);

       /* mM */
```

```
        fp_mult3_parallel4(_r02, _p, _r01, FP_DIGS, _xr02);

          }
}

/* se Z >= M entao Z = Z - M fim se */
if (C[0]._mp_d[FP_DIGS] || mpn_cmp(C[0]._mp_d, _p, FP_DIGS) >= 0) {
   mpn_sub(C[0]._mp_d, C[0]._mp_d,
        FP_DIGS + (C[0]._mp_d[FP_DIGS] ? 1 : 0), _p, FP_DIGS);
}

...

/* MULTIPLICACAOPARALELA */

  void fp_mult0_parallel4(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
     B, unsigned int _n, mp_limb_t *_r00) {

    unsigned int _l;
    _l = _n >> 1;

    /* C = A0*B0 */
    mpn_mul_n(_r00, A, B, _l);
#pragma omp barrier
    mpn_copyi(C, _r00, _l);
#pragma omp barrier

  }


  void fp_mult1_parallel4(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
     B, unsigned int _n, mp_limb_t *_r00) {

    unsigned int _l;
    _l = _n >> 1;

    /* C = A1*B1 */
    mpn_mul_n(&_r00[_l << 1], &A[_l], &B[_l], _n-_l);
#pragma omp barrier

#pragma omp barrier
  }
```

```
  void fp_mult2_parallel4(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
      B, unsigned int _n, mp_limb_t *_r00, mp_limb_t *_r01, mp_limb_t *_r02) {

    unsigned int _l;
    _l = _n >> 1;

    /* C = A1*B0 */
    mpn_mul(_r01, &A[_l], _n-_l, B, _l);
#pragma omp barrier
    _r01[_n] = mpn_add_n(_r01, _r01, _r02, _n);
    mpn_add(&C[_l], &_r00[_l], 2*_n-_l, _r01, _n+(_r01[_n] ? 1 : 0));
#pragma omp barrier
  }

  void fp_mult3_parallel4(mp_limb_t* C, const mp_limb_t* A, const mp_limb_t*
      B, unsigned int _n, mp_limb_t *_r02) {

    unsigned int _l;
    _l = _n >> 1;

    /* C = A0*B1 */
    mpn_mul(_r02, A, _l, &B[_l], _n-_l);
#pragma omp barrier

#pragma omp barrier
  }
```

ANEXO A – Teoria dos Conjuntos

A.1 Grupos

Um grupo G é um conjunto não vazio com uma lei de composição $\bullet$[1], às vezes denotado por $\{G, \bullet\}$ e possui as seguintes propriedades (STALLINGS, 2010):

- Fechamento: se $x, y \in G$ então $x \bullet y \in G$.
- Associatividade: se $x, y, z \in G$ então $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.
- Identidade: existe um elemento $e \in G$, tal que para todo $x \in G$ temos $x \bullet e = e \bullet x = x$.
- Inverso: para todo $x \in G$ existe um elemento $y \in G$, inverso de x, tal que $x \bullet y = y \bullet x = e$.

O grupo G é comutativo (ou abeliano) se satisfaz a seguinte propriedade:

- Comutatividade: $x \bullet y = y \bullet x$ para todo $x, y \in G$.

A.2 Anéis

Um anel A, às vezes denotado por $\{A, +, \times\}$, é um conjunto com duas leis de composição $+$ (adição) e $\times$ (multiplicação), tal que para $x, y, z \in A$, possui as seguintes propriedades (STALLINGS, 2010):

- A é um grupo comutativo (ou abeliano) com relação a $+$. Em um grupo aditivo, denota-se o elemento identidade como 0 e o inverso de x como $-x$.
- Fechamento em $\times$: se $x, y \in A$, então $x \times y \in A$.
- Associatividade em $\times$: $x \times (y \times z) = (x \times y) \times z$.
- Leis distributivas: $x \times (y + z) = x \times y + x \times z$ e $(x + y) \times z = x \times z + y \times z$.

O anel A é comutativo se satisfaz a seguinte propriedade:

- Comutatividade em $\times$: $x \times y = y \times x$ para todo $x, y \in A$.

[1] O operador $\bullet$ é genérico e pode se referir a qualquer operação matemática.

Um domínio integral é um anel comutativo que possui as seguintes propriedades:

- Identidade multiplicativa: exite um elemento $1 \in A$, tal que $x \times 1 = 1 \times x = x$, para todo $x \in A$
- Nenhum divisor zero: Se $x, y \in A$ e $x \times y = 0$, então $x = 0$ ou $y = 0$.

A.3 Corpos Finitos

Um corpo finito K é um anel comutativo, tal que todo elemento diferente de zero possui inverso. Um corpo é dito ser finito se possui um número finito de elementos.

A.3.1 Corpos Finitos Primos

Para qualquer inteiro a e primo p, a redução a mod p denota o único resto inteiro r, tal que $0 \leq r \leq p-1$ é obtido através da divisão de a por p. Um corpo finito de ordem p (denotado por $GF(p)$) é composto pelo conjunto de inteiros: $\{0, 1, 2, ..., p-1\}$ com adição e multiplicação realizadas módulo p. (HANKERSON; MENEZES; VANSTONE, 2003).

A.3.2 Corpos Finitos Binários

Segundo Hankerson et. al (HANKERSON; MENEZES; VANSTONE, 2003), os corpos finitos de ordem 2^m são chamados corpos binários ou corpos finitos característica dois. É possível construir $GF(2^m)$ utilizando a representação de *base polinomial*, onde os elementos são polinômios binários (com coeficientes em $GF(2) = \{0, 1\}$), conforme a Equação A.1 (LÓPEZ; DAHAB, 2000):

$$GF(2^m) = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + ... + a_2z^2 + a_1z^1 + a_0 : a_i \in \{0, 1\}\} \quad \text{(A.1)}$$

Seja $f(z)$ um polinômio binário irredutível de grau m. Por ser irredutível, $f(z)$ não pode ser fatorado como um produto de polinômios com grau menor que m. A adição de elementos é realizada com aritmética de coeficiente módulo 2. A multiplicação é realizada módulo o polinômio de redução $f(z)$. Seja $a(z)$ um polinômio binário, a redução módulo $f(z)$ é dada por $r(z) = a(z)$ mod $f(z)$, onde $r(z)$ é um polinômio de resto, cujo grau é menor que m, obtido após uma divisão de $a(z)$ por $f(z)$ (HANKERSON; MENEZES; VANSTONE, 2003).

A.3.3 Corpos de Extensão

Segundo Hankerson et. al. (HANKERSON; MENEZES; VANSTONE, 2003), a representação em base polinomial pode ser generalizada a todos os corpos de extensão.

Seja p primo e $m \geq 2$, $GF(p)[z]$ representa o conjunto de todos os polinômios em z com coeficientes de $GF(p)$, onde $f(z)$ é o polinômio (irredutível) de redução de grau m em $GF(p)$. Por sua irredutibilidade, $f(z)$ não pode ser fatorado como um produto de polinômios em $GF(p)[z]$. Os elementos em $GF(p^m)$ são os polinômios em $GF(p)[z]$ de grau até $m-1$.

$$GF(p^m) = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + ... + a_2z^2 + a_1z^1 + a_0 : a_i \in GF(p)\}$$

A.4 Aritmética modular

A aritmética modular é necessária para o cálculo as operações realizadas dentro de um corpo finito primo (Seção A.3.1). Todas as operações neste corpo são realizadas módulo um inteiro primo p. Seja p um inteiro positivo e a um inteiro qualquer, o quociente q e o resto (resíduo) r são obtidos a partir da divisão de a por p, tal que:

$$a = qp + r \quad 0 \leq r < p; q = \left\lfloor \frac{a}{p} \right\rfloor \tag{A.2}$$

Dois inteiros a e b são ditos congruentes módulo p se $(a \bmod p) = (b \bmod p)$ e são representados na forma: $a \equiv b \bmod p$. As seguintes propriedades se aplicam à congruência:

1. $a \equiv b \bmod p$ se $p|(a\ b)$ (lê-se p é um divisor de a e b).
2. $(a \bmod p) = (b \bmod p)$ implica $a \equiv b \bmod p$.
3. $a \equiv b \bmod p$ implica $b \equiv a \bmod p$.
4. $a \equiv b \bmod p$ e $b \equiv c \bmod p$ implica $a \equiv c \bmod p$.

A.4.1 Operações da aritmética modular

As seguintes operações podem ser realizadas com aritmética modular:

1. $[(a \bmod p) + (b \bmod p)] \bmod p = (a + b) \bmod p$
2. $[(a \bmod p) - (b \bmod p)] \bmod p = (a - b) \bmod p$
3. $[(a \bmod p) \times (b \bmod p)] \bmod p = (a \times b) \bmod p$

A prova dessas propriedades é encontrada em (STALLINGS, 2010).

A.4.2 Propriedades da aritmética modular

Define-se o conjunto $\mathbb{Z}_p$ como de inteiros não negativos menores que p:

$$\mathbb{Z}_p = (0, 1, ..., p-1) \tag{A.3}$$

Tabela 43: Propriedades da aritmética modular em $\mathbb{Z}_p$

Propriedade	Expressão
Comutativa	$(w + x) \bmod p = (x + w) \bmod p$ $(w \times x) \bmod p = (x \times w) \bmod p$
Associativa	$[(w + x) + y] \bmod p = [w + (x + y)] \bmod p$ $[(w \times x) \times y] \bmod p = [w \times (x \times y)] \bmod p$
Distributiva	$[w \times (x + y)] \bmod p =$ $[(w \times x) + (w \times y)] \bmod p$
Identidades	$(0 + w) \bmod p = w \bmod p$ $(1 \times w) \bmod p = w \bmod p$
Aditiva	Para cada $w \in \mathbb{Z}_p$, existe um z, tal que $w + z \equiv 0 \bmod p$

Se $(a + b) \equiv (a + c) \bmod p$, então $b \equiv c \bmod p$. No entanto, a seguinte afirmação é válida apenas com a condição incluída: Se $(a \times b) \equiv (a \times c) \bmod p$, então $b \equiv c \bmod p$ **se** a **é relativamente primo a** p ($\text{mdc}(a, p) = 1$).

ANEXO B – Custos dos sistemas de coordenadas

B.1 Sistemas de coordenadas sobre $GF(p)$

Tabela 44: Número de operações para adição e dobro de ponto em curvas $y^2 = x^3 - 3x + b$, onde I = quantidade de inversões, M = multiplicação, S = dobro.

Dobro		Adição		Adição mista	
Operação	Custo	Operação	Custo	Operação	Custo
$2\mathcal{A}$	$1I + 2M + 2S$	$\mathcal{A} + \mathcal{A}$	$1I + 2M + 1S$	$\mathcal{J} + \mathcal{A} = \mathcal{J}$	$8M + 3S$
$2\mathcal{P}$	$7M + 3S$	$\mathcal{P} + \mathcal{P}$	$12M + 2S$	$\mathcal{J} + \mathcal{J}^c = \mathcal{J}$	$11M + 3S$
$2\mathcal{J}$	$4M + 4S$	$\mathcal{J} + \mathcal{J}$	$12M + 4S$	$\mathcal{J}^c + \mathcal{A} = \mathcal{J}^c$	$8M + 3S$
$2\mathcal{J}^c$	$5M + 4S$	$\mathcal{J}^c + \mathcal{J}^c$	$11M + 3S$	-	-

Fonte: (HANKERSON; MENEZES; VANSTONE, 2003)

B.2 Sistemas de coordenadas sobre $GF(2^m)$

Tabela 45: Operações necessárias para adição e dobro em $GF(2^m)$.

Dobro		Adição	
Operação	Custo	Operação	Custo
$2\mathcal{P}$	$7M + 4S + M_2$	$\mathcal{J} + \mathcal{J}$	$15M + 3S + M_2$
$2\mathcal{J}$	$5M + 5S$	$\mathcal{P} + \mathcal{P}$	$15M + 2S + M_2$
$2\mathcal{LD}$	$4M + 4S + M_2$	$\mathcal{LD} + \mathcal{LD}$	$13M + 4S$
$2\mathcal{A} = P$	$5M + 2S + M_2$	$\mathcal{P} + \mathcal{A} = \mathcal{P}$	$11M + 2S + M_2$
$2\mathcal{A} = \mathcal{LD}$	$2M + 3S + M_2$	$\mathcal{J} + \mathcal{A} = \mathcal{J}$	$10M + 3S + M_2$
$2\mathcal{A} = \mathcal{J}$	$M + 2S + M_2$	$\mathcal{LD} + \mathcal{A} = \mathcal{LD}$	$8M + 5S + M_2$
-	-	$\mathcal{A} + \mathcal{A} = \mathcal{LD}$	$5M + 2S + M_2$
$2A$	$I + 2M + S$	$\mathcal{A} + \mathcal{A} = \mathcal{J}$	$4M + S + M_2$
$2\mathcal{A}'$	$I + M + S$	$\mathcal{A} + \mathcal{A} = \mathcal{A}'$	$2I + 3M + S$
$2\mathcal{A}' = \mathcal{A}$	$M + 2S$	$\mathcal{A} + \mathcal{A}$	$I + 2M + S$

Fonte: (COHEN et al., 2006)

Nota: A quantidade de multiplicações pelo coeficiente c é denotada por M_2.