Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação eficiente de emparelhamentos bilineares sobre curvas elípticas na plataforma ARM

Victor Henrique Hisao Taira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Diego de Freitas Aranha

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Maristela Terto de Holanda

Banca examinadora composta por:

Prof. Dr. Diego de Freitas Aranha (Orientador) — CIC/UnB
Prof. Pedro Antônio Dourado de Rezende — CIC/UnB
Prof. João José Costa Gondim — CIC/UnB

CIP — Catalogação Internacional na Publicação

Taira, Victor Henrique Hisao.

Implementação eficiente de emparelhamentos bilineares sobre curvas elípticas na plataforma ARM / Victor Henrique Hisao Taira. Brasília : UnB, 2013.
149 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. Emparelhamentos bilineares, 2. Criptografia Assimétrica,
3. Segurança Computacional, 4. Implementações Eficientes de Primitivas Criptográficas

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação eficiente de emparelhamentos bilineares sobre curvas elípticas na plataforma ARM

Victor Henrique Hisao Taira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Diego de Freitas Aranha (Orientador)
CIC/UnB

Prof. Pedro Antônio Dourado de Rezende
CIC/UnB

Prof. João José Costa Gondim
CIC/UnB

Prof. Dr. Maristela Terto de Holanda
Coordenador do Bacharelado em Ciência da Computação

Brasília, 8 de março de 2013

Dedicatória

Aos meus pais, Suely Miyako Uchida Taira e Nivaldo Taira aos meus avós maternos, Kentaro Uchida e Emiko Uchida e aos avós paternos Tsutomu Taira e Tizuko Taira.

Agradecimentos

Gostaria de primeiramente agradecer aos meus pais, Suely Miyako Uchida Taira e Nivaldo Taira, aos meus avós maternos, Kentaro Uchida e Emiko Uchida e aos avós paternos Tsutomu Taira e Tizuko Taira por sempre prestarem o suporte necessário em todas as fases da minha vida. E todos as demais pessoas da minha família, obrigado a todos.

Agradeço também a todos os meus professores, pois cada um contribui à sua maneira na minha formação acadêmica. Um agradecimento especial para as professoras Maria Emilia Machado Telles Walter e Célia Ghedini Ralha e ao professor Pedro de Azevedo Berger por me darem a oportunidade de trabalhar com o projeto de iniciação científica. Agradeço também ao professor Pedro Antônio Dourado de Rezende que soube transmitir de forma brilhante seu conhecimento em criptografia.

Gostaria de agradecer também a todos os meus amigos no trabalho que sempre foram pacientes e prestativos em todos os momentos, entendendo inclusive os momentos de aperto que a vida universitária proporciona.

Agradeço aos meus colegas e amigos da UnB, tanto os amigos da CIC quanto os dos demais cursos. Em especial para os meus amigos Getúlio Yassuyuki Matayoshi e Vinícius Tafuri Froes de Carvalho pelas noites em claro, seja estudando seja realizando algum trabalho.

Particularmente, gostaria de agradecer também a todos os meus amigos pela companhia em momentos de distração. Em especial ao amigos Thiago Amorim Onuki e Camila Silva Nishikawa sempre presentes seja em momentos de alegria seja em momentos de tristeza. Queria agradecer também ao amigos do LoL, que sempre proporcionaram momentos únicos de diversão, em especial aos primos da Kiki e a Kiki.

Gostaria de agradecer à minha namorada Aline Yoshie Uemura pela sua forma única de transmitir carinho e por sempre providenciar momentos ímpares. Queria agradecer também a sua família pelo incrível acolhimento.

Finalmente, gostaria de agradecer ao meu grande orientador, professor Diego de Freitas Aranha pelos inúmeros conselhos e por sempre estar disponível para ajudar a traçar os caminhos necessários a realização deste trabalho.

Resumo

Algoritmos de chave assimétrica são projetados para que, mesmo com o conhecimento da chave de cifração, seja inviável deduzir a chave de decifração. Esses algoritmos são também chamados de algoritmos de chave pública porque permitem que uma das chaves seja de alcance público. Este fato possibilitou a utilização de técnicas criptográficas em cenários inéditos como comércio eletrônico e assinaturas digitais. Em 1985, Neal Koblitz e Victor Miller propuseram independentemente a utilização de curvas elípticas na construção de sistemas criptograficos de chave pública. A descoberta de criptossistemas baseados em curvas elípticas produziu uma nova revolução na área de criptografia, porque possuem baixos requisitos para armazenamento de chaves e baixo custo computacional para execução. Além disso, a descoberta de técnicas criptográficas baseada em emparelhamentos bilineares sobre curvas elípticas trouxe ainda novas aplicações como sistemas baseados em identidades. No entanto, o custo computacional desses criptossistemas ainda permanece significativamente maior do que os tradicionais, representando um obstáculo para sua adoção, especialmente em dispositivos com recursos limitados. Dentre esse dispositivos, os *smartphones* e os *tablets* têm obtido grande sucesso no mercado por garantir acesso a capacidades cada vez maiores de computação aliadas a mobilidade e portabilidade. Como grande parte desses dispositivos possuem processadores ARM, este trabalho tem como objetivo geral aprimorar o desempenho dos métodos de criptografia baseado em emparelhamento bilineares sobre curvas elípticas nos processadores ARM.

Palavras-chave: Emparelhamentos bilineares, Criptografia Assimétrica, Segurança Computacional, Implementações Eficientes de Primitivas Criptográficas

Abstract

Asymmetric key algorithms are designed so that, even with the knowledge of the encryption key, it is not feasible to deduce the decryption key. These algorithms are also called public key algorithms because they allow one of the keys to be made public. This fact made possible the use of cryptographic techniques in new scenarios such as digital signatures and electronic commerce. In 1985, Neal Koblitz and Miller Victor independently proposed the use of elliptic curves for the construction of public key cryptographic systems. The discovery of cryptosystems based on elliptic curves produced a new revolution in cryptography, because of their low requirements for key storage and low computational cost for execution. Furthermore, the discovery of cryptographic techniques based on bilinear pairings on elliptic curves brought even new applications such as identity-based systems. However, the computational cost of these cryptosystems remains signi cantly higher than the traditional ones, representing an obstacle to their adoption, especially on devices with limited resources. Among such devices, smartphones and tablets have achieved great success in the market by ensuring access to increasingly large computing capacities combined with mobility and portability. Since most of these devices run on an ARM processor, this work aims to improve the overall performance of encryption methods based on bilinear pairing on elliptic curves on ARM-based processors.

Keywords: Pairing-Based, Public-key Cryptography, Computer Security , Efficient implementation of Cryptographic Primitives

Sumário

1 Introdução **1**
1.1 Objetivo 3

2 Criptografia assimétrica **5**
2.1 Motivação 5
2.2 Criptografia de chave pública 5
2.2.1 Protocolo Diffie-Helman 5
2.2.2 Cifração 7
2.2.3 Assinatura digital 8
2.3 Criptossistemas assimétricos 10
2.3.1 RSA 10
2.3.2 ElGamal 12
2.3.3 Rabin 14

3 Criptografia de curvas elípticas **16**
3.1 Fundamentação matemática 16
3.1.1 Grupo abeliano 16
3.1.2 Grupo cíclico 17
3.1.3 Corpo 17
3.2 Curvas elípticas 17
3.3 Criptografia de curvas elípticas 18
3.3.1 Problema do logaritmo discreto em curvas elípticas 18
3.4 Criptossistemas baseados em curvas elípticas 19
3.4.1 Cifração ElGamal 19
3.4.2 Assinatura ECDSA 20
3.5 Emparelhamentos bilineares 22
3.5.1 Acordo de chaves não-interativo 22

4 Aritmética em corpos primos **24**
4.1 Corpo primo 24
4.2 Representação na base b 24
4.3 Adição 25
4.4 Subtração 26
4.5 Multiplicação 26
4.5.1 Schoolbook 27
4.5.2 Comba 28
4.5.3 Híbrido 29

4.5.4 Karatsuba 29
4.6 Quadrado 31
4.7 Redução modular 32
4.8 Exponenciação 34
4.9 Inversão 34
4.9.1 Algoritmo Estendido de Euclides 34
4.9.2 Algoritmo de inversão binário 36

5 Plataforma ARM **39**
5.1 Arquitetura RISC 40
5.2 Arquitetura ARM 41
5.2.1 Banco de registradores e execuções condicionais 41
5.2.2 Chamadas de função 43

6 Implementação **44**
6.1 Adição modular 44
6.1.1 Adição 45
6.1.2 Comparação 46
6.1.3 Subtração 49
6.1.4 Adição modular com instruções de desvio 50
6.1.5 Adição modular sem instruções de desvio 51
6.2 Subtração modular 52
6.2.1 Subtração modular com instruções de desvio 53
6.2.2 Subtração modular sem instruções de desvio 53
6.3 Multiplicação 53
6.3.1 Comba 53
6.3.2 Híbrido 55
6.3.3 Karatsuba 57
6.4 Redução modular 58
6.4.1 Montgomery 58
6.4.2 Redução modular preguiçosa 60

7 Resultados obtidos **61**
7.1 Resultados 61
7.2 Comparação com trabalhos relacionados 62
7.3 Conclusão 63

Referências **64**

Lista de Figuras

1.1 Sistema Criptográfico 2
1.2 Criptografia Simétrica. Com adaptações (9) 3

2.1 Protocolo Diffie-Hellman para acordo de chaves. 6
2.2 Protocolo Diffie-Hellman com ataque Man-in-the-Middle. 7
2.3 Criptografia Assimétrica. Com adaptações (9) 7
2.4 Paradigma Merkle-Damgard. 10

3.1 Curvas Elipticas sobre $\mathbb{R}$ (16) 17
3.2 Adição dos pontos $P + (-P) = \infty$. Com adaptações (17) 18
3.3 Adição e duplicação geométrica de pontos em curvas elípticas (2) 19

4.1 Representação do número $a \in \mathbb{F}_p$ como um vetor de palavras de w *bits* (16) 24
4.2 Exemplo de multiplicação multiprecisão utilizando o algoritmo *Schoolbook* (10). Com adaptações (21). 28
4.3 Multiplicação Híbrida 30
4.4 Multiplicação Karatsuba 31

5.1 Dispositivos Móveis 39
5.2 Chamada de Funções em ARM 43

Lista de Tabelas

2.1 Criptossitema RSA - Cifração 11
2.2 Criptossitema RSA - Assinatura Digital 12
2.3 Criptossitema ElGamal - Cifração 13
2.4 Criptossitema ElGamal - Assinatura Digital 14
2.5 Criptossitema Rabin - Cifração 15

3.1 ECDSA 21
3.2 Acordo de chaves não-interativo 23

5.1 Códigos de condição 42
5.2 Condicionais Mnemônicos 42

7.1 Resultados Obtidos em milhões de ciclos 62
7.2 Comparação com trabalhos relacionados - Tempo em milhões de ciclos . . . 63

Capítulo 1

Introdução

Etimologicamente, a palavra criptografia é originada da língua grega; é formada pela junção de *kryptós*, "escondido" e *gráphein* "escrita", ou seja, escrita secreta. A criptografia é o ramo interdisciplinar da matemática e da ciência da computação que estuda técnicas para estabelecer comunicação segura na presença de um adversário. Antigamente, essas técnicas foram extensamente utilizadas na troca de mensagens, sobretudo em assuntos ligados à guerra e à diplomacia. No entanto, atualmente a utilização dessas técnicas está disseminada nas mais diversas aplicações, sendo inclusive amplamente utilizada no cotidiano das pessoas. Por exemplo, está presente nas redes de comunicação e na proteção de transações financeiras.

O objetivo fundamental da criptografia é permitir que duas pessoas, na literatura normalmente referidas como Alice e Bob, se comuniquem utilizando um canal inseguro de tal maneira que um oponente, Oscar, não possa entender o que está sendo transmitido (32). Essa canal pode ser uma rede de computador, por exemplo. A informação que Alice quer transmitir para Bob, denominada texto claro, pode ser um texto escrito em qualquer língua e deve ser cifrado usando uma chave pré-determinada. Em seguida, o resultado da cifração, denominado criptograma, é transmitido pelo canal para Bob. Oscar, que pode estar espionando o canal, consegue ver o criptograma e conhece o algoritmo de cifração utilizado, mas não pode determinar o conteúdo do texto original. Porém, Bob, que conhece a chave de cifração pode decifrar o criptograma obtendo o texto claro.

A criptografia, no entanto, não se limita somente ao uso em contextos que envolvam requisitos como sigilo. Pode-se empregar técnicas criptográficas para abranger também situações em que seja necessária a autenticação, a integridade, a irretratabilidade e o anonimato.

Atualmente, temos que usar a criptografia em diversos dispositivos e sistemas móveis que vão ganhando espaço no mercado consumidor de eletrônicos. São celulares, *tablets, smartphones* e outros equipamentos que estão cada vez mais disseminados. Com eles temos *hardware* e *software* diversificados e com a Internet móvel, também necessitamos garantir elementos de segurança e até requisitos específicos a esse contexto. A criptografia também precisa acompanhar o avanço desses dispositivos, precisando ser rápida, compacta e portátil.

Podemos classificar os algoritmos criptográficos em dois grupos: os de chave simétrica e os de chave assimétrica.

Algoritmos de chave simétrica são projetados para que a robustez da cifra não dependa da confidencialidade do algoritmo, mas sim do sigilo de suas chaves, sendo que a chave de cifração é facilmente dedutível da de decifração e vice-versa. Na maioria dos algoritmos simétricos, a chave de cifração é igual à chave de decifração (28). Para que esse tipo de modelo seja considerado seguro, é necessário que algumas premissas sejam satisfeitas. Primeiro, a função de cifração deve ser unidirecional para cada chave, ou seja, deve ser inviável realizar a decifração sem o conhecimento da chave utilizada. Segundo, a robustez da cifra depende do sigilo do par de chaves, da equiprobabilidade de qualquer chave ter sido escolhida e do espaço de chaves ser grande.

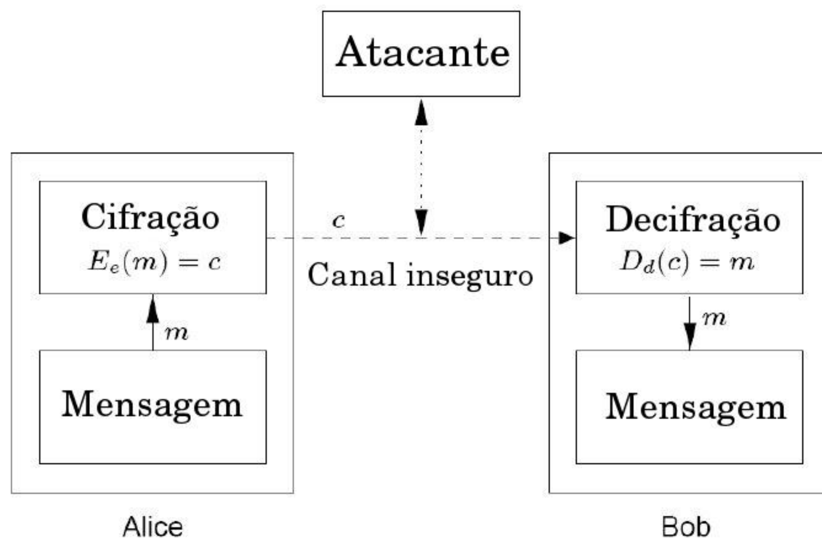

Figura 1.1: Sistema Criptográfico

O primeiro algoritmo criptográfico simétrico desenvolvido para uso comercial foi o *Data Encryption Standard* (DES) que em 1976 foi definido como padrão pela IBM conjuntamente com a NSA (*National Security Agency*). A adoção do DES como padrão durou até meados de 2001 quando foi substituido pelo *Advanced Encryption Standard* (AES).

Algoritmos de chave assimétrica são projetados para que, mesmo com o conhecimento da chave de cifração, seja inviável deduzir a chave de decifração (16). Esses algoritmos são também chamados de algoritmos de chave pública porque permitem que uma das chaves seja de alcance público. Qualquer pessoa pode usar a chave de cifração (chave pública) para cifrar uma mensagem, porém somente o responsável pela chave de decifração (chave privada) pode realizar a operação inversa, obtendo o texto original. A robustez dos algoritmos assimétricos depende do sigilo da chave privada, da dificuldade de se recuperar a chave privada a partir da chave pública, da equiprobabilidade de qualquer par de chaves ser escolhido e do espaço de chaves ser grande.

A noção de criptografia de chave pública foi introduzida em 1976 por Diffie, Hellman e Merkle. Sendo que em 1978, Ron Rivest, Adi Shamir e Len Adleman (24) propuseram o primeiro esquema de criptografia de chave pública para cifração e assinatura digital, denominado RSA, cuja segurança se baseia na dificuldade de se fatorar números inteiros. ElGamal (12) em 1985 propôs um modelo de criptografia assimétrica que se baseava em um problema diferente do RSA, o problema do logaritmo discreto.

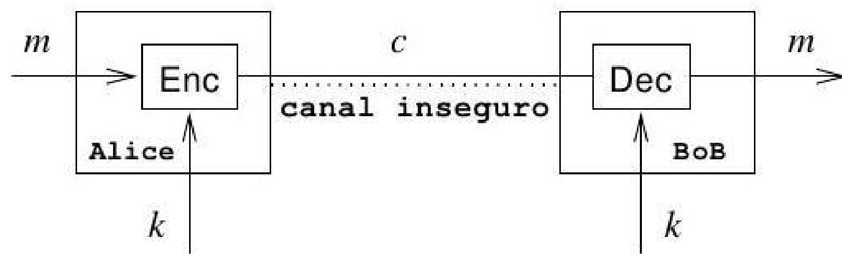

Figura 1.2: Criptografia Simétrica. Com adaptações (9)

Em 1985, Neal Koblitz e Victor Miller propuseram independentemente a utilização de curvas elípticas na construção de sistemas criptograficos de chave pública (16). A descoberta de criptossistemas de chave pública baseados em curvas elípticas produziu uma nova revolução na área de criptografia, pois criptografia baseada em curvas elípticas apresentam um desempenho superior além de exigirem um tamanho de chaves inferior para um mesmo patamar de segurança que as demais técnicas de criptografia assimétrica (4), principalmente quando comparado com o algoritmo RSA. Por esses motivos, em cenários com limitações quanto ao poder de processamento e armazenamento computacional, como a computação móvel, este modelo tem se mostrado ideal.

Desde então, muitos pesquisadores publicaram trabalhos sobre a segurança e a eficiência de implementações que utilizassem a criptografia de curvas elípticas. Sendo que a eficiência de esquemas criptográficos baseados em estruturas algébricas depende de inúmeros fatores como tamanho dos parâmetros, poder de processamento disponível, otimizações de *software* e *hardware* e de algoritmos matemáticos (21).

1.1 Objetivo

Este presente trabalho tem como objetivo geral aprimorar o desempenho dos métodos de criptografia baseado em emparelhamentos bilineares sobre curvas elípticas nos processadores ARM. Como objetivos específicos temos o projeto de técnicas de otimização de aritmética em corpos primos na arquitetura ARM e a implementação dos algoritmos em código funcional e eficiente.

Como metodologia, primeiramente foi realizado o levantamento bibliográfico, estudando os principais algoritmos de aritmética em corpos primos. Em seguida foi realizado o desenvolvimento e a implementação eficiente dos algoritmos em assembly ARM sendo realizada a sua validação. Finalmente, foi feita a tomada de tempo dos algoritmos implementados.

A apresentação do trabalho será realizada em sete capítulos. O capítulo 1 realizou uma breve introdução ao tema. O capítulo 2 fala sobre a criptografia assimétrica sendo apresentado alguns criptossistemas. O capítulo 3 discute alguns dos conceitos relativos a criptografia de curvas elípticas e aos emparelhamentos bilineares. No capítulo 4 são mostrados os principais algoritmos de aritmética em corpos primos. O capítulo 5 mostra alguns detalhes da arquiteura RISC e da plataforma ARM. No capítulo 6 descreve-se a implementação dos algoritmos em assembly ARM. Por fim, no capítulo 7 são mostrados

os resultados obtidos e realizada a comparação com trabalhos relacionados além de uma breve conclusão.

Capítulo 2

Criptografia assimétrica

2.1 Motivação

Embora a criptografia simétrica seja altamente eficiente do ponto de vista computacional, ela possui algumas desvantagem intrínsecas.

Primeiro, para o uso correto de técnicas criptográficas simétricas é necessário que antes de se transmitir a mensagem por um canal inseguro, a distribuição das chaves a serem usadas seja realizada por outro canal. É importante ressaltar que o canal pelo qual é habilitado o uso da criptografia deve ser confiável em relação às premissas (integridade e sigilo) para a eficácia das técnicas escolhidas, e portanto deve ser diferente do canal por onde será realizada a comunicação. Caso contrário, o uso da criptografia seria desnecessário ou ineficaz. Esse problema é conhecido na literatura com o problema da distribuição de chaves.

Segundo, em um cenário com várias entidades querendo comunicar-se entre si utilizando técnicas de criptografia simétrica, cada entidade deve manter uma chave diferente seguindo as premissas de sigilo e integridade para cada outra entidade. O elevado grau de complexidade dessa administração é chamado de problema do gerenciamento de chaves.

Finalmente, em criptossistemas simétricos as chaves utilizadas são compartilhadas entre duas (ou mais) entidades e não há como criar esquemas que possibilitem a assinatura digital irretratável, ou seja, esquemas que inviabilizam tecnicamente a negação da autoria da assinatura pela autor, visto que duas entidades compartilham a mesma chave e não há como distinguir a ação dessas chaves.

2.2 Criptografia de chave pública

2.2.1 Protocolo Diffie-Helman

Foi proposto por Diffie e Hellman (11) o primeiro esquema que tornou possível a derivação de chaves secretas entre duas entidades em um canal inseguro sem comprometer a segurança.

Para exemplificar o funcionamento desse protocolo, suponha que Alice e Bob queiram trocar mensagens com garantias de integridade e sigilo em uma canal inseguro. No entanto, por algum motivo não é possível utilizar um outro canal seguro para realizar a troca de chaves. Dessa forma a troca de chaves deverá ser feita no canal inseguro. Mas

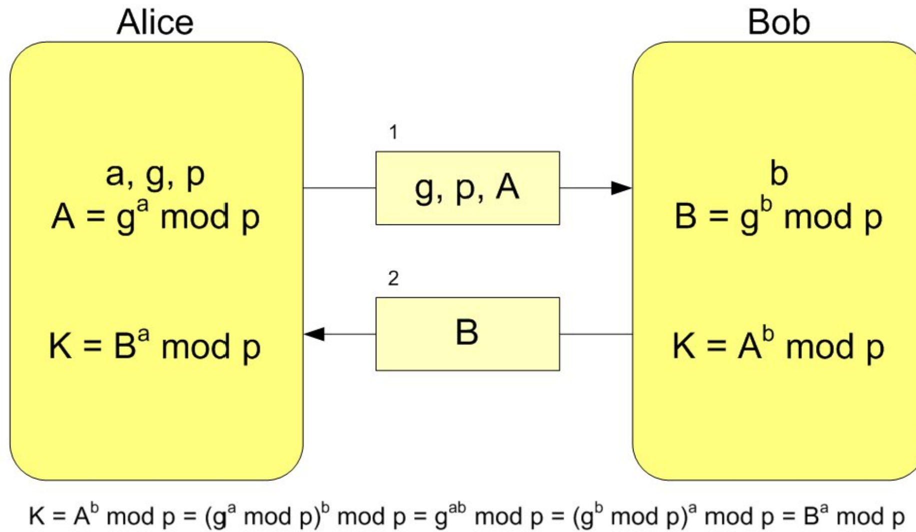

Figura 2.1: Protocolo Diffie-Hellman para acordo de chaves.

dessa forma, um atacante externo poderá observar a troca de chaves e obter o segredo que habilitaria o uso da criptografia, podendo dessa forma monitorar toda a troca de mensagens entre Alice e Bob.

O protocolo proposto por Diffie e Helman atua justamente nesse contexto, permitindo a troca de chaves entre dois indivíduos em uma canal inseguro. Para que isso ocorra, primeiramente Alice deverá gerar três inteiros uniformemente aleatórios (a, g, p), onde p é um primo. Tendo posse desses números, calcula-se $A = g^a \pmod{p}$. O inteiro a é denominado chave secreta é deve ser mantido em sigilo. Alice então, envia para Bob os inteiros (g, p, A). Bob recebe estes inteiros e gera um inteiro b. Bob então calcula $B = g^b \pmod{p}$ e $K = A^b \pmod{p}$. Bob então deve transmitir à Alice o valor de B. Finalmente, Alice recebe o valor de B de Bob e calcula a chave $K = B^a \pmod{p}$ compartilhada com Bob. Note que a chave K é a mesma, pois:

$$\begin{aligned} K &= A^b \mod p \\ &= (g^a \mod p)^b \mod p \\ &= (g^{ab} \mod p) \\ &= (g^b \mod p)^a \mod p = B^a \mod p \end{aligned} \tag{2.1}$$

Este protocolo deriva a mesma chave de sessão aos dois participantes porque a exponenciação modular é homomórfica entre anéis comutativos. No entanto, embora Diffe e Hellman tenham introduzido um grande avanço na área de criptografia, o seu protocolo não pode ser utilizado em rede aberta (Internet, por exemplo), pois é vulnerável a ataques *Man-in-the-Middle*, onde um atacante que se encontra interceptando as mensagens realiza um acordo de chaves independente com cada um dos indivíduos e age trocando as mensagens sem que estes percebam.

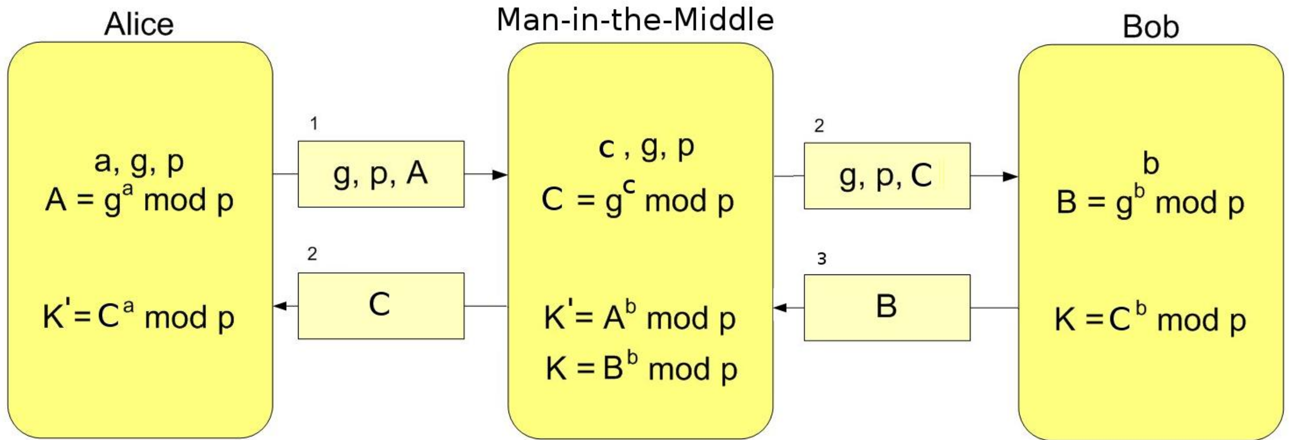

Figura 2.2: Protocolo Diffie-Hellman com ataque Man-in-the-Middle.

2.2.2 Cifração

Na criptografia assimétrica a cifração e a decifração são realizadas utilizando-se um par de chaves, e e d. Mais importante, o par de chaves deve possui a propriedade de que dada uma chave e, seja computacionalmente inviável a dedução do seu par d.

A cifração da mensagem a ser transmitida é feita utilizando-se a chave de cifração e. Suponha que Alice queira enviar uma mensagem confidencial a Bob, em um cenário na qual é utilizada a criptografia assimétrica. Primeiro é necessário que Bob coloque em um repositório público a sua chave de cifração e (por esse motivo, e também é conhecida como chave pública). Essa ação não compromete a chave privada d visto que é inviável deduzir d a partir de e. Em seguida, é preciso que Alice obtenha um cópia autêntica da chave pública de Bob. Alice deve então cifrar a mensagem a ser transmitida para Bob utilizando essa chave. Bob, ao receber o criptograma, o decifra utilizando a sua chave privada d, recuperando então a mensagem original. Repare, que nesse cenário, não é exigido o sigilo da chave e sendo assumido apenas a sua integridade, ou seja, o problema da distribuição de chaves foi simplificado ao problema de obter uma chave pública autêntica.

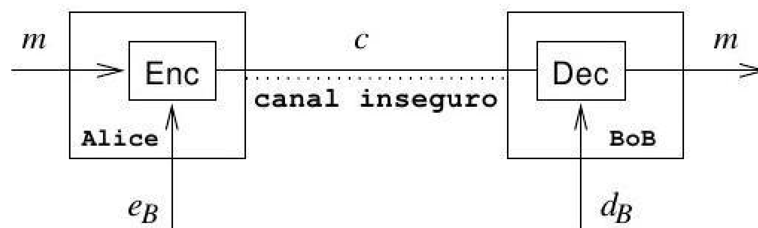

Figura 2.3: Criptografia Assimétrica. Com adaptações (9)

2.2.3 Assinatura digital

Com a descoberta de criptossistemas assimétricos foi possível a criação de esquemas de assinatura digital possibilitando a verificação da autenticidade da origem da mensagem. Por exemplo, suponha que Alice use um algoritmo de assinatura digital. Com sua chave privada d, ela gera um autenticador (assinatura) a partir da mensagem a ser enviada. Em seguida Alice envia para Bob a mensagem e o autenticador. Bob recebendo ambos utiliza a mensagem recebida e a chave pública de Alice e para verificar se o autenticador enviado foi realmente gerado utilizando a mensagem e a chave privada. Sendo presumido que a chave privada d seja de conhecimento apenas de Alice, Bob pode se assegurar que a mensagem foi de fato originada por Alice.

Esquemas de assinatura digital possuem dois algoritmos, um algoritmo de assinatura e um algoritmo de verificação. O algoritmo de assinatura produz, a partir de um mensagem e uma chave privada, um autenticador (ou assinatura). Esse assinatura pode ser utilizada posteriormente por terceiros para verificar a autenticidade da mensagem, para isso deve ser utilizado o algoritmo de verificação. O algoritmo de verificação recebe como parâmetros de entrada a assinatura e a mensagem e verifica se essa é uma assinatura válida para a mensagem e a chave pública do suposto signatário.

A assinatura digital deve simular no mundo virtual a assinatura de punho. Para que isso aconteça, devem ser preservadas as seguintes características: inforjabilidade, inviolabilidade, irrecuperabilidade e irrefutabilidade. A inforjabilidade é uma propriedade que garante ao verificador a identicação da autoria do documento assinado e a descoberta de possíveis falsificações. A inviolabilidade garante a integridade do conteúdo no momento do vínculo da assinatura. A irrecuperabilidade diz respeito a inviabilidade do reuso da assinatura e a irrefutabilidade na inviabilidade de negação da autoria da assinatura por parte do assinante. Todas essas propriedades devem estar presentes em um esquema de assinatura digital para que este possa ser utilizado de forma eficaz.

Esquemas de assinatura digital são quase sempre usados em combinação com outra técnica criptográfica denominada funções de resumo criptográfico. Essas funções serão definidas em detalhes na próxima subseção, porém podem ser entendidas informalmente como uma função de recebe como entrada uma mensagem de tamanho qualquer e retorna um cadeia de tamanho fixo. Funções de resumo são utilizadas em esquemas de assinaturas digitais no algoritmo de assinatura, onde o assinante, ao invés de realizar a assinatura diretamente sobre a mensagem, a realiza sobre o resumo criptográfico da mensagem. Esse cuidado evita uma série de ataques se a função de resumo criptográfico satisfizer algumas propriedades.

Funções de resumo criptográfico

Funções de resumo criptográfico são funções que mapeiam um conjunto de mensagem de tamanho arbitrário para um conjunto de resumos (ou autenticadores) de tamanho fixo, denotado por $y = h(x)$, onde y é o resumo produzido, x e a mensagem utililizada e $h(\cdot)$ a função de resumo. Esse tipo de função deve idealmente ter as seguintes propriedades: o cálculo do resumo da mensagem deve ser realizado em tempo polinomial; deve ser inviável computacionalmente o cálculo da inversão; deve ser inviável gerar uma mensagem que tenha um determinado resumo criptográfico; deve ser inviável obter duas mensagens diferentes com o mesmo resumo criptográfico.

Porém, podemos considerar que uma determinada função de resumo criptográfico é suficientemente segura se os três problemas a serem apresentados forem difíceis de serem resolvidos. O primeiro problema consite na resistência à pré-imagem, isto é, dado um resumo y, deve ser inviável encontrar uma mensagem x tal que $y = h(x)$. Este problema mostra que para uma função de resumo ser considerada segura, deve ser difícil realizar a operação inversa. O segundo problema, na resistência à segunda pré-imagem, ou seja dado um resumo y e uma mensagem x tais que $y = h(x)$, deve ser inviável encontrar uma mensagem distinta $x' \neq x$ tal que $h(x') = h(x) = y$. O terceiro problema é relativo à resistência a colisão, isto é, deve ser inviável encontrar duas mensagens distintas com o mesmo resumo criptográfico.

Esse tipo de função possui diversos usos em aplicações criptográficas, podendo ser utilizadas para armazenar senhas, no qual seria armazenado o resumo da senha ao invés da senha em claro. Pode ser utilizada para verificação da integridade de mensagens, pois qualquer modificação na mensagem deveria alterar pelo menos metade do seu resumo criptográfico. E uma outra área de aplicação de funções de resumo é nos esquemas de assinatura digital, no qual se assina o resumo da mensagem ao invés da mensagem em claro, evitando assim alguns ataques como forja seletiva que ocorre quando um adversário quer obter uma assinatura sobre uma determinada mensagem.

Existem diversas funções de resumo criptográfico no mercado, algumas mais conhecidas e utilizadas são o MD5, SHA-1, SHA-2. Porém, nem todas são consideradas seguras pois pode-se demostrar ataques que quebram algumas das propriedades citadas acima. Tao Xie e Dengguo (35), por exemplo, demonstraram em 2009 como encontrar colisões no MD5 com ordem de complexidade de $2^{20.96}$, podendo este ataque ser realizado em alguns segundos em computadores convencionais. Xiaoyun Wang, Hongbo Yu e Yiqun Lisa Yini (33) demonstraram ainda em 2005 como encontrar colisões no SHA-1 com complexidade menor que 2^{69}, sendo que um ataque de força bruta tem complexidade de 2^{80}. A função de resumo SHA-1 possui a mesma estrutura que o SHA-2 o que potencialmente levaria a descoberta de possíveis ataques ao SHA-2 e que ocorreu em 2009 quando Yu Sasaki, Lei Wang e Kazumaro Aoki (27) demonstraram um ataque de pré-imagem na função SHA-2 reduzida.

Por esse motivo foi criada pelo NIST (*National Institute of Standards and Technology*) em 2007 uma competição que reuniu diversos criptólogos. Essa competição teve como objetivo eleger uma nova função de resumo criptográfico, denominada SHA-3, para substituir as funções SHA-1 e SHA-2. Esse processo foi similar à competição que elegeu o AES como padrão de cifração em 2001. Essa competição terminou em outubro de 2012, sendo anunciado pelo NIST a função *Keccak* (6) como novo padrão para funções de resumo criptográfico.

É importante ressaltar que o novo SHA-3 introduziu um novo paradigma na construção de funções de resumo. As funções MD5, SHA-1, SHA-2 são todas baseadas no paradigma de Markle-Damgard (10) que permite a construção de funções de resumo com domínio infinito a partir de aplicações sucessivas de uma mesma função de compressão. Porém, o SHA-3 (*Keccak*) se baseia no paradigma de contruções esponja que recebe como parâmetro de entrada uma fluxo de *bits* de tamanho qualquer e produz um fluxo de *bits* de tamanho fixo.

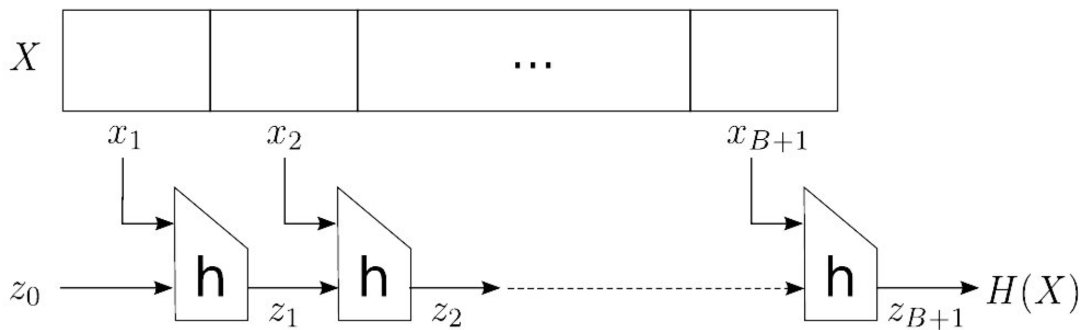

Figura 2.4: Paradigma Merkle-Damgard.

2.3 Criptossistemas assimétricos

Embora sistemas criptográficos assimétricos sejam mais lentos que sistemas simétricos, eles resolvem problemas inerentes dos sistemas simétricos (distribuição e gerenciamento de chaves), com a vantagem de ainda possibilitar esquemas de assinatura digital. No entanto, embora na criptografia assimétrica não seja mais necessário a utilização de um canal seguro para realizar a distribuição das chaves, é preciso, em contra-partida a criação de uma infra-estrutura de chaves públicas para o gerenciamento e distribuição das chaves com o objetivo de verificar que um certa chave-pública pertence de fato a um determinado indivíduo.

Nem todos algoritmos assimétricos propostos são seguros e práticos o suficiente para serem usados. Serão detalhados na seção seguinte alguns algoritmos que alcançaram tal feito: RSA, ElGamal e Rabin. Entretanto, nem todos os algoritmos assimétricos são utilizados por serem considerados inseguros, muito lentos ou por usarem chaves muito longas.

2.3.1 RSA

De todos os algoritmos de chave-pública propostos, o RSA é considerado por muitos o mais simples de se entender e implementar. A sua segurança baseia-se no problema da fatoração dos números inteiros, ou seja, recuperar a mensagem tendo acesso à chave pública e o criptograma é conjecturada como equivalente a fatorar o produto de dois primos.

Algoritmo 1 Geração de chaves no Algoritmo RSA (32) com adaptações.

Entrada: Dois inteiros primos p, q.
Saída: A chave pública (n, e) e a chave privada (p, q, d)

1: $n \leftarrow pq$
2: $\phi(n) \leftarrow (p-1)(q-1)$
3: Escolha um inteiro e aleatório sendo que $mdc(e, \phi(n)) = 1$ e $1 < e < \phi(n)$
4: $d \leftarrow e^{-1} \mod \phi(n)$
5: **return** $((n, e)$ e $(p, q, d))$

Criptossitema RSA - Cifração	
Alice	**Bob**
Cifração	
1. Escolhe uma mensagem m 2. Utiliza a chave pública de Bob (n, e) para calcular $c \equiv m^e \pmod{n}$ 3. Manda o criptograma c para Bob	
Decifração	
	1. Recebe o criptograma c 2. Utiliza a sua chave privada d para calcular $m \equiv c^d \pmod{n}$ 3. Obtém a mensagem m

Tabela 2.1: Criptossitema RSA - Cifração

No RSA, a chave pública é um par de inteiros (n, e) sendo n o produto de dois números primos aleatórios (p e q) (1). Para maximizar o nível de segurança, p e q devem ter a mesma magnitude. O expoente de cifração e deve ser um número inteiro tal que $1 < e < \phi(n)$ e deve ser co-primo de $\phi(n)$, isto é, $mdc(e, \phi(n)) = 1$, onde $\phi(n) = (p-1)(q-1)$. A chave privada d, também conhecida como expoente de decifração, é o número inteiro tal que $1 < d < \phi(n)$ e $ed \equiv 1 \pmod{\phi(n)}$.

O algoritmo funciona como uma cifra (2.1), isto é, a função de cifração é inversa da decifração, devido ao Teorema de Euler-Fermat (19). Nos esquemas de cifração e assinatura o RSA utiliza o fato de que $m^{ed} \equiv m \ mod \ n$ para qualquer m $\in \mathbb{Z}_n$. A decifração funciona porque $c^d \equiv (m^e)^d \equiv m \ mod \ n$. Como já dito, a segurança do algoritmo baseia-se na dificuldade de se obter a mensagem em claro m da cifra $c = m^e$ $mod \ n$ tendo conhecimento apenas de n e e. No entanto, o conhecimento de qualquer um dos parâmetros p,q e $\phi(n)$ permitem o cálculo trivial de d, devendo portanto serem guardados em sigilo.

Em esquemas de assinatura utilizando o RSA (2.2) é necessário também a utilização de alguma função de resumo criptográfico. Primeiramente o assinante calcula um resumo da mensagem a ser transmitida utilizando a função de resumo criptográfico, i.e., calcula $h = H(m)$. Então, o assinante utiliza a sua chave privada d para computar o autenticador $s = h^d \ mod \ n$. O assinante então transmite a mensagem e o autenticador para o outra indivíduo. Este por sua vez, calcula um resumo utilizando a mesma função de resumo e a mensagem recebida produzindo h e em seguida cálcula $h' = s^e \ mod \ n$. Se $h = h$' então pode ser assumido que não houve mudança no conteúdo da mensagem e que foi realmente o assinante o indivíduo responsável pela produção e envio da mensagem.

Um aspecto negativo no RSA, do ponto de vista da eficiência, é o elevado custo computacional para se calcular a exponenciação modular (cálculo de $m^e \ mod \ n$ na cifração e o cálculo de $c^d \ mod \ n$ na decifração). Além disso, para ser utilizado na prática, o RSA exige a utilização de técnicas de preenchimento.

Criptossitema RSA - Assinatura Digital	
Alice	**Bob**
Assinatura	
1. Escolhe uma mensagem m. 2. Utiliza a sua chave privada d para calcular $s \equiv m^d \pmod{n}$. 3. Envia a mensagem m e o autenticador s para Bob.	
Verificação	
	1. Recebe a mensagem m e o autenticador s. 2. Utiliza a chave pública de Alice (n, e) para calcular. $s' \equiv s^e \pmod{n}$ 3. Verifica se $s' = h(m)$.

Tabela 2.2: Criptossitema RSA - Assinatura Digital

2.3.2 ElGamal

Em 1985, ElGamal (12) propôs um esquema de chave pública para cifração e esquemas de assinatura utilizando como base o problema do logaritmo discreto.

Problema do logaritmo discreto

Dada a equação $y = g^x \ mod \ p$. O problema do logaritmo discreto é determinar o valor de x, tendo conhecimento dos valores de y, g e p. O algoritmo mais rápido para resolver o problema do logaritmo discreto é o *Number Field Sieve* (NFS) que tem ordem de tempo subexponencial.

Por esse motivo, a utilização do problema do logaritmo discreto em aplicações criptográficas tem sido eficaz, pois embora o seu cálculo seja presumido ser um problema difícil, a sua operação inversa pode ser calculada de maneira eficiente utilizando operações de quadrado, multiplicação e divisão inteira.

Criptossistema ElGamal

Em sistemas com base no logaritmo discreto é necessário antes a geração de três parâmetros (p, q, α) para a criação do par de chaves *(e, d)*. O número p deve ser um primo, q um primo que divide $p - 1$ e α deve ser escolhido de tal forma que tenha a mesma ordem de magnitude de q e $1 \leq \alpha \leq p - 1$. A chave privada d é um inteiro a escolhido de forma aleatória, tendo como única restrição o fato de $a \in [1, q - 1]$. A chave pública e é o valor de β calculado como $\beta = \alpha^a \pmod{p}$ além dos valores p e α.

Em um cenário em que Alice deseja enviar uma mensagem a Bob, primeiro Alice com a chave pública de Bob $e_B = (p, \alpha, \beta)$ realiza a exponenciação $\beta^k \pmod{p}$, onde k é um número aleatório qualquer. E em seguida, Alice cifra a mensagem m calculando $y_1 = \alpha^k \pmod{p}$, e $y_2 = m\beta^k \pmod{p}$, transmitindo para Bob y_1 e y_2. Bob, ao receber o conteúdo

Algoritmo 2 Geração de chaves no Algoritmo ElGamal (16) com adaptações.

Entrada: Dois inteiros primos p, q e o inteiro α.
Saída: A chave pública $e = (p, \alpha, \beta)$ e a chave privada $d = (a)$

1: Escolha o valor de a tal que $1 < a < q - 1$
2: $\beta \leftarrow \alpha^a \mod p$
3: $d \leftarrow e^{-1} \mod \phi(n)$
4: **return** $e = (p, \alpha, \beta)$ e $d = (a)$

Criptossitema ElGamal - Cifração	
Alice	**Bob**
Cifração	
1. Escolhe uma mensagem m 2. Escolhe um k aleatório 3. Utiliza a chave pública de Bob $e_B = (p, \alpha, \beta)$ e o valor de k para calcular $y_1 = \alpha^k \pmod p$ e $y_2 = m\beta^k \pmod p$ e 4. Manda o criptograma $c = (y_1, y_2)$ para Bob	
Decifração	
	1. Recebe o criptograma $c = (y_1, y_2)$ 2. Utiliza a sua chave privada $d = (a)$ para calcular $m = y_2 \cdot {y_1}^{-a} \pmod p$. 3. Obtém a mensagem m

Tabela 2.3: Criptossitema ElGamal - Cifração

transmitido por Alice, com sua chave privada $d_B = (a)$ decifra m ao calcular $m = y_2 \cdot {y_1}^{-a} \mod p$.

A operação de cifração utilizando o algoritmo de ElGamal, tem a curiosa propriedade de produzir alguns criptogramas distintos, na verdade $p - 1$ cifras, de uma mesma mensagem m. Isso acontece porque a cifra depende da mensagem x e da variável aleatoria k. Por esse motivo pode se dizer que a operação de cifração do algoritmo ElGamal não é determinística.

O método de ElGamal também pode ser adaptado para ser utilizado em esquemas de assinatura digital. Tendo este cenário em mente, suponha que Alice queira enviar uma mensagem a Bob tendo como objetivo que a mensagem que Bob receba seja a mesma que ela enviou (validar a origem e a integridade da mensagem). Neste caso, Alice escolhe um inteiro uniformemente aleatório k e, utilizando a sua chave pública $e_A = (p, \alpha, \beta)$ e a sua chave privada $d_A = a$, constrói um autenticador $s = (\gamma, \delta)$, onde $\gamma = \alpha^k \pmod p$ e $\delta = (x - a)\alpha^k \pmod p$. Alice manda para Bob o autenticador juntamente com a mensagem. Bob, então, autentica a mensagem calculando $\beta^y \gamma^\delta \equiv \alpha^m \pmod p$. Se a assinatura foi construída corretamente e não houve nenhum intervenção externa, então

Criptossitema ElGamal - Assinatura Digital	
Alice	**Bob**
Assinatura	
1. Escolhe uma mensagem m 2. Escolhe um k aleatório 3. Utiliza a sua chave pública e privada $e = (p, \alpha, \beta), d = (a)$ e o valor de k para calcular $\gamma = \alpha^k \pmod p$ e $\delta = (x - a)\alpha^k \pmod p$ e 4. Manda a mensagem e o autenticador $c = (m, (\gamma, \delta))$ para Bob	
Verificação	
	1. Recebe a mensagem e o autenticador $c = (m, (\gamma, \delta))$ 2. Utiliza a chave pública de Alice $e = (p, \alpha, \beta)$ para calcular $\beta^y \gamma^\delta \equiv \alpha^m \pmod p$. 3. Realiza a verificação autentica se e somente se $\beta^y \gamma^\delta \equiv \alpha^m \pmod p$

Tabela 2.4: Criptossitema ElGamal - Assinatura Digital

a verificação será realizada garantindo a Bob que a mensagem que ele possui é a mesma que Alice posuía quando realizou a assinatura digital.

Assim como o esquema de cifração utilizando o algoritmo de ElGamal é não-determinístico, o esquema de assinatura também é probabilístico. Isto é, podem existir várias assinaturas válidas para uma mesma mensagem, dessa forma o algoritmo de verificação deve validar qualquer uma dessas assinaturas como uma assinatura autêntica.

2.3.3 Rabin

O sistema criptográfico de Rabin (23) é um esquema com segurança computacional equivalente à fatoração de inteiros. Um fato curioso desse criptossitema é que a função de cifração não é injetiva, por esse motivo existem quatro possíveis decifrações que devem ser analisadas a fim de se obter mensagem original. No entanto, o método de Rabin tem a vantagem de que o problema no qual ele se baseia é provado ser tão difícil quanto o problema da fatoração de inteiros, fato que não é possível afirmar no caso do algoritmo RSA.

A geração de chaves para utilização no criptossistema de Rabin é bem simples. Primeiramente, deve-se escolher dois números primos distintos p e q tal que $p \equiv q \equiv 3 \pmod 4$. Esta condição é utilizada para simplificar a computação de raízes quadradas modulo p

Algoritmo 3 Geração de chaves no Algoritmo de Rabin

Entrada: Dois inteiros primos p, q com k/2 bits e $p \equiv q \equiv 3 \pmod 4$.

Saída: A chave pública (n) e a chave privada (p, q)

1: $n \leftarrow pq$

2: **return** $((n)$ e $(p, q))$

Criptossitema Rabin - Cifração	
Alice	**Bob**
Cifração	
1. Escolhe uma mensagem m 2. Utiliza a chave pública de Bob $e = (n)$ para calcular $c = m^2 \pmod n$ 3. Manda o criptograma c para Bob	
Decifração	
	1. Recebe o criptograma c 2. Utiliza a chave privada $d = (p, q)$ para calcular $m' = \sqrt{c} \pmod p$ e $m'' = \sqrt{c} \pmod q$ 3. Analisa as possíveis mensagens e identifica a mensagem original m

Tabela 2.5: Criptossitema Rabin - Cifração

e q. Em seguida, deve-se calcular o valor de $n = pq$ que será a chave pública e a chave privada que deverá ser guardada em sigilo é a tupla (p, q).

Agora suponha que Alice queira enviar uma mensagem m para Bob utilizando o criptossistema de Rabin. Alice então utiliza a chave pública de Bob $e = n$ para cifrar a sua mensagem transformando-a em $c = m^2 \pmod n$. Em seguida Alice envia o criptograma c para Bob. Este ao recebe o criptograma utiliza a sua chave privada $d = (p, q)$ calcula $m' = \sqrt{c} \pmod p$ e $m'' = \sqrt{c} \pmod q$ obtendo dessa forma quatro possíveis mensagens. Em geral, não existe uma forma de Bob distinguir quais dessas quatro possíveis mensagens é a original se a mensagem não possuir redundância suficiente para eliminar três das quatro possibilidades.

Capítulo 3

Criptografia de curvas elípticas

Recentemente a teoria de curvas elípticas encontrou uma aplicação prática no ramo da criptografia. A principal razão para esta afirmação deve-se ao fato de que implementações de criptografia de curvas elípticas necessitam de um custo de processamento e de armanezamento menor que os métodos tradicionais de criptografia de chave-pública. A descoberta da criptografia baseada em emparelhamentos bilineares sobre curvas elípticas permitiu ainda sua flexibilização e a construção de sistemas criptográficos com propriedades inovadoras, como sistemas baseados em identidades e suas variantes (2).

3.1 Fundamentação matemática

3.1.1 Grupo abeliano

Sejam G um conjunto não vazio e $(x, y) \to x * y$ uma lei de composição interna em G. Dizemos que G é um grupo abeliano em relação a essa lei se, e somente se:

1. $a * (b * c) = (a * b) * c, \forall a, b, c \in G$ (propriedade associativa)
2. $\exists e \in G$ tal que $a * e = e * a = a, \forall a \in G$ (elemento neutro)
3. $\forall a \in G, \exists a' \in G \mid a * a' = a' * a = e$ (elemento inverso)
4. $a * b = b * a, \forall a, b \in G$ (propriedade comutativa)

A composição interna em G é normalmente denominada adição $(+)$ ou multiplicação $(\cdot)$. No primeiro caso, o grupo em específico recebe o nome de grupo aditivo sendo geralmente usado o *0* para representar o elemento neutro e tendo o inverso aditivo de a denotado por $-a$. No segundo, o grupo é denominado grupo multiplicativo sendo normalmente usado o *1* para representar o elemento neutro e tendo o inverso multiplicativo de a denotado por a^{-1}

3.1.2 Grupo cíclico

Seja G um grupo multiplicativo. Dizemos que G é um grupo cíclico se existe um elemento $a \in G$ tal que $G = \{a^m | m \in \mathbb{Z}\}$. O elemento a é chamado de gerador de G.

3.1.3 Corpo

Seja $\mathbb{F}$ um conjunto composto de duas operações $(+)$ e $(\cdot)$. Dizemos que $(\mathbb{F}, +, \cdot)$ é um corpo se, e somente se:

1. $(\mathbb{F}, +)$ é um grupo abeliano com elemento identidade 0
2. $(\mathbb{F}^*, \cdot)$ é um grupo abeliano com elemento identidade 1, onde $\mathbb{F}^* = F \backslash \{0\}$
3. $a * (b + c) = a * b + b * c$, $\forall a, b, c \in G$ (a operação $(\cdot)$ e distributiva sobre a operação $(+)$)

Dizemos que um corpo é finito se $\mathbb{F}$ é um conjunto finito. Nesse caso, o número de elementos de $\mathbb{F}$ é chamado de ordem de $\mathbb{F}$.

3.2 Curvas elípticas

Seja p um número primo, e seja $\mathbb{F}_p$ o corpo dos inteiros módulo p. Uma curva elíptica E sobre um corpo $\mathbb{F}_p$ é definida pela equação:

$$E : y^2 = x^3 + ax + b \tag{3.1}$$

onde $a, b \in \mathbb{F}_p$, satisfaz $4a^3 + 27b^2 \neq 0 \pmod{p}$. Uma par (x, y) onde $x, y \in \mathbb{F}_p$ é um ponto da curva se (x, y) satisfaz a equação acima. O ponto no infinito (Figura 3.2), denotado por ∞ também é um ponto da curva elíptica. O conjunto de todos os pontos da curva E é denotado por $E(\mathbb{F}_p)$.

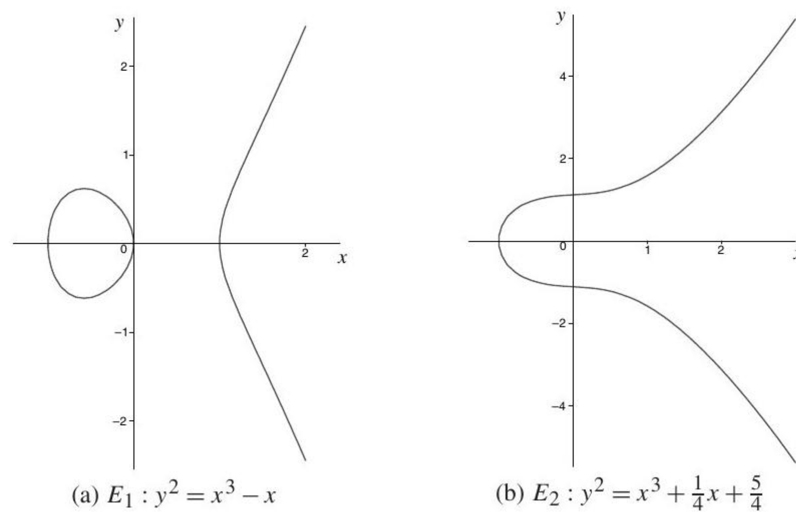

Figura 3.1: Curvas Elipticas sobre $\mathbb{R}$ (16)

Por exemplo, se E é uma curva elíptica definida sobre $\mathbb{F}_7$ dada pela equação: $y^2 = x^3 + 2x + 4$ então os pontos sobre a curva E são: E($\mathbb{F}_7$) = { ∞, (0,2), (0,5), (1,0), (2,3), (2,4), (3,3), (3,4), (6,1), (6,6) }

Curvas elípticas fornecem uma regra geométrica para a adição de pontos. Sejam $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ pontos distintos sobre uma curva elíptica E. A soma dos pontos P e Q é um ponto R definido como a reflexão sobre o eixo x do ponto R' formado pela intersecção entre a curva E e a reta que passa por P e Q (Figura 3.3). O conjunto de pontos $E(\mathbb{K})$ em conjunção com a regra de adição forma o grupo abeliano $E(\mathbb{K}, +)$, tendo como elemento identidade o ponto no infinito ∞. A duplicação de um ponto P $(P + P)$ é um ponto R definido como a reflexão no eixo x do ponto R' formado pela intersecção entre a curva E e a reta tangente ao ponto P.

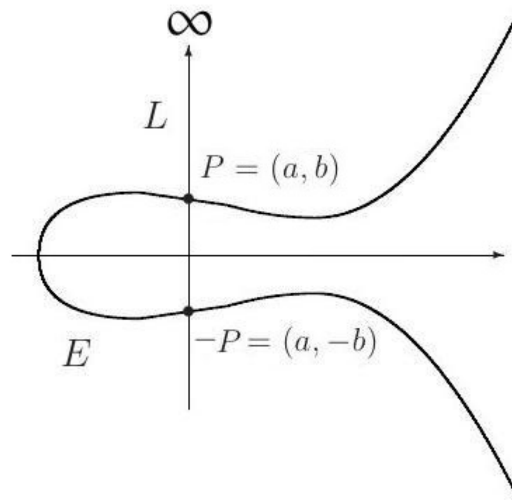

Figura 3.2: Adição dos pontos $P + (-P) = \infty$. Com adaptações (17)

3.3 Criptografia de curvas elípticas

Seja E uma curva elíptica definida sobre o corpo $\mathbb{F}_p$. Seja P um ponto em $E(\mathbb{F}_p)$. Então, o subgrupo cíclico de $E(\mathbb{F}_p)$ gerado por P é:

$$< P >= \{\infty, P, 2P, 3P, ..., (n-1)P\} \tag{3.2}$$

Em criptossistemas de curvas elípticas, a chave privada é um inteiro d escolhido aleatoriamente entre o intervalo $[1, n-1]$ e a chave pública é $Q = dP$. A segurança desses tipos de criptossistemas baseia-se no problema do logaritmo discreto em curvas elípticas.

3.3.1 Problema do logaritmo discreto em curvas elípticas

Seja E uma curva elíptica sobre um corpo finito $\mathbb{F}_p$ e seja P e Q pontos na curva elíptica $E(\mathbb{F}_p)$. O problema do logaritmo discreto em curvas elípticas pode ser definido

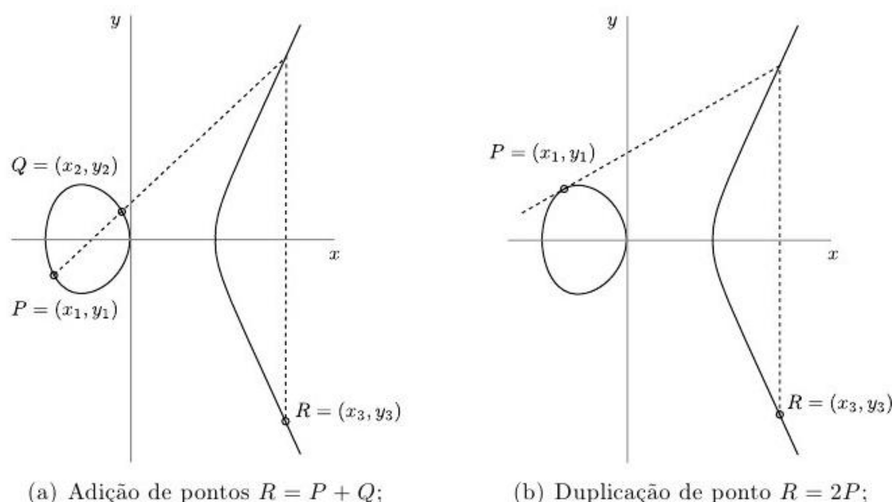

Figura 3.3: Adição e duplicação geométrica de pontos em curvas elípticas (2)

como o problema de se encontrar um inteiro n tal que $Q = nP$. Sendo o ponto Q a soma do ponto P a ele mesmo n vezes, ou seja, $Q = \overbrace{P + P + P + ... + P}^{n} = nP$.

Fazendo uma analogia com o problema do logaritmo discreto, podemos denotar n explícitamente como: $n = log_P(Q)$.

O algoritmo mais rápido para resolver o problema do logaritmo discreto em curvas elípticas é o método *ρ de Pollard* (7) que tem ordem de tempo exponencial. Uma das principais razões para a difusão cada vez maior de criptossistemas baseados em curvas elípticas é o fato de que não se conhece algoritmo para resolver o problema do logaritmo discreto em curvas elípticas mais rápido do que o método de ordem de tempo exponencial. Embora exista uma técnica denominada cálculo de índice (34) para atacar o problema do logaritmo discreto que tem ordem de tempo subexponencial, ele não pode ser utilizado em sistemas baseados em curvas elípticas, pois reduz o problema do logaritmo discreto para o problema da fatoração. No entanto, não existe a noção de fatoração de pontos em uma curva elíptica, desse modo esta técnica não pode ser utilizada nessa classe de sistemas criptográficos, fazendo com que a técnica mais eficiente para a atacar criptografia de curvas elípticas seja de ordem exponencial.

3.4 Criptossistemas baseados em curvas elípticas

3.4.1 Cifração ElGamal

Como exemplo de criptossitemas baseados em curvas elípticas pode-se utilizar um esquema de cifração e decifração utilizando curvas elípticas análogo ao método de ElGamal. Primeiramente, para gerar as chaves a serem utilizadas no sistema criptográficos é necessário um primo p, uma equação de curvas elípticas E sobre $\mathbb{F}_p$, um ponto na curva P e sua ordem n. A chave privada d será um inteiro uniformemente aleatório escolhido tal que $1 < d < n - 1$ e a sua correspondente chave pública é calculada como $Q = dP$. Os

parâmetros (p, E, P, n) são de domínio público e o problema de se determinar d tendo o conhecimento de Q é denominado problema do logaritmo discreto em curvas elípticas.

Algoritmo 4 Geração de chaves no Algoritmo de Curvas Elípticas (21)

Entrada: Os parâmetros (p, E, P, n).
Saída: A chave pública Q e a chave privada d

1: Escolher um inteiro $d \in (1, n-1)$
2: $Q = dP$
3: **return** (Q, d)

Suponha agora que Alice queira enviar uma mensagem a Bob. Primeiramente, Alice transforma a mensagem m em um ponto M na curva elíptica. Tendo o ponto M, Alice cifra a mensagem adicionando ao ponto M o valor kQ, onde k é um inteiro uniformemente aleatório e Q é a chave pública de Bob. Além disso, deve-se calcular o valor kP, onde P é um ponto na curva. Alice então transmite a Bob os pontos $C_1 = kP$ e $C_2 = M + kQ$. Bob então ao receber esse conteúdo utiliza a sua chave privada d para obter a mensagem. Para isso Bob deve realizar a seguinte computação: $dC_1 = d(kP) = k(dP) = kQ$ e adicionar esse ponto ao valor C_2 recuperando então a mensagem M, pois $C_2 - kQ = M + kQ - kQ = M$.

Algoritmo 5 Cifração - Algoritmo de Curvas Elípticas (21)

Entrada: Os parâmetros (p, E, P, n), a chave pública Q e a mensagem m.
Saída: Mensagem cifrada (C_1, C_2)

1: Representar a mensagem m como um ponto na curva elítica $(M \in E(\mathbb{F}_p))$
2: Escolher um k uniformemente variado tal que $k \in [1, n-1]$
3: $C_1 = kp$
4: $C_2 = M + kQ$
5: **return** (C_1, C_2)

Algoritmo 6 Decifração - Algoritmo de Curvas Elípticas (21)

Entrada: Os parâmetros (p, E, P, n), a chave privada d e a mensagem cifrada (C_1, C_2).
Saída: Mensagem m

1: $M = C_2 - dC_1$
2: **return** (m)

3.4.2 Assinatura ECDSA

É um protocolo de assinatura digital que nada mais é do que uma variante baseada em curvas elípticas de outro protocolo chamado DSA - *Digital Signature Algorithm* que é um algoritmo proposto pelo NIST.

O ECDSA (*Elliptic Curve Digital Signature Algorithm*) funciona da seguinte forma, primeiramente é necessário a especificação de alguns parâmetros públicos $(E(\mathbb{F}_p), A, q, H)$ onde $E(\mathbb{F}_p)$ é uma curva elíptica e A é um ponto de ordem q em E e H é uma função de

Algoritmo 7 Geração de chaves no DSA (13) com adaptações.

Entrada: Parâmetros públicos $(E(\mathbb{F}_p), A, q)$.
Saída: A chave pública $e = (p, \alpha, \beta)$ e a chave privada $d = (a)$

1: Escolha o valor de d tal que $1 < d < n - 1$
2: $Q \leftarrow dG$
3: **return** (Q) e (d)

ECDSA	
Alice	**Bob**
Assinatura	
1. Escolhe uma mensagem m 2. Escolhe um k aleatório 3. Utiliza a sua chave privada d para calcular a assinatura (r, s): $kA = (u, v)$, $r = u \mod q$ e $s = (h(m) + dr)k^{-1}) \mod q$ 4. Manda a mensagem e a assinatura $c = (m, (r, s))$ para Bob	
Verificação	
	1. Recebe a mensagem e a assinatura $c = (m, (r, s))$ 2. Utiliza a chave pública de Alice (B) para calcular $s' = s^{-1} \mod q$ e $(u, v) = (h(m)s')A + (rs')B$. 3. Realiza a verificação autenticando se e somente se $u \mod q = r$

Tabela 3.1: ECDSA

resumo criptográfico. Em seguida deve ser realizada a geração das chaves, a qual, deve ter como entrada os parâmetros públicos $(E(\mathbb{F}_p), A, q)$. A chave privada d é escolhida aleatoriamente no intervalo entre 2 e $n - 2$, ou seja, $d \in (1, n - 1]$ e a chave pública B é obtida realizando a operação $B = dA$. Suponha agora que Alice deseja assinar uma mensagem m. Para tanto, ele deve escolher um k inteiro uniformemente aleatório e deve calcular a assinatura (r, s), onde $kA = (u, v)$, $r = u \mod q$, $s = (h(m) + dr)k^{-1}) \mod q$. Alice então envia para Bob a mensagem e a assinatura. Agora suponha que Bob deseja verifica se a assinatura gerada por Alice é uma assinatura válida. Então, Bob deve calcular $s' = s^{-1} \mod q$ e $(u, v) = (h(m)s')A+(rs')B$ com a chave pública B de Alice para validar a assinatura. A assinatura será uma assinatura válida se e somente se $u \mod q = r$.

3.5 Emparelhamentos bilineares

Sejam $\mathbb{G}_1$ e $\mathbb{G}_2$ grupos cíclicos aditivos e $\mathbb{G}_T$ um grupo cíclico multiplicativo tais que $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T|$. Seja P o gerador de $\mathbb{G}_1$ e o gerador de $\mathbb{G}_2$. Um mapeamento $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ é dito um emparelhamento bilinear se:

1. Dados $(V, W) \in \mathbb{G}_1 \times \mathbb{G}_2$, temos:

$$e(P, Q + Z) = e(P, Q) \cdot e(P, Z) \tag{3.3}$$

$$e(P + V, Q) = e(P, Q) \cdot e(V, Q) \tag{3.4}$$

2. $e(P, Q) \neq 1_{\mathbb{G}_T}$, onde $1_{\mathbb{G}_T}$ é a identidade do grupo $\mathbb{G}_T$

3. O mapeamento e deve ser ter complexidade de ordem polinomial.

O emparelhamento bilinear era utilizado anteriormente para atacar sistemas criptográficos de curvas elípticas ao reduzir os logaritmos da curva para logaritmos em corpos finitos (20) facilitando a realização dos cálculos por se tratar de uma estrutura algébrica mais rica. Porém, com a descoberta da utilidade do emparelhamento para realizar a cifração baseada em identidade, um repertório de novos protocolos foi desenvolvido. A idéia de sistemas baseados em identidade é se aproveitar da existência de informações públicas autênticas para simplificar a autenticação de chaves públicas.

No entanto, o custo de técnicas criptográficas baseadas no emparelhamento bilinear ainda é significativamente mais caro que as técnicas alternativas. Porém, por ser uma área de pesquisa nova, os algoritmos para cálculo do emparelhamento ainda se encontram em aperfeiçoamento, estando longe de ter atingido um limite computacional, haja em vista o grande esforço em pequisa que esta sendo investido nessa área.

3.5.1 Acordo de chaves não-interativo

Umas das aplicações dos emparelhamentos bilineares sobre curvas elípticas é o acordo de chaves não-interativo baseado em identidades. A ideia dos criptossistemas baseados em identidade é de aproveitar a autenticidade de informações publicamente conhecidas para simplificar o acordo de chaves.

O acordo de chaves proposto por Sakai et al. (25) funciona da seguinte maneira. Suponha que Alice queira trocar um par de chaves com Bob. Primeiramente, uma autoridade central gera uma chave mestra s. Em seguida, Alice, com uma informação pública relativa à sua identidade (ID_{Alice}), calcula $P_{Alice} = h(ID_{Alice})$ e, a partir dessa informação, a autoridade central calcula então a chave privada $S_{Alice} = sP_{Alice}$.

Bob realiza o mesmo procedimento, calculando $P_{Bob} = h(ID_{Bob})$ a partir de uma informação relativa à sua identidade. A autoridade central fornece então a chave privada $S_{Bob} = sP_{Bob}$. Finalmente, Alice e Bob derivam a mesma chave a partir das informações públicas, calculando $e(S_{Alice}, P_{Bob}) = e(P_{Alice}, S_{Bob})$.

Acordo de chaves não-interativo (Sakai-Ohgishi-Kasahara)	
Alice	**Bob**
Inicialização	
1. Autoridade Central gera chave mestra s	
Geração de chaves	
1. Com identidade ID_{Alice} calcula $P_{Alice} = h(ID_{Alice})$ 2. Autoridade fornece chave privada $S_{Alice} = sP_{Alice}$	1. Com identidade ID_{Bob} calcula $P_{Bob} = h(ID_{Bob})$ 2. Autoridade fornece chave privada $S_{Bob} = sP_{Bob}$
Derivação de chaves	
1. Alice e Bob então derivam a mesma chave $e(S_{Alice}, P_{Bob}) = e(P_{Alice}, S_{Bob})$	

Tabela 3.2: Acordo de chaves não-interativo

Capítulo 4

Aritmética em corpos primos

A implementação eficiente de aritmética em corpos finitos é um pré-requisito importante para a otimização de criptossistemas baseados em curvas elípticas porque as operações na curva são realizadas utilizando essas operações. Dessas, as mais importantes para os protocolos de curvas elípticas e de criptografia baseada em emparelhamentos é o cálculo da multiplicação e do quadrado, pois cerca de 75% do tempo é gasto realizando essas operações (13). Neste capítulo serão discutidos implementações eficientes em *software* para as operações aritméticas de adição (6.1.1), subtração (6.1.3), multiplicação (6.3), quadrado (4.6), exponenciação (4.8), redução modular (6.4) e inversão (4.9) em corpos primos.

4.1 Corpo primo

Seja p um número primo. Os inteiros {0, 1, 2, ... , p - 1} com as operações de adição e multiplicação modulares é um corpo primo denotado por $\mathbb{F}_p$. Denominamos p como o módulo de $\mathbb{F}_p$. Além disso, para todo inteiro a, a redução de a módulo p, denotada a mod p, é um único inteiro r obtido calculando-se o resto da divisão de a por p.

4.2 Representação na base b

Os números podem ser representados em qualquer base númerica, sendo que a representação mais utilizada é a da base 10. Em qualquer base numérica b, o valor do i-ésimo dígito a é: $a \cdot b^i$. A representação de um número inteiro positivo como a soma de multiplicação de potências b é chamada de representação na base b. Por exemplo, o número 256 na base 10 pode ser escrito como $a = 2 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0$. Além disso, nesta representação geralmente se denomina o dígito mais à esquerda de dígito mais significativo (dígito a_n) é o dígito mais à direita de dígito menos significativo (dígito a_0).

Figura 4.1: Representação do número $a \in \mathbb{F}_p$ como um vetor de palavras de w *bits* (16)

Seja W o tamanho da palavra suportada pela arquitetura de *hardware* da plataforma. Atualmente, a maioria das plataformas é de 64 ou 32 *bits*, porém, sistemas que necessitam de um baixo consumo de energia, como alguns sistemas embarcados, possuem arquiteturas de 16 ou 8 *bits*. Como no corpo primo $\mathbb{F}_p$ os seus elementos são números inteiros de 0 até $p-1$, podemos representar um número multiprecisão $a \in \mathbb{F}_p$ como um vetor de palavras de W *bits* de tamanho t, onde $t = \lceil m/W \rceil$, sendo m o comprimento em *bits* de a. Desse modo, poderiamos representar a como $a = 2^{(t-1)W}A[t-1] + ... + 2^{2W}A[2] + 2^{W}A[1] + A[0]$.

4.3 Adição

Primeiramente, para realizar a adição multiprecisão de dois números é necessário que ambos estejam representados em uma mesma base númerica e possuam o mesmo comprimento. Para que os números tenham o mesmo comprimento, basta adicionar '0's à esquerda do menor número.

O algoritmo para adição multiprecisão recebe como parâmetros de entrada dois inteiros a e b representados como um vetor de palavras de W *bits* cada, tal que $a, b \in [0, 2^{Wt})$ e retorna como saída o inteiro c tal que $c = a + b \ mod \ 2^{Wt}$ e um *bit* ε denominado *bit* de *carry*. Primeiramente, é realizada a adição dos dígitos menos significativos de cada operando, i.e, $C[0] = A[0] + B[0]$. Se o valor de $C[0]$ for maior que 2^W, então o *bit* de *carry* (ε) é atribuido como 1 e o valor de $C[0]$ é reduzido módulo 2^W. Caso contrário, o *bit* de *carry* (ε) é atribuido como 0. Em seguida, as demais posições dos vetores são somadas adicionando-se também o valor do *bit* de *carry*, i.e., $C[i] = A[i] + B[i] + \varepsilon$. Lembrado-se que o *bit* de *carry* é atribuido como 1 se o valor de $C[i]$ for maior que 2^W e $C[i]$ é reduzido módulo 2^W. Ao final das adições, se o *bit* de *carry* for igual a 1, deve-se subtrair p de c.

Algorithm 8 Adição Multiprecisão em $\mathbb{F}_p$ (16).

Entrada: Módulo p, e inteiros $a,\ b \in [0, p-1]$.
Saída: $c = (a + b) \ mod \ p$

1: $(\varepsilon, C[0]) \leftarrow A[0] + B[0]$, onde ε é o *bit* de *carry*
2: **for** $i = 1$ **downto** $t-1$ **do**
3: $\quad (\varepsilon, C[i]) \leftarrow A[i] + B[i] + \varepsilon$
4: **end for**
5: **if** $\varepsilon = 1$ **then**
6: $\quad$ Subtraia p de c $= (C[t-1], ..., C[2], C[1], C[0])$
7: **end if**
8: **return** (c)

No entanto, após as devidas operações, pode ocorrer o caso em que o valor final de c é maior que o módulo p. Nesse caso, é necessária a realização de uma subtração adicional, ou seja, $c = c - p$ para se obter um inteiro $c \in [0, p-1]$.

Para maior eficiência do ponto de vista computacional a base b deve escolhida de uma maneira tal que $(a_i + b_i + \varepsilon) \ mod \ p$ possa ser computado pelo *hardware* do dispositivo (21). Além disso, alguns processadores possuem em seu conjunto de instruções operações de adição com *carry*.

4.4 Subtração

Para a subtração de números em multipla precisão, assim como na adição, é necessário que os operando tenham o mesmo tamanho. O algoritmo para a operação de subtração multiprecisão, aliás, é bem parecido com o algoritmo de adição. No entanto, nesta operação, o *bit* de *carry* é denominado de *borrow*.

Algorithm 9 Subtração Multiprecisão em $\mathbb{F}_p$ (16).

Entrada: Módulo p, e inteiros a, $b \in [0, p-1]$.
Saída: $c = (a - b)\ mod\ p$

```
1: (ε, C[0]) ← A[0] − B[0], onde ε é o bit de borrow
2: for i = 1 downto t − 1 do
3:     (ε, C[i]) ← A[i] − B[i] − ε
4: end for
5: if ε = 1 then
6:     Adicione p ao c = (C[t − 1], ...., C[2], C[1], C[0])
7: end if
8: return (c)
```

O algoritmo para subtração segue os mesmos passos do algoritmo da adição. Primeiro é realizada a subtração das posições menos significativas dos operandos, i.e, $C[0] = A[0] - B[0]$. Se o valor de $C[0]$ for maior que 2^W então o *bit* de *borrow* (ε) é atribuído com o valor 1 e o valor de $C[0]$ é reduzido módulo 2^W, senão o *borrow* é atribuído com o valor 0. Em seguida, as demais posições dos vetores são subtraídas com o valor do *bit* de *borrow*, i.e., $C[i] = A[i] - B[i] - \varepsilon$. Ao final das subtrações, se o *bit* de *borrow* for igual a 1, deve-se adicionar p ao valor de c.

4.5 Multiplicação

O algoritmo para a multiplicação multiprecisão recebe como parâmetros de entrada dois números inteiros a (multiplicando) e b (multiplicador) $\in [0, p-1]$ e retorna como saída um inteiro c (produto) tal que $c = a \cdot b$. Para que o algoritmo da multiplicação funcione corretamente, é necessário que os operandos estejam representados em um mesma base númerica x. Além disso, se o número a for representado por um vetor de palavras de W *bits* com *n+1* posições, i.e., $a = (a_n a_{n-1} ... a_1 a_0)_x$ e o número b for representado por um vetor com *t+1* posições, i.e., $b = (b_t b_{t-1} ... b_1 b_0)_x$. Então, o vetor que armazena o produto dos números $a \cdot b$ terá no máximo $(n + t + 2)$ posições na base x. Ou seja, uma observação a ser feita é que o número de dígitos do produto é muito maior do que o número de dígitos de qualquer um dos operandos. É importante mencionar também, que para os algoritmos de multiplicação utilizamos a notação $(U \cdot V)$ para denotar um número de (2W)-*bits* obtido como a concatenação dos W-*bits* dos números representados por U e V. Nas próximas subseções serão apresentadas três estratégias para implementar

a multiplicação multiprecisão: *Schoolbook* (4.5.1), Comba (6.3.1) e Karatsuba (6.3.3).

4.5.1 Schoolbook

O método mais fácil para se realizar a multiplicação é o algoritmo *Schoolbook*, que é o mesmo algoritmo aprendido na escola. Inicialmente, para realizar a multiplicação, todas as possíveis posições do vetor C que armazena o resultado da operação são inicializados com o valor 0. Em seguida, devemos fixar o primeiro dígito do multiplicador (iteração externa) e pegar os dígitos do multiplicando um a um, da direita para a esquerda, (iteração interna) calculando a multiplicação e deslocando a posição do produto intermediário um dígito para a esquerda dos produtos intermediários anteriores. No passo seguinte utiliza-se o próximo dígito do multiplicador e percorre-se novamente todos os dígitos do multiplicando. O produto final é obtido após percorrer todos os dígitos do multiplicador. Lembrando que para o conjunto dos corpos primos, é necessário um passo adicional para reduzir o produto final para um inteiro entre 0 e $p-1$.

Algorithm 10 Multiplicação Multiprecisão (*Schoolbook*) em $\mathbb{F}_p$. Com adaptações (21).

Entrada: Inteiros a, $b \in [0, p-1]$ contendo $n+1$ e $t+1$ dígitos na base x, respectivamente.
Saída: $c = a \cdot b \ mod \ p$.

1: **for** $i \leftarrow 0$ **to** $n+t+1$ **do**
2: $\quad c[i] \leftarrow 0$
3: **end for**
4: **for** $i \leftarrow 0$ **to** t **do**
5: $\quad w \leftarrow 0$
6: $\quad$ **for** $j \leftarrow 0$ **to** n **do**
7: $\quad\quad (U \cdot V) \leftarrow c[i+j] + a[i] \cdot b[j] + w$
8: $\quad\quad c[i+j] \leftarrow V$
9: $\quad\quad w \leftarrow U$
10: $\quad$ **end for**
11: $\quad c[i+n+1] \leftarrow U$
12: **end for**
13: $c \leftarrow c \ mod \ p$
14: **return** (c)

O cálculo de $C[i+j] + A[i] \cdot B[i] + U$ é denominado de operação do produto interno. Como $C[i+j]$, $A[i]$, $B[i]$ e U estão todos representados na base x, o resultado do produto interno possui no máximo $(x-1) + (x-1)^2 + (x-1) = x^2 - 1$ podendo dessa forma ser representado por dois vetores de palavras de W-*bit* denotado por (UV).

O algoritmo *Schoolbook* requer $(n+1)(t+1)$ multiplicações de precisão simples, dessa forma, a qualidade e eficácia da implementação dessa instrução é de suma importância para a eficiência do algoritmo como um todo (21).

Note que esse algoritmo nada mais é do que a implementação computacional do algoritmo utilizado desde o ensino primário. Observe no exemplo apresentado na figura (4.2) onde é mostrado o passo-a-passo do algoritmo *Schoolbook* para realizar a multiplicação do inteiro $A = 9274$ pelo inteiro $B = 847$ e o rasunho para o cálculo utilizando o método do

"papel e caneta". Repare que no algoritmo computacional, ao final de cada iteração (cinza) o resultado é o produto intermediário das operações anteriores que vai sendo acumulado.

i	j	w	$c_{i+j} + a_j b_i + \text{w}$	u	v	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	$0+28+0$	2	8	0	0	0	0	0	0	8
	1	2	$0+49+2$	5	1	0	0	0	0	0	1	8
	2	5	$0+14+5$	1	9	0	0	0	0	9	1	8
	3	1	$0+63+1$	6	4	0	0	6	4	9	1	8
1	0	0	$1+16+0$	1	7	0	0	6	4	9	7	8
	1	1	$9+28+1$	3	8	0	0	6	4	8	7	8
	2	3	$4+8+3$	1	5	0	0	6	5	8	7	8
	3	1	$6+36+1$	4	3	0	4	3	5	8	7	8
2	0	0	$8+32+0$	4	0	0	4	3	5	0	7	8
	1	4	$5+56+4$	6	5	0	4	3	5	0	7	8
	2	6	$3+16+6$	2	5	0	4	5	5	0	7	8
	3	2	$4+72+2$	7	8	7	8	5	5	0	7	8

			9	2	7	4	
			×	8	4	7	
		6	4	9	1	8	(row 1)
	3	7	0	9	6		(row 2)
7	4	1	9	2			(row 3)
7	8	5	5	0	7	8	

Figura 4.2: Exemplo de multiplicação multiprecisão utilizando o algoritmo *Schoolbook* (10). Com adaptações (21).

4.5.2 Comba

Em 1990 foi proposto por Paul G. Comba (8) uma variação do algoritmo *Schoolbook*. Nesse variante, o n-ésimo dígito do produto é computado multiplicando-se todos os pares de dígitos cujo somatório dos índices sejam equivalentes a n e somando cada resultado acumulado. Por exemplo, suponha que o algoritmo esteja na quarta iteração, ou seja $n = 4$. Dessa forma, a quarta posição do vetor de palavras de W *bits* que armazena o produto será computada como: $C[4] = A[0] \cdot B[4] + A[1] \cdot B[3] + A[2] \cdot B[2] + A[3] \cdot B[1] + A[4] \cdot B[0]$.

A cerne do algoritmo de multiplicação de Comba consiste na ideia de se multiplicar dois dígitos armazenados cada um em um vetor de palavras de W *bits* resultando em um inteiro de $2W$ *bits*, e em seguida somá-los em um acumulador de três dígitos, sendo que o terceito dígito é utilizado como um dígito que acumula os *carries* das somas intermediárias.

Veja que esse algoritmo nada mais é do que uma versão orientada a colunas do algoritmo *Schoolbook*. Logo para qualquer um dos algoritmos mostrados acima, para se realizar uma operação multiplicando dois inteiros de *n-bits* leva-se uma ordem de tempo quadrática ao número de *bits* do inteiro, ou seja, utilizando a notação da complexidade computacional $O(n^2)$.

O algoritmo Comba tende a ser mais eficiente do ponto de vista computacional do que o algoritmo *Schoolbook*, pois o primeiro executa menos operações de memória que o

Algorithm 11 Multiplicação Multiprecisão Comba em $\mathbb{F}_p$. Com adaptações (16).

Entrada: Inteiros a, $b \in [0, p-1]$.
Saída: $c = a \cdot b \bmod p$.

1: $r_0, r_1, r_2 \leftarrow 0$
2: **for** $k \leftarrow 0$ **to** $2t-2$ **do**
3:   **for each** $(i,j)|i+j=k, 0 \leq i,j \leq t-1$ **do**
4:     $(ur) \leftarrow a[i] \cdot b[j]$
5:     $(\varepsilon, r_0) \leftarrow r_0 + r$
6:     $(\varepsilon, r_1) \leftarrow r_1 + u + \varepsilon$
7:     $r_2 \leftarrow r_2 + \varepsilon$
8:   **end for**
9:   $c[k] \leftarrow r_0$
10:   $r_0 \leftarrow r_1$
11:   $r_1 \leftarrow r_2$
12:   $r_2 \leftarrow 0$
13: **end for**
14: $c[2t-1] \leftarrow r_0$
15: $c \leftarrow c \ mod \ p$
16: **return** (c)

segundo, pois boa parte das operações pode ser realizada em registradores, uma vez que os resultados intermediários são acumulando nas três variáveis.

4.5.3 Híbrido

Foi desenvolvido também um algoritmo híbrido (15) para a multiplicação que combina as vantagens dos algoritmos *Schoolbook* e Comba. O método híbrido mantém a vantagem da multiplicação Comba e tenta minimizar o número de instruções de leitura à memória do tipo *load* utilizando os registradores extras.

O método híbrido pode ser implementado utilizando dois laços, sendo o laço externo uma multiplicação *Schoolbook* e o laço mais interno uma multiplicação Comba, conforme a figura (4.3). Veja que os produtos parciais são processados seguindo uma separação em blocos, sendo que em cada bloco os operandos são processados seguindo o algoritmo *Schoolbook*. Dessa forma, pode-se economizar algumas instruções de acesso à memória do tipo *load* caso existam registradores disponíveis. No entanto, no laço externo do método híbrido os operandos são processados seguindo o algoritmo Comba. Dessa forma, entre dois blocos consecutivos não é possível aproveitar os operandos, sendo necessário carregar todos os operandos da memória novamente, pois não há o compartilhamento dos mesmos.

Porém, o método híbrido consegue reduzir o número de instruções de acesso à memória o que aumenta o desempenho, sendo aconselhavél o seu uso em arquiteturas que disponham de um número grande de registradores.

4.5.4 Karatsuba

O algoritmo Karatsuba foi inventado em 1960 por Anatolii Alexeevitch Karatsuba

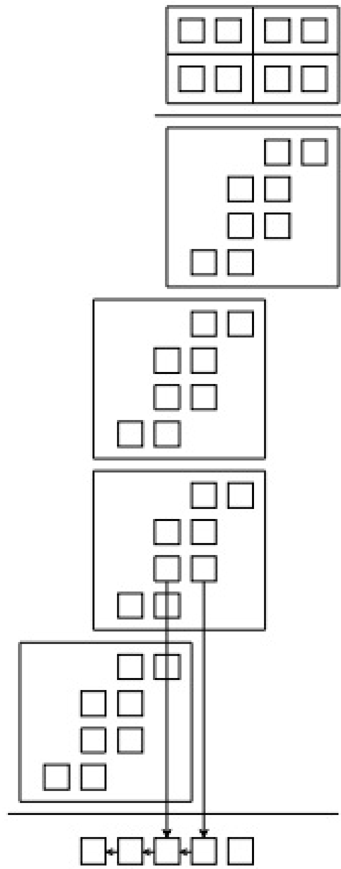

Figura 4.3: Multiplicação Híbrida

sendo publicado somente em 1962 (18). Esse algoritmo utiliza uma técnica de projeto e análise de algoritmos chamada de "dividir para conquistar". Esta técnica consiste em dividir um problema maior recursivamente em instâncias menores para que estes últimos possam ser resolvidos mais facilmente. Consequentemente, a solução do problema como um todo é construída por meio da combinação dos resultados das instâncias menores do problema. Graças a essa abordagem, foi possível reduzir o problema da multiplicação de números inteiros de ordem de tempo quadrática $O(n^2)$ para uma ordem de tempo sub-quadrática $O(n^{log_2^3})$.

A técnica desenvolvida por Karatsuba funciona de tal forma que é possível computar a multiplicação de dois números inteiros utilizando três operações de multiplicação de inteiros de metade do tamanho e mais duas operações de adição e subtração. Por exemplo, seja dois números inteiros de n *bits* denotados por A e B. Podemos decompor ambos números em duas partes $A = A_1 \cdot 2^{n/2} + A_0$ e $B = B_1 \cdot 2^{n/2} + B_0$. Dessa forma, podemos calcular o produto $C = A \cdot B$ como:

$$
\begin{aligned}
C &= A \cdot B \\
C &= (A_1 \cdot 2^{n/2} + A_0)(B_1 \cdot 2^{n/2} + B_0) \\
C &= A_1B_1 \cdot 2^n + [(A_0 + A_1)(B_0 + B_1) - A_1B_1 - A_0B_0] \cdot 2^{n/2} + A_0B_0 \qquad (4.1)
\end{aligned}
$$

Verifique que se for armazenado os resultados das operações A_1B_1 e de A_0B_0 na memória, é necessário apenas três multiplicações (as duas já mencionadas, mais a multiplicação $(A_0 + A_1)(B_0 + B_1)$), pois embora sejam necessárias cinco multiplicações, duas delas são repetidas, bastando realizar a operação uma única vez e guardar o resultado.

Esse método é vantajoso para números nos quais o valor de n *bits* é suficientemente grande, pois nestes casos, o custo das operações de adição e subtração pode ser considerado insignificante em comparação com o custo de se realizar uma operação de multiplicação. No entanto, para inteiros nos quais o valor de n não é tão grande assim, o *overhead* introduzido pela maior quantidade de operações pode fazer com que a adotação desse método não seja tão eficiente.

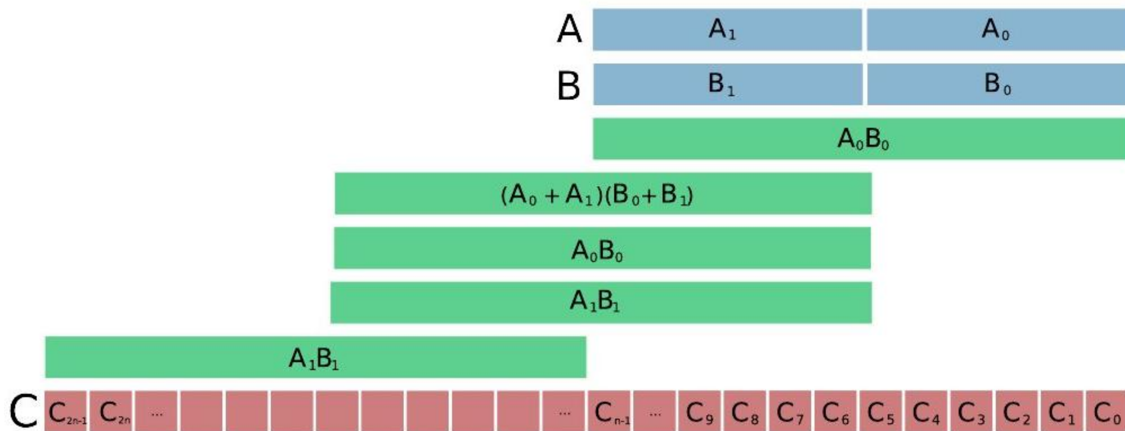

Figura 4.4: Multiplicação Karatsuba

4.6 Quadrado

A operação de quadrado pode ser implementada realizando uma pequena modificação no algoritmo Comba. Uma vez que, com alguns ajustes é possível implementar o quadrado mais rápido do que a multiplicação, pois consegue-se reduzir o número de multiplicações de precisão simples pela metade. Por exemplo, suponha que seja necessário a computação do quadrado do número a. Utilizando o algoritmo Comba para realizar a multiplicação, para o cálculo da quarta posição do vetor de palavras de W *bits* que armazena o produto deve-se realizar a seguinte operação: $C[4] = A[0] \cdot A[4] + A[1] \cdot A[3] + A[2] \cdot A[2] + A[3] \cdot A[1] + A[4] \cdot A[0]$. Note que no caso da operação de quadrado, não é necessário a realização das cinco operações de multiplicação, mas somente de três, porque a multiplicação é comutativa.

Para realizar a mudança explicada anteriormente, note que foi adicionado uma nova condição na quinta linha do algoritmo (12). Neste ponto, é verificado se o primeiro índice é menor que o segundo, se essa condição for verdadeira, o resultado é multiplicado por dois. Para ilustrar esse construção, verifique que no exemplo citado no parágrafo acima, na primeira iteração para o cálculo da quarta posição do produto teriamos a variável i

contendo o valor 0 e a variável j contendo o valor 4. Como $0 < 4$, o produto $A[0] \cdot A[4]$ seria multiplicado por dois, dessa forma não seria mais necessário calcular $A[4] \cdot A[0]$, pois este valor já foi inserido no cálculo ao realizar a multiplicação pela constante dois.

Algorithm 12 Quadrado Multiprecisão Comba em $\mathbb{F}_p$. Com adaptações (16).

Entrada: Inteiro $a \in [0, p-1]$.
Saída: $c = a^2 \bmod p$.

```
1:  r0, r1, r2 ← 0
2:  for k ← 0 to 2t − 2 do
3:    for each (i, j)|i + j = k, 0 ≤ i ≤ j ≤ t − 1 do
4:      (ur) ← a[i] · a[j]
5:      if (i < j) then
6:        (ε, ur) ← (ur) · 2
7:        r2 ← r2 + ε
8:      end if
9:      (ε, r0) ← r0 + r
10:     (ε, r1) ← r1 + u + ε
11:     r2 ← r2 + ε
12:   end for
13:   c[k] ← r0
14:   r0 ← r1
15:   r1 ← r2
16:   r2 ← 0
17: end for
18: c[2t − 1] ← r0
19: c ← c mod p
20: return (c)
```

Visando uma maior eficiência do ponto de vista computacional, deve-se implementar a multiplicação por dois no algoritmo (12) utilizando a operação de deslocamento com *carry* se a plataforma na qual está sendo realizada a implementação disponibilizar tal instrução; ou então, utilizar adições com *carry*.

4.7 Redução modular

A operação de redução modular consiste em calcular o resto da divisão de um inteiro por um determinado módulo. No entanto, o algoritmo mais simples para realizar essa cálculo em inteiros de múltipla precisão consiste em realizar uma operação de divisão entre o inteiro e o módulo sendo o resultado o resto obtido dessa divisão. No entanto, assim como em precisão simples, a divisão multiprecisão é uma operação custosa exigindo um maior esforço computacional devido à grande quantidade de operações de subtração e multiplicação.

Para os algoritmos (8) e (9), a redução modular é simples, porque o valor do inteiro resultante é próximo do módulo p. No caso da adição, se essa operação produzir um número maior que o inteiro p basta subtrair uma vez o valor módulo p do resultado

para que este resulte em um número que esteja no conjunto dos corpos primos $\mathbb{F}_p$. Na subtração, tem-se um caso similar ao da adição, pois se o resultado da subtração for negativo, basta somar a esse número o valor do módulo p. Note que nesses dois casos a operação de redução modular foi realizada de maneira simples porque o resultado das operações em questão e bem próximo do valor do módulo. No entanto, em operações como a multiplicação multiprecisão em que o valor do produto geralmente é muito maior que o módulo, não é possível realizar operações simples como ocorre com a adição e a subtração, sendo necessário no algoritmo mais simples a realização de uma divisão multiprecisão.

Para tentar resolver esse problema, foram propostas algumas técnicas que substituem a custosa operação de divisão por operações mais simples. A redução de Montgomery (22) por exemplo, é uma técnica que realiza uma transformação no inteiro em questão permitindo que a redução multiprecisão seja implementada com maior eficiência, pois troca a operação de divisão por quatro operações de multiplicação.

Algorithm 13 Redução modular em $\mathbb{F}_p$ (16).

Entrada: Primo p e inteiro a
Saída: $c = a \ mod \ p$

1: **for** $i \leftarrow 0$ **to** $t-1$ **do**
2: $\quad c \leftarrow c + \mu n 2^{wi}$
3: **end for**
4: $c \leftarrow c/2^{wi} = c/R$
5: **while** $c \geq p$ **do**
6: $\quad c \leftarrow c - p$
7: **end while**
8: **return** (c)

Por exemplo, supondo-se que seja necessário a realização do seguinte cálculo: $r = a \cdot b \pmod p$, i.e., uma operação de multiplicação seguida de uma redução modular. Para se evitar que seja utilizada uma operação de divisão, utiliza-se a redução de Montgmery. Primeiramente, deve-se escolher um número R tal que $R = 2^{|p|}$, sendo p o módulo. Dessa forma, tem-se o conhecimento de que $mdc(R, p) = 1$ e portanto R tem inverso módulo p. Em seguida, deve-se converter os operandos utilizados para a forma de Montgomery, essa conversão é realizada multiplicando o operando pelo inteiro R, ou seja, $\overline{a} = a \cdot R \pmod p$ e $\overline{b} = b \cdot R \pmod p$. Tendo os operandos transformados para a forma de Montgomery, podemos realizar a seguinte multiplicação para obter o produto r mantendo a forma de Montgomery:

$$\begin{aligned}
\overline{r} &= \overline{a} \cdot \overline{b} \cdot R^{-1} \pmod p \\
&= (a \cdot R) \cdot (b \cdot R) \cdot R^{-1} \pmod p \\
&= a \cdot R \cdot b \pmod p \\
&= a \cdot b \cdot R \pmod p \qquad (4.2)
\end{aligned}$$

Finalmente, pode-se calcular o valor de $r = a \cdot b \pmod p$ em termos de $\overline{r}$ realizando a seguinte operação: $r = \overline{r} \cdot R^{-1} \pmod p$. Verifique que, ao realizar a transformação para a forma de Montgomery, pode-se trocar uma divisão por quatro multiplicações.

Para aumentar a eficiência do algoritmo, pode-se antecipar ou adiar as conversões e realizar reduções seguidas. Além disso, o inverso de R módulo p pode ser pré-calculado.

4.8 Exponenciação

O algoritmo utilizado para o cálculo da exponenciação multiprecisão se baseia na seguinte recursão:

$$a^k = \begin{cases} 1, & \text{se} \quad k = 0, \\ a \cdot \left(a^{\frac{k-1}{2}}\right)^2, & \text{se} \quad k \equiv 1 \pmod 2, \\ \left(a^{\frac{k}{2}}\right)^2, & \text{se} \quad k \equiv 0 \pmod 2) \end{cases}$$

Algorithm 14 Exponenciação Modular (16).

Entrada: Inteiros $a,\ e,\ n$

Saída: $c = a^e \ mod \ n$

1: $c \leftarrow 1$
2: **for** $i \leftarrow |e| - 1$ **downto** 0 **do**
3: $\quad c \leftarrow c^2 \ mod \ n$
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad c \leftarrow c \cdot a \ mod \ n$
6: $\quad$ **end if**
7: **end for**
8: **return** (c)

Este algoritmo funciona, pois a iteração presente na linha 2 realiza de modo implícito a operação $a_i = a^{e_1 2^i} \mod n$. Sendo obtido ao final da iteração o valor de $a^e \mod n$, pois

$$c = \prod_{i=0}^{|e|-1} a_i = \prod_{i=0}^{|e|-1} a^{e_i 2^i} \mod n = a^{\sum e_i 2^i} \mod n = a^e \mod n$$

4.9 Inversão

A operação de inversão de um número inteiro a módulo p, denotado por $a^{-1} \pmod p$, sendo a um inteiro diferente de zero e $a \in \mathbb{F}_p$, tem como resultado um único elemento x tal que $x \in \mathbb{F}_p$ e que satisfaça a seguinte condição: $ax \equiv 1 \pmod p$. Inversos podem ser calculados classicamente utilizando o algoritmo estendido de Euclides (16), ou com uma outra abordagem mais eficiente utilizando o algoritmo binário (18).

4.9.1 Algoritmo Estendido de Euclides

O Algoritmo de Euclides pode ser considerado como um dos algoritmos mais antigos, tendo surgido por volta de 300 a.C. É utilizado para se calcular de modo simples o máximo

divisor comum de dois números. Sejam a e b inteiros não ambos nulos. O máximo divisor comum (mdc) de a e b, denotado por $mdc(a, b)$ é o maior inteiro d que divide ambos os números sem deixar resto. Esse algoritmo baseia-se no teorema que afirma que o $mdc(a, b) = mdc(b - ca, a)$ para todos os inteiros c. O cálculo do máximo divisor comum é importante para a operação de inversão, pois existe um outro teorema que diz que para dois números inteiros a e p com $p > 0$, o inverso de a módulo p existe se e somente se $mdc(a, p) = 1$.

Para se obter o valor do máximo divisor comum entre dois números inteiros a e b, onde $b \geq a$ utilizando o algoritmo de Euclides, primeiramente, deve-se dividir o inteiro b por a obtendo com essa operação o quociente q e o resto r. Ou seja, obtém-se que $b = qa + r$, sendo r um número positivo menor que a. Rearranjando essa equação, tem-se que $r = b - qa$. Agora, aplicando o teorema mencionado acima, concluímos que $mdc(a, b) = mdc(r, a)$. Isto é, reduzimos o problema de se determinar o $mdc(a, b)$ para o problema de se determinar o $mdc(r, a)$, tendo como vantagem a redução do tamanho dos parâmetros de entrada. Esse processo é repetido até que o valor do primeiro inteiro seja igual a zero, porque dessa forma pode-se obter o máximo divisor comum de forma direta, pois $mdc(0, d) = d$.

Algorithm 15 Algoritmo de Euclides (21).

Entrada: Dois inteiros não negativos a e b tal que $a \geq b$
Saída: $d = mdc(a, b)$

1: **while** $b \neq 0$ **do**
2: $\quad r \leftarrow a \mod b$
3: $\quad a \leftarrow b$
4: $\quad b \leftarrow r$
5: **end while**
6: **return** (a)

Do ponto de vista da eficiência computacional, pode-se provar que esse algoritmo termina porque o resto da operação de divisão está sempre diminuindo. Além disso, o Algoritmo Estendido de Euclides tem ordem de tempo $O(n^2)$ (21).

O algoritmo de Euclides pode ser estendido para não somente calcular o mdc de dois números, mas também obter dois inteiros, x e y tal que $ax + by = d$, sendo d o máximo divisor comum de a e b, ou seja, $d = mdc(a, b)$. Dessa forma o algoritmo retornaria uma tripla de inteiros (d, x, y), onde $d = mdc(a, b)$ e $d = ax + by$. Podemos rearranjar a segunda equação obtendo $ax = by - d$, ou seja $ax \equiv d \pmod b$. Dessa forma, quando o $mdc(a, b) = 1$, o inteiro x será o inverso de a módulo b ($a^{-1} \mod b$).

A extensão do Algoritmo de Euclides mantém a mesma ordem de tempo da versão clássica, i.e., $O(n^2)$ (21). No entanto, repare que para o cálculo da inversa é utilizado uma divisão, que como explicado anteriormente, é uma operação relativamente cara do ponto de vista computacional, devendo por esse motivo ser evitada sempre que possível. Por isso foi proposto um outro método para a inversão modular que não requer a utilização de divisões.

Algorithm 16 Inversão em $\mathbb{F}_p$ usando o Algoritmo Estendido de Euclides (16).

Entrada: Primo p e inteiro $a \in [1, p-1]$
Saída: $c = a^{-1}\ mod\ p$

1: $u \leftarrow a$
2: $v \leftarrow p$
3: $x_1 \leftarrow 1$
4: $x_2 \leftarrow 0$
5: **while** $u \neq 1$ **do**
6: $q \leftarrow \lfloor v/u \rfloor$
7: $r \leftarrow v - qu$
8: $x \leftarrow x_2 - qx_1$
9: $v \leftarrow u$
10: $u \leftarrow r$
11: $x_2 \leftarrow x_1$
12: $x_1 \leftarrow x$
13: **end while**
14: $c \leftarrow x_1\ mod\ p$
15: **return** (c)

4.9.2 Algoritmo de inversão binário

Uma desvantagem de se calcular o inverso módulo primo utilizando o Algoritmo Estendido de Euclides é que este requer a utilização da custosa operação de divisão. Para contornar esse problema, foi proposto por Josef Stein (31) o algoritmo de inversão binário que elimina a operação de divisão substituindo-a por operação mais simples, como a adição e o deslocamento.

Para o cálculo do máximo divisor comum, o algoritmo de inversão binário, assim como o algoritmo de Euclides, tenta reduzir o problema até o ponto em que seja possível encontrar um resultado de maneira direta. Isto ocorre quando se chega à identidade $mdc(0, d) = d$, pois todos os inteiros dividem o número zero e d é o maior número que divide ele mesmo. Além desse fato, o algoritmo em questão também se baseia em outras três identidades.

Primeiramente, é verificado se os dois parâmetros de entrada são pares, ou seja, podem ser divididos por dois. Se for o caso, então pode-se reduzir o problema de tal forma que $mdc(a, b) = 2 \cdot mdc(a/2, b/2)$, porque o número dois é um divisor comum. Este procedimento é realizado até que um dos inteiros seja ímpar. Em um segundo passo, é verificado qual dos parâmetros continua sendo par, pois neste caso pode-se continuar com a redução do problema, porque se o inteiro a for par e b ímpar então $mdc(a, b) = mdc(a/2, b)$ uma vez que o número dois não é mais um divisor comum. De modo similar, se o inteiro b for par e a ímpar então $mdc(a, b) = mdc(a, b/2)$. Finalmente, se ambos inteiros a e b forem ímpares e $a \geq b$, então $mdc(a, b) = mdc((a-b)/2, b)$, se $a < b$, então $mdc(a, b) = mdc((b-a)/2, a)$. Esses procedimentos são repetidos até que seja possível o cálculo do máximo divisor comum de maneiro direta, i.e., quando $a = 0$.

Repare que neste algoritmo, ao contrário do algoritmo de Euclides, não é utilizado nenhuma divisão, embora o algoritmo binário necessite de mais passos para computar o

mdc. É importante notar que se os inteiros em questão estiverem representados na base 2, então a divisão por dois nada mais é do que uma simples operação de deslocamento para a direita.

Algorithm 17 Algoritmo binário para o cálculo do máximo divisor comum (16).

Entrada: Dois inteiros não negativos a e b
Saída: $d = mdc(a, b)$

```
 1: u ← a
 2: v ← b
 3: e ← 1
 4: while (u (mod 2) = 0 and v (mod 2) = 0) do
 5:    u ← u/2
 6:    v ← v/2
 7:    e ← 2e
 8: end while
 9: while u ≠ 0 do
10:    while (u (mod 2) = 0) do
11:       u ← u/2
12:    end while
13:    while (v (mod 2) = 0) do
14:       v ← v/2
15:    end while
16:    if u ≥ v then
17:       u ← u − v
18:    else
19:       v ← v − u
20:    end if
21: end while
22: return (e · v)
```

Pode-se utilizar o algoritmo binário para computar a inversão de inteiros multiprecisão módulo primo. Para isto, é necessário uma versão estendida análoga à versão do Algoritmo Estendido de Euclides. Utilizando-se o algoritmo de inversão binária, as únicas operações multiprecisão necessárias são a adição e a subtração, pois a divisão por 2, como já foi dito, pode ser implementada como um deslocamento para a direita se os inteiros estiverem representados na base 2.

Algorithm 18 Inversão em $\mathbb{F}_p$ usando o algoritmo binário (16).

Entrada: Primo p e inteiro $a \in [1, p-1]$

Saída: $c = a^{-1}$ *mod* p

```
1:  u ← a, v ← p
2:  x1 ← 1, x2 ← 0
3:  while u ≠ 1 and v ≠ 1 do
4:     while (u (mod 2) = 0) do
5:        u ← u/2
6:        if (x1 (mod 2) = 1) then
7:           x1 ← x1/2
8:        else
9:           x1 ← (x1 + p)/2
10:       end if
11:    end while
12:    while (v (mod 2) = 0) do
13:       v ← v/2
14:       if (x2 mod 2 = 1) then
15:          x2 ← x2/2
16:       else
17:          x2 ← (x2 + p)/2
18:       end if
19:    end while
20:    if u ≥ v then
21:       u ← u − v, x1 ← x1 − x2
22:       v ← v − u, x2 ← x2 − x1
23:    end if
24: end while
25: if u = 1 then
26:    return (x1 mod p)
27: else
28:    return (x2 mod p)
29: end if
```

Capítulo 5

Plataforma ARM

Os avanços significativos na área de comunicação digital sem fio e o contínuo desenvolvimento de dispositivos portáteis com capacidade de processamento cada vez maior é a combinação perfeita para a popularização da computação móvel. Fato este que pode ser notado pelo número crescente de *smartphones, netbooks, tablets* e outros dispositivos móveis disponíveis no mercado.

Dentre esse dispositivos, os *smartphones* e os *tablets* têm obtido grande sucesso no mercado por garantir acesso a avançadas capacidades de computação, aliadas à mobilidade e portabilidade. Existe um número crescente de usuários comuns e empresas que utilizam esses aparelhos não somente como ferramentas de comunicação, mas também como um meio de armazenar, planejar e organizar o seu trabalho e/ou a sua vida cotidiana. Por esse motivo, esse tipo de aparelho tem se tornado um novo alvo para indivíduos maliciosos, pois de fato esses dispositivos guardam uma quantidade relativamente grande de informação sensível que deve ser protegida para garantir a sua integridade e o seu sigilo.

Figura 5.1: Dispositivos Móveis

No entanto, implementações criptográficas em computação móvel precisam de cuidados específicos devido às limitações inerentes desses tipos de dispositivos. O acesso móvel a

Internet por meio das redes 3G é normalmente mais lento do que conexões a cabo. Além de as tecnologias de rede sem fio, em comparação às redes cabeadas, serem mais sensíveis em relação à segurança, especialmente se essa transmissão ocorrer em uma longa área. Esse fato ocorre, em tese, devido ao meio de transmissão estar “acessível"a todos, fazendo com que sejam um alvo fácil para usuários maliciosos.

Um dos grandes problemas da computação móvel é o consumo de energia. Todos os dispositivos são alimentados por uma bateria, que é o um dos componentes mais caros. E, ao contrário dos computadores convencionais que são projetados para ficar ligados a uma fonte estática, os dispositivos móveis são projetados para garantir mobilidade aos usuários, tendo em contrapartida o custo de ficarem dependentes de uma constante recarga de energia. É importante ressaltar ainda que as baterias têm sua capacidade limitada, em muitos casos devido ao tamanho do dispositivo e em muitas ocasiões por ser difícil ou impossível realizar uma recarga.

Outro fator crítico é a capacidade de recursos. Pela necessidade de serem portáteis, esses tipos de dispositivos precisam ser relativamente pequenos e leves. Para tornar isso viável, parte do poder de processamento e quantidade de memória e armazenamento desses dispositivos é sacrificado tornando-os muitas vezes inferior aos computadores tradicionais.

5.1 Arquitetura RISC

Grande parte dos *smartphones* e *tablets* rodam sobre uma arquitetura denominada ARM (*Advenced RISC Machine*). Essa arquitetura é uma arquitetura de processador de 32 bits do tipo RISC (*Reduced Instruction Set Computer*) possuindo dessa forma um conjunto reduzido de instruções.

Arquiteturas do tipo RISC foram desenvolvidas visando criar um conjunto pequeno de instruções simples, mas poderoso que pudesse ser executado rapidamente, geralmente em um único ciclo. Essa arquitetura tem como princípio reduzir a complexidade das instruções realizadas pelo *hardware*, pois é mais fácil deixar a complexidade ao nível de *software*.

Arquiteturas CISC (*Complex Instruction Set Computer*), em oposição à arquitetura RISC, possuem um conjunto de instruções complexo. Arquiteturas CISC possuem um conjunto de instruções muito maior se comparado a arquiteturas RISC. Além disso, muitas dessas instruções são bastante complicadas, executando múltiplas operações em uma única instrução. Dessa forma uma vantagem da arquitetura CISC é que já existem instruções complexas no processador, o que facilita o trabalho do programador. O problema é que a implementação desse conjunto de instruções complexas aumenta também a complexidade do *hardware*.

A arquitetura RISC foi idealizada levando em conta características como: 1) processador com um número reduzido de instruções ficando a cargo do compilador ou do programador realizar a combinação dessas instruções para produzir operações mais complexas. 2) banco de registradores de próposito geral, ou seja, qualquer registrador pode armazenar um dado ou um endereço. 3) operações sempre realizada sobre dados contidos em registradores, sendo necessário para isso instruções *load* e *store* para transferir dados entre os registradores e a memória. 4)*pipeline*, isto é, o processamento da instrução é dividido em partes menores, podendo assim ser executadas em paralelo.

5.2 Arquitetura ARM

A arquitetura ARM foi desenvolvida tendo em vista algumas especificidades da plataforma alvo. Sistemas móveis, como já dito, precisam ter uma bateria como fonte de energia. Os processadores ARM foram projetados para reduzir o consumo de energia, estendendo dessa forma o período útil do dispositivo antes de uma recarga. Além disso, um outro empecilho é a quantidade de memória existente, que em muitos casos é restringida pelo tamanho físico do dispositivo. Por esse motivo, aumentar a densidade do código seria útil nesses tipos de aplicação pois economizaria a memória em sistemas que já possuem uma quantidade limitada. Ainda levando em conta o tamanho físico do dispositivo, um outro requisito importante é que quanto menor a área ocupada pelo processador, mais espaço poderia ser utilizado para armazenar dispositivos periféricos.

Visando um melhor desempenho em sistemas embarcados, a arquitetura ARM não segue a risca todos os conceitos pregados pela arquitetura RISC (30). Mesmo porque o objetivo não é somente uma grande velocidade natural do processador, mas um desempenho eficiente do sistema como um todo e o consumo de energia. Por exemplo, nem todas as instruções em ARM são executadas em um único ciclo. Instruções que realizam múltiplos *load-store* podem variar no número de ciclos dependendo do número de registradores envolvidos na operação. Veja que esse instrução múltipla aumenta a densidade do código uma vez que esse tipo de operação é comum no início e no final das funções. Outra exemplo de instrução contida na arquitetura ARM que acaba não seguindo os princípios da arquitetura RISC é a que possibilita o deslocamento (*shift*) de um operando em registrador antes de ser realizada a operação, sendo que todo esse processamento é realizado em uma única instrução. As execuções condicionais, outro aspecto adicionado à arquitetura ARM, permitem que uma instrução seja executada se e somente se uma determinada condição for satisfeita. Essa adição tem como vantagem a redução potencial de instruções de desvio condicional, embora diferencie um pouco das definições RISC.

5.2.1 Banco de registradores e execuções condicionais

Essa arquitetura dispõe de dezesseis registradores de uso geral, embora três desses registradores tenham funções específicas:

- O registrador 13 (`r13`) é o ponteiro de pilha (*sp - stack pointer*) e armazena o endereço do início da pilha;

- O registrador 14 (`r14`) armazena o endereço de retorno da função (*lr - link register*) e guarda o endereço de retorno de uma função para possibilitar que o código volte a ser executado do ponto exato em que ele estava antes de ser realiza uma chamada de uma subrotina;

- O registrador 15 (`r15`) é o contador de programa (*pc - program counter*), isto é, registrador que contém o endereço da próxima instrução a ser executada pelo processador.

Além desses 16 registradores, a arquitetura ARM contém mais dois registradores de status do programa: `cpsr` e `spsr`, do inglês *current program status register* e *saved program status register* que são utilizados para monitoramento e controle interno das operações.

Códigos de condição		
Flag	**Nome**	**Ocorrência**
V	o**V**erflow	o resultado gera um *overflow*
C	**C**arry	o resultado gera um *carry*
Z	**Z**ero	o resultado é zero
N	**N**egativo	bit 31 do resultado é 1

Tabela 5.1: Códigos de condição

Condicionais Mnemônicos		
Mnemônico	**Nome**	**Códigos de condição**
EQ	**EQ**ual	Z
NE	**N**ot **E**qual	z
CS HS	**C**arry **S**et / unsigned **H**igher or **S**ame	C
CC LO	**C**arry **C**lear / unsigned **LO**wer	c
MI	**MI**nus / negative	N
PL	**PL**us / positive or zero	n
VS	overflow	V
VC	no overflow	v
HI	unsigned **HI**gher	zC
LS	unsigned **LO**wer or same	Z ou c
GE	signed **G**reater than or **E**qual	NV ou nv
LT	signed **L**ess **T**han	Nv ou nV
GT	signed **G**reater **T**han	NzV ou nzv
LE	signed **L**ess than or **E**qual	Z ou Nv ou nV
AL	**AL**ways	incondicional

Tabela 5.2: Condicionais Mnemônicos

Por exemplo, códigos de condição ou *flags* atualizam o registrador `cpsr`. Essas *flags* são atualizadas ao realizar comparações ou operações aritméticas com instruções adicionadas do sufixo `S`. A instrução `ADDS` soma o valor de dois registradores e armazena o resultado em um registrador de destino. Se a operação resultar em um *carry* então o código de condição `C` do registrador `cpsr` recebe o valor 1.

Por meio dos códigos de condição são realizados os controles de execução condicional que determina se o processador irá executar ou não determinada instrução. A grande maioria das instruções ARM possui condicionais mnemônicos que determinam se esta será executada ou não baseando no conjunto de códigos de condição. Antes da execução da instrução é realizada a comparação de quais códigos de condição no registrador `cpsr` estão com o valor 1 ou 0. Se essa comparação for igual a condição pré-estabelecida pelo condicional mnemônico a instrução será executada pelo processador, caso contrário ela é apenas ignorada.

5.2.2 Chamadas de função

As chamadas de função na arquitetura ARM devem seguir o padrão (APCS - *ARM Procedure Call Standard*) que define como são passados os argumentos e o retorno de valores nos registradores durante uma chamada de função.

Em uma chamada de função ARM, os quatro primeiros argumentos são passados por meio dos registradores `r0, r1, r2, r3`. Se por acaso houver mais argumentos, eles são colocados na pilha seguindo a ordem em que foram passados. O valor de retorno da função é passado pelo registrador `r0`. Este descrição abrange os casos em que são passados valores inteiros ou endereços. No caso de argumentos de precisão dupla, como *long long* ou o *double* eles são passados em pares de registradores e tem o seu valor retornado nos registradores `r0, r1`.

Repare que do ponto de vista do desempenho, funções com quatro ou menos parâmetros são bem mais eficientes do que funções com mais de quatro parâmetros, uma vez que neste caso pode-se passar todos os argumentos já em registradores. No caso das funções com mais argumentos é necessario realizar operações de acesso à pilha que são bem custosas, já que envolvem acesso a memória.

É importante ressaltar também que, nas chamadas das funções os valores contido nos registradores `r4` até `r12` devem ser salvos antes de utilizarmos esse registradores dentro da função. E deve-se restaurá-los antes de finalizar a execução da função, pois caso contrário podemos sobrescrever valores que estão sendo utilizados em outra parte do programa.

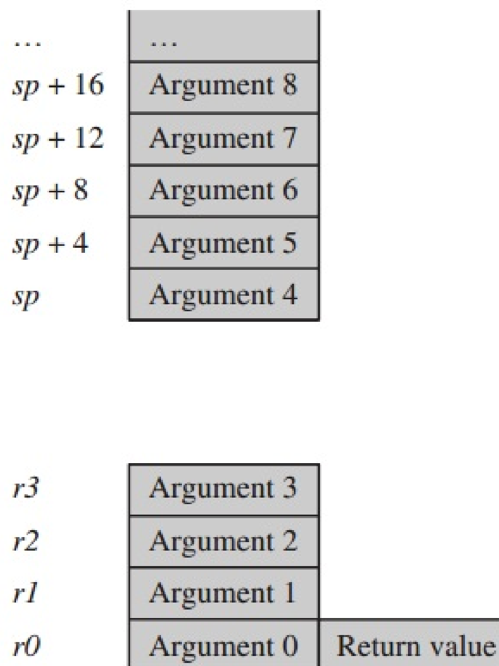

Figura 5.2: Chamada de Funções em ARM

Capítulo 6

Implementação

Neste trabalho, foi realizada a otimização e implementação de operações aritméticas em corpos primos utilizando *assembly* ARM. Utilizou-se como base a biblioteca RELIC (Aranha and Gouvêa). Como ambiente de programação, foi usado a IDE Eclipse com os plugin ADT (*Android Development Tools*) e Android SDK e NDK para permitir a criação de um ambiente para a construção de aplicações para o sistema Android. Para realizar as medições de tempo, foi utilizado o dispositivo Samsung Galaxy Note com a versão 4.0.3 do sistema operacional Android.

Utilizou-se como parâmetros a curva elíptica $y^2 = x^3 + 2$ definida sobre um corpo primo, com $|p| = 254$ bits sendo p representado por um vetor de palavras de 32 *bits* (`0x25236482 0x40000001 0xBA344D80 0x00000008 0x61210000 0x00000013 0xA7000000 0x00000013`) e ordem em torno de 252 bits (2).

6.1 Adição modular

Primeiramente, foi implementada a função para adição modular de dois inteiros multiprecisão. Essa função recebe dois inteiros a e b e retorna o inteiro c tal que $c = (a + b) \bmod p$ sendo p o primo. O algoritmo em C, pode ser visto no código (6.1) .

Código 6.1: Função para adição modular de inteiros multiprecisão em C

```
void fp_add_basic (fp_t c, fp_t a, fp_t b) {
        digt_t carry;

        carry = fp_addn_lowc(c, a, b);
        if (carry || (fp_cmp(c, fp_prime_get()) != CMP_LT)) {
                carry = fp_subn_low(c, c, fp_prime_get());
        }
}
```

Veja que para realizar a adição modular são necessárias na verdade três primitivas: a adição, a comparação e a subtração de inteiros multiprecisão. Por esse motivo foram implementadas em assembly ARM as três funções `fp_addn_low`, `fp_cmp` e `fp_subn_low`.

6.1.1 Adição

A implementação em C para a adição de dois números multiprecisão pode ser vista no código (6.2) . Note que na implementação na linguagem C, a verificação da existência de *carry* deve ser realizada por meio de uma operação de comparação. Esse fato ocorre devido à falta de suporte da linguagem a esse tipo de tratamento.

Código 6.2: Função para adição de inteiros multiprecisão em C

```c
dig_t fp_addn_low(dig_t *c, dig_t *a, dig_t *b) {
        int i;
        dig_t carry, c0, c1, r0, r1;

        carry = 0;
        for (i = 0; i < FP_DIGS; i++, a++, b++, c++) {
                r0 = (*a) + (*b);
                c0 = (r0 < (*a));
                r1 = r0 + carry;
                c1 = (r1 < r0);
                carry = c0 | c1;
                (*c) = r1;
        }
        return carry;
}
```

A implementação em assembly ARM foi realizada desenrolando o laço de repetição, uma vez que o número de iterações é previamente conhecido. Como estamos utilizando inteiros de 254 bits são necessárias oito iterações. Com isso, evitamos a utilização de instruções de desvio condicional e economizam-se registradores e instruções, pois não é preciso reservar um registrador para a variável de controle do laço nem realizar o incremento da mesma.

Código 6.3: Início da função de adição de inteiros multiprecisão em *assembly* ARM

```
fp_addn_low:
        STMDB sp!, {r4}

ADDN_STEP:
        /**** Primeira iteracao ****/
        LDR r3, [r1, #0]
        LDR r4, [r2, #0]
        ADDS r3, r3, r4
        STR r3, [r0, #0]

        /**** Segunda iteracao ****/
        LDR r3, [r1, #4]
        LDR r4, [r2, #4]
        ADCS r3, r3, r4
        STR r3, [r0, #4]
```

Quanto ao tratamento da *flag* de *carry*, em assembly ARM existem instruções específicas para realizar esta função. No bloco referente à primeira iteração do algoritmo, veja

que inicialmente é realizado o carregamento dos valores a serem adicionados em registradores com instruções do tipo *load*. Em seguida, na primeira iteração, é feita a adição dos valores utilizando a instrução ADD seguida do sufixo S. Esse sufixo atualiza alguns códigos de condição do registrador *cpsr* dependendo da instrução que a acompanha. No caso da instrução ADD são atribuídos os códigos de condição N, Z, C e V (5). Ao final do bloco, o resultado é armazenado em memória na sua devida posição com a instrução *store*.

Para a segunda iteração, os próximos valores a serem utilizados são carregados novamente em registradores e são adicionados com a instrução ADCS. Essa instrução adiciona os valores contidos nos registradores além do valor contido no código de condição de *carry*. Dessa forma, ao contrário da linguagem C, o tratamento do *carry* é realizado de maneira mais simplificada e inteligível. Além disso, utiliza-se o sufixo S para atualizar os códigos de condição novamente para capturar um eventual *carry*. No final da iteração, o resultado é armazenado na posição correspondente.

Código 6.4: Final da função de adição de inteiros multiprecisão em *assembly* ARM

```
/**** Setima iteracao ****/
LDR r3, [r1, #24]
LDR r4, [r2, #24]
ADCS r3, r3, r4
STR r3, [r0, #24]

/**** Oitava iteracao ****/
LDR r3, [r1, #28]
LDR r4, [r2, #28]
ADC r3, r3, r4
STR r3, [r0, #28]

MOV r0, #0

LDMIA sp!, {r4}
MOV pc, lr
```

As demais iterações, até a setima, são idênticas, diferindo somente a posição em que o resultado é armazenado na memória. Na última iteração, oitava, não é necessário a utilização do sufixo S, pois não será mais realizada nenhuma operação com o código de condição de *carry*. Como estão sendo utilizados inteiros de 254 bits, mesmo no caso em que a operação de adição resulte em um *carry*, um dos bits extras poderá armazená-lo. Dessa forma a função sempre retorna o valor 0 para o *carry*. É importante ressaltar também, que como foi utilizado o registrador r4, segundo o padrão APCS, este deve ter o seu valor salvo antes do ínicio da função e o seu valor restaurado antes do término da mesma.

6.1.2 Comparação

Em seguida, foi implementada a função de comparação de inteiros multiprecisão. Essa função recebe dois inteiros a e b e retorna uma *flag* que pode ser CMP_EQ se os inteiros forem iguais, CMP_GT caso a seja maior que b e CMP_LT caso contrário. O algoritmo em C, pode ser visto no código (6.5) .

Código 6.5: Função para comparação de inteiros multiprecisão em C

```c
int fp_cmpn_low(dig_t *a, dig_t *b) {
        int i, r;

        a += (FP_DIGS - 1);
        b += (FP_DIGS - 1);
        r = CMP_EQ;
        for (i = 0; i < FP_DIGS; i++, --a, --b) {
                if (*a != *b && r == CMP_EQ) {
                        r = (*a > *b ? CMP_GT : CMP_LT);
                }
        }
        return r;
}
```

O algoritmo funciona da seguinte maneira: os dois números inteiros a serem comparados são inicializados de tal forma que a comparação seja realizada dos *bits* mais significativos até os menos significativos. Consequentemente, caso a comparação parcial resulte em algum valor diferente da *flag* `CMP_EQ`, este resultado poderá ser associado à comparação como um todo. Dessa forma, na primeira iteração são comparados os dígitos da oitava posição do número, assumindo que os inteiros são representados como um vetor de palavras de 32 *bits*. Se a comparação tiver como resultado um valor diferente da *flag* `CMP_EQ` o algoritmo para e retorna a *flag* em questão como resultado da operação. Esse procedimento é repetido até a posição menos significativa, que corresponde à posição zero do vetor.

Código 6.6: Inicio da função de comparação de inteiros multiprecisão em *assembly* ARM

```
CMPN_STEP:
        /**** Primeira iteracao ****/
        LDR r2, [r0, #28]
        LDR r3, [r1, #28]
        CMP r2, r3      /* set condition "HI" if (r2 > r3),*/
                        /*             or "LO" if (r2 < r3)*/
        BHI GREATER_THAN
        BLO LESS_THAN

        /**** Segunda iteracao ****/
        LDR r2, [r0, #24]
        LDR r3, [r1, #24]
        CMP r2, r3      /* set condition "HI" if (r2 > r3),*/
                        /*             or "LO" if (r2 < r3)*/
        BHI GREATER_THAN
        BLO LESS_THAN
```

Levando em conta o desempenho, o melhor caso ocorre quando o dígito mais significativo dos inteiros é diferente (necessário apenas 1 iteração) e o pior caso ocorre quando os dois inteiros são iguais (necessário 8 iterações).

A implementação em *assembly* ARM foi realizada de maneira semelhante ao algoritmo na linguagem C. A comparação dos inteiros é realizada da mesma forma, iniciando dos

dígitos mais significativos até os menos significativos. Por esse motivo, a instrução de acesso à memória do tipo *load* tem na primeira iteração o endereço 28, correspondente à oitava posição do vetor de palavras que armazena o inteiro multiprecisão.

Para a comparação em *assembly* foi utilizado a instrução `CMP`. Esse instrução compara valores de 32 bits em registradores e atualiza os códigos de condição do registrador `cpsr` de acordo com o resultado. Em seguida, utilizou-se essa informação para alterar o fluxo do programa utilizando execuções condicionais. A instrução `CMP`, ao contrário das instruções `ADD` ou `ADC`, não necessita do sufixo `S` para atualizar as *flags*.

A alteração do fluxo do programa é realizado utilizando a instrução de desvio `B` *branch* combinada dos mnemônicos de condição `HI` (*unsigned Higher*) e `LO` (*unsigned Lower*). A instrução *branch* força a mudança no fluxo do programa fazendo com que o registrador de contador de programa (`pc`) aponte para o endereço passado como parâmetro na instrução. Na função implementada, a instrução *branch* sempre é utilizada após uma instrução `CMP`, pois com já foi dito, esta última instrução de desvio atualiza os códigos de condição do registrador `cpsr`. Logo, dependendo das *flags* setadas o programa pode ser desviado para o endereço do *label* `GREATER_THAN` ou `LESS_THAN` que coloca no registrador de retorno a *flag* correspondente. Caso não seja satisfeita nenhuma das condições (`HI` ou `LO`), então o programa não é desviado e segue para a próxima comparação. Se ao final das oito iterações não for realizado nenhum desvio no fluxo do programa, a *flag* de igualdade `CMP_EQ` é colocada no registrador de retorno.

Note que esta função utiliza apenas os registradores `r0`, `r1` para armazenar o endereço dos inteiros multiprecisão a serem comparados e os registradores `r2, r3` para armazenar o dígito da posição do vetor de cada inteiro em cada iteração. No final da função o resultado da comparação é colocado diretamente no registrador de retorno de função `r0`. Por esse motivo, não é necessário salvar/restaurar o contexto de nenhum registrador, pois não se espera que os registradores `r0, r1, r2` e `r3` sejam preservados durante uma chamada de procedimento.

Código 6.7: Final da função de comparação de inteiros multiprecisão em *assembly* ARM

```
        /**** Oitava iteracao ****/
        LDR r2, [r0, #0]        /* r2 = *a */
        LDR r3, [r1, #0]
        CMP r2, r3      /* set condition "HI" if (r2 > r3),*/
                        /*             or "LO" if (r2 < r3)*/
        BHI GREATER_THAN
        BLO LESS_THAN

EQUAL:
        MOV r0, #CMP_EQ
        MOV pc, lr

GREATER_THAN:
        MOV r0, #CMP_GT
        MOV pc, lr              /* return r*/

LESS_THAN:
        MOV r0, #CMP_LT
```

```
        MOV pc, lr              /* return r*/
```

6.1.3 Subtração

Fechando as três operações primitivas necessárias para a adição modular, foi implementada a subtração de inteiros multiprecisão. O algoritmo em C assemelha-se bastante com o da adição e pode ser visto no código (6.8). Igualmente a operação de adição, o tratamento da *flag* de *carry* teve de ser improvisado utilizando comparações e operações lógicas.

Código 6.8: Função para subtração de inteiros multiprecisão em C

```
dig_t fp_subn_low(dig_t *c, dig_t *a, dig_t *b) {
        int i;
        dig_t carry, r0, diff;

        // Zero the carry.
        carry = 0;
        for (i = 0; i < FP_DIGS; i++, a++, b++, c++) {
                diff = (*a) - (*b);
                r0 = diff - carry;
                carry = ((*a) < (*b)) || (carry && !diff);
                (*c) = r0;
        }
        return carry;
}
```

Código 6.9: Função para subtração de inteiros multiprecisão em *assembly* ARM

```
fp_subn_low:
        STMDB sp!, {r4}

        SUBN_STEP:
                /**** Primeira iteracao ****/
                LDR r3, [r1, #0]
                LDR r4, [r2, #0]
                SUBS r3, r3, r4
                STR r3, [r0, #0]

                /**** Segunda iteracao ****/
                LDR r3, [r1, #4]
                LDR r4, [r2, #4]
                SBCS r3, r3, r4
                STR r3, [r0, #4]

                        .
                        .
                        .

                /**** Oitava iteracao ****/
                LDR r3, [r1, #28]
                LDR r4, [r2, #28]
```

```
SBCS r3, r3, r4
STR r3, [r0, #28]

MOV r0, #0
SBCS r0, r0, r0

LDMIA sp!, {r4}
MOV pc, lr              /* return carry*/
```

A implementação em *assembly* (6.9) foi realizada novamente desenrolando o laço de repetição para obter as vantagem já mencionadas. O tratamento da flag de *carry*, ou mais precisamente do *borrow*, é realizado da mesma maneira que o algoritmo da adição. No caso da subtração, foi utilizada a instrução `SUB` seguida do pós-fixo `S` na primeira iteração e nas demais a instrução `SBCS`. A instrução `SBC` subtrai o valor contido nos registradores além da *flag* de *carry* e o pós-fixo `S` é utilizado para atualizar os códigos de condição a fim de capturar um possível *carry*. A diferença da subtração em relação a da adição é que na última iteração ainda é necessário a atualização das *flags*, pois mesmo utilizando inteiros de 254 bits, pode ser o caso do primeiro inteiro multiprecisão ser menor que o segundo inteiro. Então a operação de subtração gera um *carry* que deve ser retornado pelo procedimento.

6.1.4 Adição modular com instruções de desvio

Finalizada as três operações aritméticas necessárias para a construção do algoritmo de adição modular foi realizada uma implementação com o uso de instruções de desvio (*branch*). O primeiro passo do algoritmo foi a utilização da adição de inteiros multiprecisão conforme discutido na seção (6.1.1).

Terminada esse operação, o segundo passo é a comparação do resultado da adição com o primo. O primo p é uma constante que é representado como um vetor de palavras de 32 bits. Para esse passo, foi utilizado um algoritmo semelhante ao descrito na seção (6.1.2), porém com algumas otimizações. Como o módulo primo já é previamente conhecido, foram utilizadas macros para definir as constantes de cada posição do vetor contendo o primo. Dessa forma, antes de realizar a comparação, basta colocar a constante em questão em registrador por meio da instrução `MOV`. Porém, em alguns casos (formato não esparso), não foi possível armazenar o valor em registrador diretamente com a instrução `MOV`. Foi necessário realizar a carga em dois passos, utilizando as instruções `MOVT` (*move top*) e `MOVW` (*move wide*). A instrução `MOVT` escreve na metade mais alta do registrador o valor do imediato e `MOVW` na metade mais baixa.

Veja que uma das diferenças dessa implementação em relação a da seção (6.1.2) é o armazenamento em registrador de um dos parâmetros da comparação que é realizado utilizando uma (ou duas) instruções do tipo *move* ao invés de utilizar a instrução de acesso à memória do tipo *load*. Esse registrador guarda o valor referente ao primo. O outro parâmetro, o resultado da adição, é armazenado nos registradores r4 até r12, sendo que este valor foi guardado nestes locais no passo anterior (adição de inteiros) pela instrução de adição (`ADD` ou `ADC`). Logo, antes da instrução de comparação não é necessário a utilização de instruções do tipo *load* como no algoritmo (6.6), pois um dos operandos é originado pelo instrução `ADD` e o outro pela `MOV`.

Outra diferença é em relação aos endereços. No caso da adição modular é preciso saber se o resultado é maior ou igual ao primo (`GREATER_THAN_OR_EQUAL`) ou não (`LESS_THAN`). Se essa primeira condição for verdadeira, então deve-se subtrair o primo. Por esse motivo, após cada iteração da comparação é realizada um teste de execução condicional utilizando os mnemônicos `HI` (*Higher*) ou `LO` (*Lower*). Se o resultado for maior, então o fluxo do programa será desviado para o endereço `GREATER_THAN_OR_EQUAL` no qual é realizada a subtração do primo p. Se o resultado for menor, então será desviado para o endereço `LESS_THAN` que escreve o resultado em memória e retorna o controle para o procedimento chamador. Se o resultado for igual, então o programa não terá o seu fluxo desviado, porém ao término das comparações ele entra no espaço relativo ao endereço `GREATER_THAN_OR_EQUAL` realizando a subtração conforme esperado (Código 6.10).

Código 6.10: Trecho da função de adição de inteiros multiprecisão em ARM com *branch*

```
/**** Primeira iteracao ****/
MOVW r3, #0x6482
MOVT r3, #0x2523
CMP r12, r3
BHI GREATER_THAN_OR_EQUAL
BLO LESS_THAN

/**** Segunda iteracao ****/
CMP r11, #0x40000001
BHI GREATER_THAN_OR_EQUAL
BLO LESS_THAN

/**** Terceira iteracao ****/
MOVW r3, #0x4D80
MOVT r3, #0xBA34
CMP r10, r3
BHI GREATER_THAN_OR_EQUAL
BLO LESS_THAN
```

Finalmente, no caso do programa ser desviado ou chegar ao endereço `GREATER_THAN_OR_EQUAL` é realizada a subtração semelhante a realizada na seção (6.1.3). No entanto, como é preciso subtrair o valor do primo p, ele é colocado em registrador novamente utilizando as instruções `MOV` ou `MOVT` e `MOVW`. A subtração é realizada em iterações utilizando as instruções `SUBS` e `SBCS`. Sendo que ao final da operação de subtração de cada iteração, o resultado já pode ser armazenado em memória utilizando a instrução *store*.

6.1.5 Adição modular sem instruções de desvio

Foi realizada uma outra implementação sem a utilização de instruções de desvio condicional. Sabe-se que esse tipo de instrução afeta drasticamente o desempenho de processadores *pipeline*, devido às dependências de controle. E os desvios condicionais são as instruções que melhor representam este efeito negativo. Por esse motivo são, empregadas algumas técnicas de previsão de desvios, que ajudam a diminuir o efeito, pois pode-se saber antecipadamente qual das ramificações no fluxo do programa será selecionada cada vez que uma instrução de desvio condicional for executada.

Para tentar minimizar esses efeitos foi realizada uma outra implementação. Primeiramente, é realizado a adição dos inteiros multiprecisão, conforme já explicado na seção anterior. Porém, em seguida, ao invés de se realizar uma comparação, é realizada a subtração do primo p. Essa subtração é feita utilizando-se o mesmo algoritmo descrito na seção anterior. Após a subtração é verificado se a operação resultou em um *carry*. Se houver *carry*, então o resultado ficou negativo, devendo-se adicionar o primo p novamente.

Código 6.11: Trecho da função de adição de inteiros multiprecisão em ARM sem *branch*

```
MOV r3, #0              /* r3 = carry = 0 */
SBCS r3, r3, r3   /* Armazenando o resultado do carry no r3*/

/**** Primeira iteracao ****/
AND  r4, r3, #0x13
ADDS r5, r5, r4

/**** Segunda iteracao ****/
AND  r4, r3, #0xA7000000
ADCS r6, r6, r4

/**** Terceira iteracao ****/
AND  r4, r3, #0x13
ADCS r7, r7, r4
```

Essa adição é realizada utilizando as instruções `ADDS` ou `ADCS` conforme a iteração. No entanto, o segredo para não ser necessário a utilização de instruções de desvio é o teste realizado na instrução anterior ao `ADDS` ou `ADCS`. Antes da operação de adição é utilizado a instrução `AND` para realizar uma operação lógica com o registrador que armazena o *carry* da subtração e o registrador que contém a posição do vetor que armazena o primo p correspondente a iteração. Dessa forma, se não houver *carry*, o resultado do `AND` será zero. Então, ao realizar a adição o resultado não será alterado, conforme o esperado (ver código 6.11). Porém, se a subtração gerar um *carry* ao realizar a operação lógica `AND` o resultado será o primo correspondente a iteração. Logo, caso haja *carry* deve-se adicionar o primo p novamente. Por fim, o resultado é armazenado em memória com instruções *store*.

6.2 Subtração modular

Foi implementada também a função para subtração modular de dois inteiros multiprecisão. Essa função recebe dois inteiros a e b e retorna o inteiro c tal que $c = (a - b) \bmod p$ sendo p o primo. O algoritmo em C pode ser visto no código (6.12) . No caso da subtração modular é necessário a utilização somente das operações de subtração e adição de inteiros multiprecisão.

Código 6.12: Função para subtração modular de inteiros multiprecisão em C

```
void fp_sub_basic(fp_t c, fp_t a, fp_t b) {
        dig_t carry;
```

```
		carry = fp_subn_low(c, a, b);
		if (carry) {
			fp_addn_low(c, c, fp_prime_get());
		}
	}
```

Assim como para a adição modular, para a subtração modular também foram realizadas duas implementações. Uma utilizando instruções de desvio do tipo *branch* (6.2.1) e outra sem a utilização dessas instruções (6.2.2).

6.2.1 Subtração modular com instruções de desvio

A versão da subtração modular com instruções de desvio tem como primeiro passo o algoritmo da subtração de inteiros multiprecisão já apresentado na seção (6.1.3). Ao término da subtração é verificado se a operação em questão resultou em uma *flag* de *carry*.

Essa operação é realizada pela instrução `CMP` que também atualiza os códigos de condição do registrador *cpsr*. É feita a comparação do valor armazenado na *flag* de *carry* com o inteiro zero. Se houver *carry*, ou seja, quando a instrução `CMP` levantar o código de condição `Z`, o fluxo do programa é desviado para o endereço `IS_ONE` que realiza a adição do primo p. Caso contrário, o fluxo programa é desviado para o endereço `IS_ZERO` que armazena o resultado em memória com instruções do tipo *store*. O desvio de fluxo é realizado pela instrução de desvio condicional *branch*. Após a verificação da existência ou não de *carry* é utilizada a instrução *branch* seguida dos mnemônicos de condição `EQ` (*equal*) e `NE` (*not equal*) para realizar os devidos desvios no fluxo do programa.

6.2.2 Subtração modular sem instruções de desvio

A versão da subtração modular sem instruções de desvio tem como primeiro passo, assim como o algoritmo com instruções de desvio, a subtração de inteiros multiprecisão apresentado na seção (6.1.3). Ao término da subtração é verificado novamente se esta resultou em uma *flag* de *carry*. Porém, não é realizado uma comparação como descrito na seção anterior (6.2.1). É utilizado a operação lógica `AND` para determinar o valor a ser adicionado conforme já explicado no algoritmo de adição modular sem instruções de desvio (6.1.5). Finalmente, como útlimo passo tem-se o armazenamento do resultado em memória com instruções *store*.

6.3 Multiplicação

Em seguida, foram realizadas implementações para a multiplicação multiprecisão. Por questões de desempenho, não foi feita a implementação do algoritmo *Schoolbook*, sendo realizada somente a da multiplicação Comba (6.3.1) e da versão híbrida (6.3.2).

6.3.1 Comba

O algoritmo para a muliplicação em C do método Comba pode ser visualizada na figura (6.13). Conforme explicado anteriormente, este algoritmo realiza o processamento de cada

produto parcial seguindo uma orientação a colunas. Dessa forma, todas as operações de multiplicação e adição relativas a um determinada coluna são realizadas consecutivamente sendo que ao término desse processamento uma parte do resultado da multiplicação deverá ser obtido.

No algoritmo em C foi utilizado as variáveis `r0`, `r1` e `r2` para acumular o resultado do processamento parcial. Além disso, a operação de multiplicação foi divida em duas partes, correspondentes ao dois laços do programa. O primeiro laço calcula as primeiras oito posições do resultado da multiplicação, admitindo novamente que o resultado seja representado por um vetor de palavras de 32 bits. E o segundo laço as próximas oito posições. Lembrando que a multiplicação resulta em um inteiro de dupla precisão.

Código 6.13: Função para multiplicação Comba de inteiros multiprecisão em C

```c
void fp_muln_low(dig_t *c, dig_t *a, dig_t *b) {
        int i, j;
        dig_t *tmpa, *tmpb, r0, r1, r2;

        r0 = r1 = r2 = 0;
        for (i = 0; i < FP_DIGS; i++, c++) {
                tmpa = a;
                tmpb = b + i;
                for (j = 0; j <= i; j++, tmpa++, tmpb--) {
                        COMBA_STEP(r2, r1, r0, *tmpa, *tmpb);
                }
                *c = r0;
                r0 = r1; r1 = r2; r2 = 0;
        }
        for (i = 0; i < FP_DIGS; i++, c++) {
                tmpa = a + i + 1;
                tmpb = b + (FP_DIGS - 1);
                for (j = 0; j < FP_DIGS - (i + 1); j++, tmpa++, tmpb--) {
                        COMBA_STEP(r2, r1, r0, *tmpa, *tmpb);
                }
                *c = r0;
                r0 = r1; r1 = r2; r2 = 0;
        }
}
```

A implementação em assembly ARM foi realizada desenrolando o laço. Para facilitar a construção e a legibilidade do código foram utilizadas duas macros: `COMBA_STEP` e `COMBA_MFIN` (6.14).

A macro `COMBA_STEP` primeiramente carrega em registradores os operandos da multiplicação com a instrução de acesso à memória do tipo *load*. Em seguida, é utilizada a instrução `UMLAL` (*unsigned multiply accumulate long*) (5). Esta instrução recebe quatro parâmetros, os dois primeiros armazenam a parte alta e a parte baixa do resultado. Os outros dois parâmetros referem-se aos valores a serem multiplicados. Note que esta instrução primeiro realiza a multiplicação dos dois últimos parâmetros e em seguida adiciona (acumula) o resultado nos dois primeiros parâmetros, sendo o resultado final armazenado nesses par de registradores.

No entanto, a instrução `UMLAL` não captura uma possível *flag* de *carry* que poderia ser resultado da acumulação. Dessa forma, deve-se obter essa possível *flag* de *carry* de forma

manual. A janela de registradores utilizadas para acumular os resultados parciais no código em *assembly* corresponde aos registradores r3, r4 e r5. No entanto, são passados como parâmetro para conter o resultado final do UMLAL o registrador r5 e o registrador r12. Este último pode ser entendido como um registrador auxiliar para a captura do código de condição de *carry*. Para obter a *flag* de *carry*, antes da instrução UMLAL, o registrador auxiliar tem o seu valor zerado. E em seguida é utilizada a instrução UMLAL tendo o resultado dessa operação, parte alta e parte baixa, armazenado nos registradores r5 e r12 respectivamente. Repare que como o registrador r12 foi zerado no passo anterior, ele armazena somente a parte baixa da multiplicação. Não foi realizada ainda a operação de acumulação. Esse acumulação é realizada manualmente utilizando a instrução ADDS. Utiliza-se a instrução ADDS tendo como operando o conteúdo do registrador r12 e r4. Como foi utilizado o pós-fixo S uma possível *flag* de *carry* será capturada nessa instrução. Finalmente, esse possível *flag* é adicionada ao último registrador da janela de acumuladores com a instrução ADC, finalizando o tratamento para captura manual da *flag* de *carry*.

A macro COMBA_MFIN é sempre chamada após a computação de todas operações de todos os operando envolvidos no processamento de determinada coluna. Nessa macro, ocorre simplesmente o armazenamento em memória da posição do produto cálculada na respectiva iteração e o giro dos registradores de acumulação, preparando-os para a próxima iteração do algoritmo.

Código 6.14: Macros para multiplicação Comba de inteiros multiprecisão em ARM

```
.macro COMBA_STEP i j
        LDR r6, [r1, #(4*\i)]
        LDR r7, [r2, #(4*\j)]
        MOV r12, #0
        UMLAL r5, r12, r6, r7
        ADDS r4, r4, r12
        ADC r3, r3, #0
.endm

.macro COMBA_MFIN i
        STR r5, [r0, #(4*\i)]
        MOV r5, r4
        MOV r4, r3
        MOV r3, #0
.endm
```

6.3.2 Híbrido

Foi realizada também uma implementação em assembly ARM do método híbrido. Assim como no método Comba, foram criadas macros para facilitar o entendimento do código. As macros utilizadas nesse caso foram: H2_STEP e H2_MFIN (6.15). Nesta implementação, utilizou-se 2x2 multiplicações dentro do bloco.

A macro H2_STEP realiza as 2x2 multiplicações dentro do bloco do híbrido. Lembre-se que essa multiplicação é realizada segundo o algoritmo *Schoolbook*. Utiliza-se a instrução UMLAL para realizar a multiplicação e a acumulação dos possíveis *carries*, sendo necessário

utilizar instruções ADDS e ADC para capturar um possível *carry* que possa resultar dessa operação, conforme já descrito na seção anterior.

Porém, nessa implementação, utiliza-se apenas um registrador (ao invés dos três tradicionais) para servir como *carry-catcher* (29). Um registrador utilizado como *carry-catcher* serve para capturar e acumular a soma dos possíveis *bits* de *carry*. No entanto, eles não precisam de um registrador de 32 bits para isso, um *byte* já é o suficiente. Dessa forma, na implementação realizada, os tradicionais três *carry-catchers* são compactados em apenas um registrador. O segredo dessa implementação é utilizar a instrução ADDCS (*ADD with Carry Set*), que somente executa uma instrução ADD se a *flag* de *carry* estiver setada, para adicionar o *carry* em uma posição específica do registrador utilizado como *carry-catcher*.

A macro H2_MFIN é utilizada para realizar o processamento após o alinhamento dos blocos. Como o registrador utilizado como *carry-catcher* foi compactado, o seu conteúdo está mascarado, precisando dessa forma de um pré-processamento antes de poder ser adicionado aos registradores de acumulação. Esse pré-processamento é realizado com a instrução AND que seleciona uma parte específica do *carry-catcher* e do deslocamento antes da adição que no processador ARM pode ser realizado sem um custo adicional.

Código 6.15: Macros para multiplicação híbrida de inteiros multiprecisão em ARM

```
.macro H2_STEP i j
        LDR r8,[r1,#(4 * \i)]
        LDR r7,[r2,#(4 * \j)]
        LDR r9,[r1,#(4 * \i+4)]
        MOV r11, #0
        UMLAL r3,r11,r8,r7
        MOV r10, #0
        ADDS r4, r4, r11
        ADC r12,r12,#0
        UMLAL r4,r10,r9,r7
        MOV r11, #0
        ADDS r5, r5, r10
        ADDCS r12,r12,#0x100
        LDR r7,[r2,#(4 * \j+4)]
        UMLAL r4,r11,r8,r7
        MOV r10, #0
        ADDS r5, r5, r11
        ADDCS r12,r12,#0x100
        UMLAL r5,r10,r9,r7
        ADDS r6, r6, r10
        ADDCS r12,r12,#0x10000
.endm

.macro H2_MFIN i j
        STR r3,[r0,#(4 * \i)]
        STR r4,[r0,#(4 * \j)]
        AND r10,r12,#0xFF
        ADDS r3,r5,r10
        AND r10,r12,#0xFF00
        ADCS r4,r6,r10,LSR#8
        ADDCS r12,r12,#0x10000
        MOV r5,r12,LSR#16
        MOV r6,#0
        MOV r12,#0
```

```
.endm
```

A partir dessa implementação, foi realizada uma implementação alternativa. Foi possível liberar o registrador `r14`, dessa forma pode-se utilizar três registradores como *carry-catcher*. As macros utilizadas na múltiplicação híbrida com três *carry-catchers* podem ser observada no código (6.16) .

Código 6.16: Macros para multiplicação híbrida de inteiros multiprecisão em ARM com 3 *carry-catchers*

```
.macro H1_STEP i j
        LDR r8,[r1,#(4 * \i)]
        LDR r7,[r2,#(4 * \j)]
        LDR r9,[r1,#(4 * \i+4)]
        MOV r11, #0
        UMLAL r3,r11,r8,r7
        ADDS r4, r4, r11
        ADC r12,r12,#0
        MOV r11, #0
        UMLAL r4,r11,r9,r7
        ADDS r5, r5, r11
        ADC r10, r10, #0
        LDR r7,[r2,#(4 * \j+4)]
        MOV r11, #0
        UMLAL r4,r11,r8,r7
        ADDS r5, r5, r11
        ADC r10, r10, #0
        MOV r11, #0
        UMLAL r5,r11,r9,r7
        ADDS r6, r6, r11
        ADC r14, r14, #0
.endm

.macro H1_MFIN i j
        STR r3,[r0,#(4 * \i)]
        STR r4,[r0,#(4 * \j)]
        ADDS r3,r5,r12
        ADCS r4,r6,r10
        ADC r5,r14,#0
        MOV r6,#0
        MOV r10,#0
        MOV r12,#0
        MOV r14,#0
.endm
```

6.3.3 Karatsuba

Foi realizada uma implementação do algoritmo Karatsuba. O algoritmo utiliza a técnica de "dividir para conquistar", reduzindo o problema em instâncias menores. Como já descrito anteriormente, o algoritmo calcula a múltiplicação de inteiros utilizando multiplicações e algumas operações de adição e subtração de precisão reduzida.

Como foi implementado o algoritmo Karatsuba em apenas um nível, foi necessário utilizar apenas as operações de adição, subtração e multiplicação de inteiros de 128 *bits*. Para isso foi desenvolvido algoritmos semelhantes aos apresentados nas seções (6.1.1, 6.1.3, 6.3.1) ajustados para trabalhar com metade da precisão.

No entanto, o custo de processamento dos passos da "divisão e conquista"acabou penalizando o desempenho do algoritmo de forma considerável.

6.4 Redução modular

6.4.1 Montgomery

Foi implementada também a redução modular utilizando o algoritmo de Montgomery que substitui a operação de divisão por operações de deslocamento. A implementação da redução modular em C pode ser vista na figura (6.17).

Código 6.17: Função para redução modular de inteiros multiprecisão em C

```
void fp_rdcn_low(dig_t *c, dig_t *a) {
	int i, j;
	dig_t r0, r1, r2, u;
	dig_t *m, *tmp, *tmpm, *tmpc;

	u = *(fp_prime_get_rdc());
	m = fp_prime_get();
	tmpc = c;

	r0 = r1 = r2 = 0;
	for (i = 0; i < FP_DIGS; i++, tmpc++, a++) {
		tmp = c;
		tmpm = m + i;
		for (j = 0; j < i; j++, tmp++, tmpm--) {
			COMBA_STEP(r2, r1, r0, *tmp, *tmpm);
		}
		COMBA_ADD(r2, r1, r0, *a);
		*tmpc = (dig_t)(r0 * u);
		COMBA_STEP(r2, r1, r0, *tmpc, *m);
		r0 = r1;
		r1 = r2;
		r2 = 0;
	}

	for (i = FP_DIGS; i < 2 * FP_DIGS - 1; i++, a++) {
		tmp = c + (i - FP_DIGS + 1);
		tmpm = m + FP_DIGS - 1;
		for (j = i - FP_DIGS + 1; j < FP_DIGS; j++, tmp++, tmpm--) {
			COMBA_STEP(r2, r1, r0, *tmp, *tmpm);
		}
		COMBA_ADD(r2, r1, r0, *a);
		c[i - FP_DIGS] = r0;
		r0 = r1;
		r1 = r2;
		r2 = 0;
	}
```

```
        COMBA_ADD(r2, r1, r0, *a);
        c[FP_DIGS - 1] = r0;

        if (r1 || fp_cmpn_low(c, m) != CMP_LT) {
                fp_subn_low(c, c, m);
        }
}
```

A implementação em *assembly* ARM da redução de Montgomery tem por base o algoritmo da multiplicação Comba descrita na seção (6.3.1). Nesta implementação, foram adicionadas as macros `COMBA_STEP` que nada mais é do que a multiplicação de dois inteiros de 32 *bits*, tendo o seu resultado somado à janela de registradores de acumulação do Comba. E o `COMBA_ADD` que é a soma de um inteiro de 32 *bits* à janela de registradores.

O processamento principal desse algoritmo ocorre com a parte baixa do resultado, ou seja, nas oito primeiras iterações do algoritmo. A multiplicação da parte baixa do resultado com o vetor de 32 *bits* que armazena o primo p é feita utilizando a multiplicação Comba. A macro `COMBA_MFIN_LO` que corresponde à computação após as multiplicações de determinada coluna é composta de uma operação `COMBA_ADD`, seguida de uma multiplicação com a constante de redução e uma multiplicação com a instrução `UMLAL` para multiplicar o inteiro a ser reduzido com a primeira posição do vetor que armazena o primo p. No final da macro `COMBA_MFIN_LO` é realizado o giro dos registradores de coluna necessários na multiplicação Comba.

O passo final o algoritmo compara se o resultado da redução modular é menor que o primo p utilizando o algoritmo descrito na seção (6.1.2). Se o resultado ainda for maior subtrai-se o primo p do resultado como o algoritmo da seção (6.1.3).

Código 6.18: Macros para redução modular de inteiros multiprecisão em ARM

```
.macro COMBA_STEP i j
        LDR r6, [r0, #(4*\i)]
        MOV r12, #0
        MOV r7, #\j
        UMLAL r5, r12, r6, r7
        ADDS r4, r4, r12
        ADC r3, r3, #0
.endm

.macro COMBA_ADD i
        LDR  r6, [r1, #(4*\i)]
        ADDS r5, r5, r6
        ADCS r4, r4, #0
        ADC  r3, r3, #0
.endm

.macro COMBA_MFIN_LO i
        COMBA_ADD \i

        UMULL r6, r10, r5, r9
        STR r6, [r0, #(4*\i)]
```

```
        MOV r12, #0
        UMLAL r5, r12, r6, r8
        ADDS r4, r4, r12
        ADC r3, r3, #0

        MOV r5, r4
        MOV r4, r3
        MOV r3, #0
.endm

.macro COMBA_MFIN_HI i
        COMBA_ADD \i
        STR r5, [r0, #(4*(\i-8))]
        MOV r5, r4
        MOV r4, r3
        MOV r3, #0
.endm
```

6.4.2 Redução modular preguiçosa

A redução modular preguiçosa tem como ideia principal reduzir o número de operações de redução modular. Para isso, substitui-se as ocorrências de $((ab) \mod p + (cd) \mod p))$ no cálculo do emparelhamento por $((ab + cd) \mod p))$, conforme proposto em (2).

Do ponto de vista de desempenho, isso equivale a substituir reduções modulares por adições e subtrações de precisão dupla. Como a redução modular de Montgomery tem complexidade de ordem quadrática e a adição tem complexidade de ordem linear o ganho de desempenho é considerável. Para implementar essas operações de adição e subtração foram utilizados os algoritmos das seções (6.1.3 e 6.1.3) ajustando-os para trabalhar em precisão dupla.

Capítulo 7

Resultados obtidos

Para a tomada de tempo foi utilizado como padrão o número de instruções (em milhões de ciclos) necessárias para execução do cálculo do emparelhamento bilinear sobre curvas elípticas. Para verificar a correção dos algoritmos implementados foi utilizado um módulo específico de testes de corretude da biblioteca RELIC. Neste capítulo serão mostrados os resultados obtidos (7.1) e depois será realizada uma comparação com alguns trabalhos relacionados (7.2) seguida de uma breve conclusão (7.3) .

7.1 Resultados

Inicialmente foi medido o tempo de execução do cálculo do emparelhamento na versão básica da RELIC sem nenhuma aceleração em *assembly*. Em seguida foram realizadas as medições incluindo uma aceleração por vez. Nos casos em que foram realizadas mais de uma implementação para determinada operação foi escolhida a que resultou na melhor otimização. A versão base da biblioteca executou o emparelhamento bilinear em 28.1 milhões de ciclos.

A primeira tomada de tempo foi realizada adicionando a implementação da comparação descrita na seção (6.1.2). Somente com essa otimização, a execução do emparelhamento foi reduzida para 27.8 milhões de ciclos.

Em seguida, foi adicionada a aceleração com as adições e subtrações sem desvios condicionais referentes às seções (6.1.5 e 6.2.2). Essa alteração reduziu a execução do emparelhamento para 25.8 milhões de ciclos. Como também foi realizada uma implementação com desvios condicionais, em um segundo momento foram utilizados esses algoritmos descritos nas seções (6.1.4 e 6.2.1) para medir o tempo do cálculo do emparelhamento. Obteve-se como resultado a redução do tempo para 25.7 milhões de ciclos. Embora as instruções de desvio condicional afetem drasticamente o desempenho dos processadores *pipeline*, nos resultados obtidos a implementação com desvios foi ligeiramente superior à implementação sem desvios. Uma possível explicação para esse fato é que a implementação sem desvios possui mais instruções que a implementação com desvios e, embora tenham sido utilizados condicionais mnemônicos (tabela 5.2) para o controle de fluxo do programa, na implementação com desvios, mesmo que a instrução não deva ser executada é necessário pagar o seu custo de execução. Por obter uma otimização ligeiramente superior, adotou-se a implementação da adição/subtração com desvio condicional.

Otimização	Tempo*	%
1. Versão básica, sem aceleração em Assembly	28.1	0
2. Aceleração da comparação	27.8	1.1
3. Aceleração com adições/subtrações sem desvio condicional	25.8	8.2
4. Aceleração com adições/subtrações com desvio condicional	25.7	8.5
5. Aceleração com adições/subtrações de precisão dupla	23.0	18.2
6. Aceleração com multiplicador Comba	16.3	42.0
7. Aceleração com multiplicador híbrido e 1 *carry catcher*	15.8	43.8
8. Aceleração com multiplicador híbrido e 3 *carry catchers*	15.7	44.1
9. Aceleração com redução modular	12.3	56.2

Tabela 7.1: Resultados Obtidos em milhões de ciclos

Então, foi adicionada a aceleração da adição/subtração de precisão dupla que também é utilizada para otimizar a redução modular preguiçosa. Esta implementação é semelhante à apresentada na seção (6.1.1), realizando apenas os devidos ajustes para trabalhar em precisão dupla. Com essa aceleração, o tempo do emparelhamento foi reduzido para 23.0 milhões de ciclos. Somente com as implementações nas operações de adição e subtração, foi obtido uma redução de cerca de 18.2% no tempo do emparelhamento em relação à versão base.

Com a adição da aceleração do multiplicador Comba, explicado na seção (6.3.1), reduziu-se o tempo do emparelhamento para 16.3 milhões de ciclos. No entanto, foram realizadas outras implementações da multiplicação de inteiros múltiprecisão utilizando o método híbrido (6.3.2). Na implementação com um *carry-catcher*, reduziu-se o tempo para 15.8 milhões de ciclos, enquanto a implementação com três *carry-catchers* foi ligeiramente superior, reduzindo o tempo para 15.7 milhões de ciclos. Adotando a melhor implementação (método híbrido com três *carry-catchers*), obteve-se uma redução no tempo do emparelhamento em torno de 31.7% com a aceleração da multiplicação.

Finalmente, foi adicionada a aceleração da redução modular explicada na seção (6.4). Com essa otimização o tempo do emparelhamento foi reduzido para 12.3 milhões de ciclos. Com a redução modular, obteve-se uma melhoria de cerca de 21.7% no tempo do cálculo do emparelhamento.

Levando em conta todas as acelerações, o tempo do emparelhamento foi reduzido de 28.1 milhões de ciclos para 12.3 milhões de ciclos, o que corresponde a uma melhoria de 56.2% em relação à primeira versão sem nenhuma aceleração em *assembly*.

7.2 Comparação com trabalhos relacionados

Foi realizada também a comparação dos resultados obtidos neste trabalho com trabalhos relacionados (tabela 7.2). Para uma comparação justa, foi executado o mesmo emparelhamento bilinear em todas as implementações. Comparando o resultado com a implementação em C de Sanchez et al.(26) obteve-se uma redução em torno de 9.5% do tempo de execução. A implementação dos mesmo autores em ARM é mais eficiente que a do presente trabalho, no entanto possui um nível de complexidade maior por utilizar instruções vetoriais.

Trabalhos	Tempo*	%
Implementação (26) em Galaxy Note (ARM v7) Exynos 4 Cortex-A9 a 1.4 GHz (ASM com instruções vetoriais NEON)	9.5	-22.8
Implementação (14) em Galaxy Nexus (ARM v7) TI OMAP 4460 Cortex-A9 a 1.2 GHz (ASM)	11.9	-3.3
Este trabalho em Galaxy Note (ARM v7) Cortex-A9 a 1.4 GHz	12.3	0
Implementação (14) em Galaxy Nexus (ARM v7) TI OMAP 4460 Cortex-A9 a 1.2 GHz (C)	13.6	9.6
Implementação (26) em Galaxy Note (ARM v7) Exynos 4 Cortex-A9 a 1.4 GHz (C)	13.6	9.6
Implementação (1) em NVidia Tegra 2 (ARM v7) Cortex-A9 a 1.0 GHz (C)	51.0	75.9

Tabela 7.2: Comparação com trabalhos relacionados - Tempo em milhões de ciclos

Em relação à implementação em C de Grewal et al. (14), obteve-se uma melhoria de cerca de 8,9%. Comparando agora com a implementação em *assembly* dos mesmos autores, houve uma pequena diferença, sendo a implementação de Grewal et al. cerca de 3,3% mais rápida.

Já em relação aos trabalhos de Acar et al. (1) houve uma melhoria de cerca de 75.9% no tempo do cálculo do emparelhamento.

7.3 Conclusão

Neste trabalho foram apresentadas diversas implementações de operações aritméticas em corpos primos com o objetivo de providenciar a construção eficiente de sistemas criptográficos sobre emparelhamentos bilineares baseados em curvas elípticas em processadores ARM. Foram propostas várias implementações, sendo adotadas as que apresentaram as melhores reduções no tempo do cálculo do emparelhamento.

Com base na primeira implementação em C, foi possível verificar, após a adição das acelerações em *assembly* ARM, uma redução de cerca de 56% no tempo de execução do emparelhamento. Comparando com outras implementações relacionadas em C, a presente implementação apresentou uma melhoria em torno de 9.6%. Sendo cerca de 76% mais rápida que a implementação de Acar et al (1).

Como trabalhos futuros, sugere-se continuar os esforços de otimização em busca de superar o desempenho de todos os trabalhos relacionados.

Referências

[1] Acar, T., Lauter, K., Naehrig, M., and Shumow, D. (2013). Affine pairings on arm. In Abdalla, M. and Lange, T., editors, *PAIRING 2012*, volume 7708 of *Lecture Notes in Computer Science*, pages 203–209. Springer. 63

[2] Aranha, D. F. (2011). *Implementação eficiente em software de criptografia de curvas elípticas e emparelhamentos bilineares*. PhD thesis, Instituto de Computação, Unicamp, Campinas, SP, Brazil. vii, 16, 19, 44, 60

[Aranha and Gouvêa] Aranha, D. F. and Gouvêa, C. P. L. RELIC is an Efficient LIbrary for Cryptography. http://eprint.iacr.org. 44

[4] Aranha, D. F., Oliveira, L. B., López, J., and Dahab, R. (2010). Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187. 3

[5] ARM Limited (2003). ARM Architecture Reference Manual. ARM Doc No. DDI-0100. 46, 54

[6] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2008). On the indifferentiability of the sponge construction. *EUROCRYPT 2008*, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques:181–197. 9

[7] Brent, R. P. (1980). An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184. 19

[8] Comba, P. G. (1990). Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538. 28

[9] Dahab, R. and Hernandez, J. C. L. (1997). Técnicas criptográficas modernas - algoritmos e protocolos. In *Atualizações em Informática*, pages 115–170. Editora PUC-Rio. vii, 3, 7

[10] Damgard, I. (1989). A design principle for hash functions. *In Advances in Cryptology - CRYPTO '89 Proceedings*, Lecture Notes in Computer Science Vol. 435:416–427. 9

[11] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEE Transactions on Information Theory*, IT-22:644–654. 5

[12] ElGamal, T. (1985). A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEE Transactions on Information Theory*, 31 (4):469–472. 2, 12

[13] Gouvêa, C. P. L. and Lopez, J. (2012). Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *J. Cryptographic Engineering*, 2(1):19–29. 21, 24

[14] Grewal, G., Azarderakhsh, R., Longa, P., Hu, S., and Jao, D. (2012). Efficient Implementation of Bilinear Pairings on ARM Processors. In Knudsen, L. R. and Wu, H., editors, *Selected Areas in Cryptography – SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 149–165. Springer. 63

[15] Gura, N., Patel, A., Wander, A., Ebrele, H., and Shantz, S. C. (2004). Comparing elliptic curve cryptography and RSA on 8-bit CPUs. *CHES 2004*, LNCS, vol. 3156:119–132. 29

[16] Hankerson, D., Menezes, A. J., and Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography*. Springer, New York, NY. vii, 2, 3, 13, 17, 24, 25, 26, 29, 32, 33, 34, 36, 37, 38

[17] Hoffstein, J., Pipher, J., and Silverman, J. H. (2010). *An Introduction to Mathematical Cryptography*. Springer, San Francisco. vii, 18

[18] Karatsuba, A. and Ofman, Y. (1962). Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294. 30

[19] Koblitz, N. (1994). *A Course in Number Theory and Cryptography*. Springer-Verlag, New York. 11

[20] Menezes, A., Okamoto, T., and Vanstone, S. A. (1992). Reducing elliptic curve logarithms to logarithms in a finite field. *23rd annual ACM Symposium on Theory of Computing*, pages 80–89. 22

[21] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. (2001). *Handbook of Applied Cryptography*. CRC Press, Waterloo. vii, 3, 20, 25, 27, 28, 35

[22] Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521. 33

[23] Rabin, M. O. (1979). Digital signatures and public-key functions as intractable as factorization. *MIT Laboratory of Computer Science Technical Report*. 14

[24] Riviest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 (2):120–126. 2

[25] Sakai, R., Ohgishi, K., and Kasahara, M. (2001). Cryptosystems based on pairing over elliptic curve. *The 2001 Symposium on Cryptography and Information Security*. 22

[26] Sánchez, A. H. and Rodrıguez-Henrıquez, F. (2013). Neon implementation of an attribute-based encryption scheme. Technical Report CACR-2013-7, Centre for Applied Cryptographic Research (CACR) at the University of Waterloo. 62, 63

[27] Sasaki, Y., Wang, L., and Aoki, K. (2009). Preimage attacks on 41-step sha-256 and 46-step sha-512. *Cryptology ePrint Archive Report*, 2009/479:http://eprint.iacr.org. 9

[28] Schneier, B. (1996). *Applied Cryptography*. John Wiley & Sons, Inc, New York. 2

[29] Scott, M. and Szczechowiak, P. (2007). Optimizing multiprecision multiplication for public key cryptography. *Cryptology ePrint Archive Report*, 2007/299:http://eprint.iacr.org. 56

[30] Sloss, A. N., Symes, D., and Wright, C. (2004). *ARM system developer's guide: designing and optimizing system software*. Elsevier, San Francisco. 41

[31] Stein, J. (1967). Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405. 36

[32] Stinson, D. R. (2005). *Cryptography Theory and Practice*. Chapman & Hall/CRC, Waterloo. 1, 10

[33] Wang, X., Yu, H., and Yini, Y. L. (2005). Efficient collision search attacks on sha-0. *CRYPTO 2005*. 9

[34] Western, A. E. and Miller, J. C. P. (1968). Tables of indices and primitive roots. *Royal Society Math at the Cambridge University Press*, 9:MR 39:7792. 19

[35] Xie, T. and Feng, D. (2009). How to find weak input differences for md5 collision attacks. *Cryptology ePrint Archive Report*, 2009/223:http://eprint.iacr.org. 9