Uma Implementação de Criptografia de Curvas Elípticas no Java Card

Danival Taffarel Calegari

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Implementação de Criptografia de Curvas Elípticas no Java Card

Danival Taffarel Calegari[1]

Março 2002

Banca Examinadora:

- Prof. Dr. Ricardo Dahab
  Instituto de Computação, Unicamp (Orientador)
- Prof. Dr. José Raimundo de Oliveira
  Faculdade de Engenharia Elétrica e de Computação, Unicamp
- Prof. Dr. Guido Costa Souza de Araújo
  Instituto de Computação, Unicamp
- Prof. Dr. Paulo Lício de Geus
  Instituto de Computação, Unicamp (Suplente)

[1] **O autor é Bacharel em Ciência da Computação pela Universidade Federal do Mato Grosso do Sul.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Calegari, Danival Taffarel

C128i Uma implementação de criptografia de curvas elípticas no Java Card / Danival Taffarel Calegari -- Campinas, [S.P. :s.n.], 2002.

Orientador : Ricardo Dahab

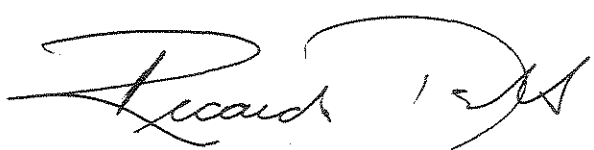
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Java (Linguagem de programação). 3. Redes de computação – Medidas de segurança. 4. Curvas elípticas. I. Dahab, Ricardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Uma Implementação de Criptografia de Curvas Elípticas no Java Card

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Danival Taffarel Calegari e aprovada pela Banca Examinadora.

Campinas, 8 de março de 2002.



Prof. Dr. Ricardo Dahab
Instituto de Computação, Unicamp
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 08 de abril de 2002, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. José Raimundo de Oliveira
FEEC

Prof. Dr. Guido Costa Souza de Araújo
IC - UNICAMP

Prof. Dr. Ricardo Dahab
IC - UNICAMP

Agradecimentos

Agradeco primeiramente a Deus, por ter me concedido este dom maravilhoso que é a vida, por estar sempre ao meu lado nos momentos mais difíceis e por nunca ter deixado de atender às minhas preces. Sem ele eu não teria encontrado forças para superar os inúmeros desafios que encontrei em minha vida.

Agradeço aos meus pais, Danilo e Zaida, por terem sempre feito grandes sacrifícios para não deixarem faltar nada a mim e para permitir que eu realizasse meus sonhos. Um obrigado especial a minha noiva Cristiane, por estar ao meu lado sempre, dando muito amor, carinho e apoio, tornando minha vida muito mais iluminada.

Agradeço aos colegas de república Rogério e José Augusto, pela grande amizade, pelo companheirismo nas "sessões de auto-piedade", pela convivência que tornou a vida longe de casa uma experiência fantástica. Agradeço ao meu grande amigo Valtemir, por toda a força que sempre me deu em grandes momentos de decisão e escolhas de minha vida. Agradeço também a Hana e ao Fred, que sempre foram amigos muito especiais.

Aos grandes amigos do fut99ic, ou mais conhecido por *fut-fight*, pelos momentos de descontração e amizade (e alguns hematomas também) que tornavam as tardes de quarta-feira muito esperadas e alegres. Um muito obrigado também aos meus amigos da Computação da UFMS, aos colegas da pós-graduação da Unicamp e aos colegas do CPqD, que além de oferecer um ambiente agradável de convivência, sempre se dispuseram a trocar conhecimento e oferecer sua amizade.

Um muito obrigado ao meu orientador Ricardo Dahab, pela sua disposição em sempre indicar qual o caminho, em sua paciência de aguardar resultados demorados, por sempre oferecer uma palavra amiga e de apoio quando ocorreram resultados ruins, sem a qual não seria possível dar continuidade a jornada que levou a conclusão deste trabalho.

Resumo

Os smart cards são dispositivos com tamanho e formato semelhantes ao de um cartão de crédito comum, com o diferencial de serem equipados com um *chip* com poder de processamento e uma quantidade de memória muito superior a dos cartões de tarja magnética, cerca de 8K bytes. Estas características permitem o armazenamento de informações sigilosas, além de possibilitar o cálculo de sofisticadas funções criptográficas. Esses fatores fazem dos smart cards dispositivos móveis ideais para identificação de usuários e, portanto, extremamente úteis em aplicações voltadas a prestação de serviços como cartões de saúde, de crédito e outros.

Uma das várias plataformas de smart cards que tem obtido destaque é o Java Card, uma versão reduzida da arquitetura Java para sua adequação à ambientes restritos. No entanto, a especificação dos recursos criptográficos disponibilizados no Java Card definiu o uso do algoritmo de chave pública RSA, que atualmente necessita de chaves com tamanho grande demais para dispositivos com pouca memória. Além disso, a aritmética modular necessária para o RSA requer o uso de um co-processador, o que introduz um custo adicional nos cartões.

Uma alternativa ao RSA é a utilização de sistemas criptográficos baseados em curvas elípticas, que têm se mostrado os mais adequados para dispositivos com recursos limitados, como é o caso dos smart cards. Assim, o objetivo deste estudo foi verificar a viabilidade da implementação de algoritmos criptográficos baseados em curvas elípticas no Java Card utilizando a linguagem Java disponível no cartão. Um dos resultados do nosso trabalho foi a construção de uma biblioteca portável para a aritmética de curvas elípticas. No entanto, o desempenho dessa aritmética mostrou que ainda há muito o que melhorar antes que tais sistemas sejam úteis na arquitetura Java Card.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O crescimento no comércio por meios eletrônicos experimentado nos últimos anos veio acompanhado da necessidade de prover um grande nível de segurança para os mesmos. Os cartões com tarja magnética, comumente usados em bancos e outras instituições financeiras, permitem que as informações gravadas neles sejam lidas com extrema facilidade e possibilitam o surgimento de diversos tipos de fraudes, tal como a clonagem. Assim, vários estudos foram realizados para que uma alternativa ao uso de tais cartões fosse encontrada, mantendo-se a comodidade e a praticidade de um cartão com dimensões reduzidas com a adição de um elevado grau de segurança.

A solução para o problema descrito acima veio com o surgimento da tecnologia de smart cards. Esses dispositivos possuem o tamanho e o formato semelhantes ao de um cartão de crédito, com poder computacional suficiente para controlar o acesso às informações contidas neles. Além disto, os smart cards têm diversas funcionalidades criptográficas, permitindo que as aplicações consigam realizar comunicação cifrada ou a geração de assinaturas digitais sem a necessidade de expor informações vitais, tais como chaves criptográficas ou mesmo dados pessoais do portador do cartão.

Apesar das diversas vantagens de utilização, os primeiros smart cards podiam ser carregados com apenas uma aplicação e sua programação era feita em hardware ou em uma linguagem proprietária, o que tornava o ciclo de desenvolvimento de software para estes dispositivos muito custoso. O Java Card surgiu como uma alternativa para viabilizar

o desenvolvimento de aplicações em smart cards, através do uso de uma linguagem de programação amplamente difundida, a linguagem Java, e a especificação de uma API padrão. Estas características, aliadas a uma especificação de um ambiente de execução, promoveram a portabilidade de código defendida pela plataforma Java. Além disso, a possibilidade de inserção de várias aplicações e a substituição destas por versões mais novas também foram previstas, tornando o cartão mais versátil e reduzindo o custo de manutenção.

Tendo em vista que os principais serviços oferecidos pelo uso de smart cards são relacionados às funcionalidades criptográficas, na API do Java Card também foram disponibilizadas funcionalidades de acesso a serviços criptográficos. No entanto, a API provê somente o uso do algoritmo RSA para o oferecimento de serviços de chave pública, tais como os de geração de assinaturas digitais. Apesar deste algoritmo ser um padrão de mercado e apresentar um bom nível de segurança, o número de bits utilizado em suas chaves é muito grande para dispositivos com espaço de armazenamento tão limitado quanto os smart cards. Outra desvantagem está no fato de ser necessário um co-processador para a realização da aritmética requerida pelo RSA, o que aumenta significativamente o custo do cartão.

Considerando os problemas supracitados em relação ao RSA, nos últimos anos várias alternativas têm sido estudadas para sua substituição. A alternativa que tem obtido maior destaque é o uso de sistemas criptográficos baseados em curvas elípticas, com diversos trabalhos mostrando sua vantagem em relação ao RSA [25, 65]. No entanto, grande parte dos trabalhos baseiam suas implementações em linguagem *assembly* dos processadores dos cartões, ficando assim estas funcionalidades restritas às plataformas de smart cards empregadas nestes trabalhos.

A construção de uma biblioteca portável que disponibilizasse serviços criptográficos que pudessem evitar o uso de um co-processador e diminuir o tamanho das chaves criptográficas, economizando assim um dos recursos mais escassos dos cartões, a memória, se mostrou uma atividade de grande importância. Assim, nosso trabalho tem como proposta verificar a viabilidade e efetivar a construção de um sistema criptográfico baseado com curvas elípticas, provendo economia de memória e poder de processamento, em uma

plataforma aberta e com grande portabilidade de código, o Java Card.

Os sistemas criptográficos baseados em curvas elípticas são constituídos de três camadas, sendo a inferior um conjunto de operações aritméticas sobre corpos finitos, que servem de base para operações sobre elementos de um grupo algébrico cíclico, a segunda camada. A terceira camada é formada pelos algoritmos e protocolos criptográficos, que fazem uso das funcionalidades da última camada. Em termos de desempenho da biblioteca, a camada que possui o requisito de desempenho mais crítico é a aritmética no corpo finito, uma vez que uma operação em uma camada superior pode desencadear um número muito grande de operações nesta camada.

Tendo em vista as observações feitas acima, daremos maior enfoque às operações realizadas entre elementos do corpo finito, uma vez que é o local do sistema onde identificamos um maior risco com relação ao desempenho. Para tanto, adotamos a abordagem de construir um gerador automático de código para a aritmética no corpo finito, com o objetivo de combinar as vantagens de duas estratégias de implementação encontradas na literatura, a velocidade obtida em implementações para corpos específicos e a flexibilidade de implementações para corpos parametrizáveis.

Uma contribuição importante foi a forma como as classes foram modeladas. Esta modelagem, que fez grande uso de interfaces, teve como objetivo acompanhar as tendências observadas na evolução da API do Java Card. Assim, foi obtida uma estrutura aderente à API padrão do Java Card e com baixo acoplamento entre as camadas do sistema criptográfico, sem o que o gerador de código construído não alcançaria o nível de flexibilidade de mudança dos parâmetros do corpo finito que foi almejado pelo projeto.

Tendo em vista que a implementação de criptografia envolve um bom conhecimento das características físicas da arquitetura alvo, no Capítulo 2 será apresentada uma visão geral dos smart cards, detalhando sua arquitetura, os mecanismos de comunicação com o mundo externo e seus principais padrões. Como a plataforma alvo para implementação foi o Java Card, o Capítulo 3 contém uma discussão de como a tecnologia Java foi portada para o ambiente restrito dos smart cards, incluindo as modificações realizadas na arquitetura e no ambiente de execução, a API definida para o Java Card e os mecanismos de segurança inseridos para atender às necessidades impostas pelos smart cards.

Após apresentarmos os aspectos físicos da implementação, no Capítulo 4 abordamos os conceitos relacionados com a teoria de curvas elípticas e as formas de utilização destas estruturas matemáticas para finalidades criptográficas. Um roteiro para a implementação de sistemas criptográficos baseados em curvas elípticas será apresentado no Capítulo 5, com a descrição das decisões de projeto que foram feitas, da forma como os geradores de código para a aritmética básica foram construídos, da implementação da aritmética na curva elíptica e do protocolo criptográfico ECDH.

Uma vez descrita a construção do sistema criptográfico, no Capítulo 6 mostraremos os resultados obtidos em nossa implementação, com medições de tempos para a execução das operações na aritmética básica e com uma discussão sobre a viabilidade da utilização prática da biblioteca disponibilizada. O Capítulo 7 é dedicado às conclusões extraídas do projeto e apresentação de trabalhos que possam ser desenvolvidos futuramente.

Capítulo 2

Smart Cards

Os smart cards são dispositivos com tamanho e formato semelhantes ao de um cartão de crédito comum, com o diferencial de possuírem, em seu interior, um *chip* capaz de armazenar uma quantidade de informações muito superior à dos cartões de tarja magnética. Além disso, este *chip* pode oferecer um poder de processamento suficiente para garantir a integridade e a segurança dos dados armazenados em seu interior, tornando assim os smart cards dispositivos ideais para o armazenamento de informações sigilosas, tais como chaves criptográficas e senhas, inviabilizando atividades fraudulentas, como a clonagem de cartões de crédito.

A idéia de incorporar um circuito integrado em um cartão de plástico surgiu primeiramente em 1968 com dois inventores alemães, Jürgen Dethloff e Helmut Grötrupp, que registraram uma patente na Alemanha. De forma independente, Kuniataka Arimura, do Instituto de Tecnologia Arimura do Japão, registrou uma patente de smart card em 1970. Porém, o progresso real veio por intermédio de Roland Moreno, que entre 1974 e 1979 registrou 47 patentes em 11 países [47]. Uma das idéias revolucionárias de Moreno foi o armazenamento de valores monetários nos cartões para que eles pudessem ser usados em substituição ao papel moeda, de forma segura. Ele também propôs o uso de um PIN (*Personal Identification Number*) para a identificação do portador legítimo do cartão.

Em 21 de março de 1979, a empresa CII-Honeywell Bull (hoje Groupe Bull), em cooperação com a Motorola, apresentou o primeiro smart card totalmente operacional [62],

sendo composto por dois *chips*: uma memória EPROM 2716 e um microprocessador de 8 bits 3870, que originalmente foi desenvolvido pela Fairchild [23]. Apesar desta separação em dois *chips* tornar o dispositivo vulnerável a ataques, este cartão mostrou a viabilidade da idéia e serviu como uma excelente ferramenta para estudos.

Somente em 1981 tornou-se operacional o CP8, produzido pela CII-Honeywell Bull e pela Motorola, um smart card com apenas um *chip* embutido [23]. A partir de então, começaram a ser realizados diversos projetos pilotos na França e na Alemanha com o uso de smart cards como cartões telefônicos pré-pagos e cartões de movimentação de contas em bancos. O sucesso desses projetos mostraram o potencial dos smart cards como dispositivos que oferecem grande segurança e flexibilidade [12].

Recentemente, com os diversos avanços na tecnologia de produção de *chips* e na criptografia moderna, os smart cards se tornaram boas alternativas para o armazenamento de dinheiro eletrônico substituindo papel moeda, para o armazenamento de registros médicos, para impedir o acesso não autorizado a canais de televisão paga e para aumentar a segurança na telefonia sem fio.

Neste capítulo é apresentada uma visão geral dos smart cards. Primeiramente, na Seção 2.1, é apresentado como os smart cards podem ser classificados. Na Seção 2.2 os aspectos físicos dos smart cards são abordados, tais como sua organização interna e modelo de memória. A forma como os cartões se comunicam com o meio externo é mostrada na Seção 2.3, sendo descrito o principal protocolo de comunicação empregado. Por fim, na Seção 2.4, alguns dos padrões mais importantes de smart cards são discutidos brevemente, com um enfoque maior dado ao ISO 7816, o padrão mais importante.

2.1 Classificação dos Smart Cards

A classificação dos smart cards pode ser feita por diversos critérios, sendo os mais comuns sua capacidade de processamento e seu mecanismo de acesso. De acordo com o primeiro critério mencionado, eles são divididos em *memory cards* e em *microprocessor cards* [15, 21]. Já com relação ao segundo critério, que diz respeito à forma de comunicação entre o cartão e o meio externo, os smart cards podem ser classificados em cartões com contato e

sem contato, existindo também versões híbridas que englobam as duas abordagens [47, 23]. Nas seções seguintes são descritas as duas formas de classificação mencionadas acima.

2.1.1 Memory Cards versus Microprocessor Cards

De acordo com a capacidade do *chip* embutido no smart card, ele pode ser classificado em memory card ou em microprocessor card. Em geral, os memory cards contam com uma capacidade de armazenamento que varia entre 1K e 4K bytes, e todo o processamento de dados feito pelo cartão é realizado por um circuito simplificado capaz de executar um conjunto pequeno de instruções programadas [12].

Além disso, nos memory cards as instruções a serem executadas são introduzidas no cartão no momento de sua confecção e não podem ser alteradas após sua emissão, o que é um fator limitante em sua funcionalidade. No entanto, o custo de produção deste tipo de cartão é significativamente menor que o dos microprocessor cards, o que torna esta alternativa mais interessante para mercados que exijam sua utilização em larga escala, como é o caso de algumas empresas do setor de telecomunicações que os empregam em seus sistemas de cartões pré-pagos. Em geral, estes cartões são programados para não aceitarem a inserção de créditos após sua emissão, evitando assim a inclusão de valores de forma ilegal. No entanto, essa característica impede sua reutilização após os créditos carregados acabarem, sendo portanto os cartões vazios descartados.

Os microprocessor cards têm associada à sua grande capacidade de memória um processador capaz de realizar tarefas mais complexas e de gerenciar o acesso aos recursos oferecidos pelo cartão, aumentando assim sua utilidade e segurança. Dessa forma, as aplicações que empregam esta tecnologia ficam limitadas somente pela quantidade de memória disponível e pelo poder de processamento requerido. Muitos dos modelos recentes de microprocessor cards possuem suporte a serviços de criptografia, o que permite a esta tecnologia oferecer um nível elevado de segurança de informações.

Um caso em que a segurança oferecida por este dispositivo fica evidente é em uma aplicação que requeira a emissão de assinaturas digitais, estando a chave privada contida dentro do cartão. Nesse caso, a cadeia de caracteres a ser assinada é transmitida ao cartão e a assinatura digital é gerada pelo próprio microprocessador do dispositivo, **não** ficando

a chave secreta, em questão, exposta ao mundo externo em nenhum momento durante a operação.

Outro fator importante a ser levado em consideração é o custo desta tecnologia. Como resultado de sua produção em massa, o custo dos microprocessor cards caiu drasticamente durante os anos 1990 [12]. Esta queda viabilizou seu emprego em diversas aplicações, tais como telecomunicação sem fio (wireless), controle de acesso, programas de fidelidade, aplicações bancárias, entre outras.

2.1.2 Cartões com Contato versus Cartões Sem Contato

Um dos modos usuais de se classificar smart cards é pela forma como o cartão se comunica com o mundo externo. Nessa classificação, é feita a distinção entre cartões com contato e sem contato. Os cartões com contato possuem em sua superfície uma pequena placa, como mostrado na Figura 2.1, através da qual é realizada a comunicação serial com uma leitora apropriada. Este é o tipo de smart card mais comumente utilizado, sendo empregado em diversas aplicações por bancos, companhias telefônicas, companhias aéreas, entre outras.

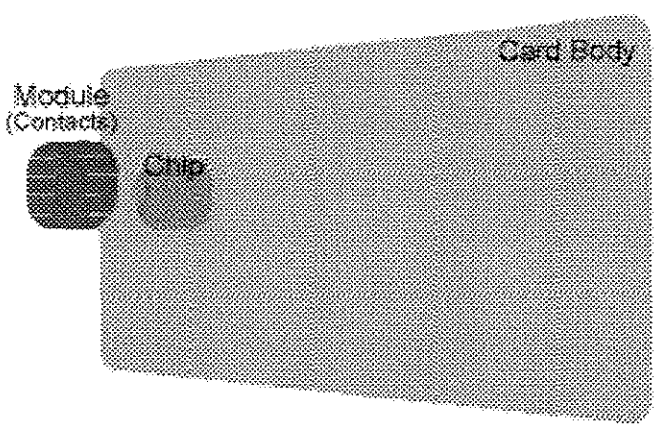


Figura 2.1: Smart card com contato.

Já nos cartões sem contato, a comunicação com o mundo externo é feita através de uma antena embutida no cartão, como apresentado na Figura 2.2. O fornecimento de energia para o cartão pode ser feito através de uma bateria interna ou por uma corrente

induzida na antena [12]. A grande vantagem deste tipo de cartão reside no fato de não ser necessária a inserção do cartão em um dispositivo de leitura, o que torna seu uso muito ágil em aplicações que requerem um grande fluxo de usuários utilizando os cartões, como na cobrança de pedágio em uma rodovia movimentada. Entretanto, este tipo de cartão oferece uma taxa de transmissão muito pequena em relação ao cartão com contato, além de ser uma tecnologia mais cara.

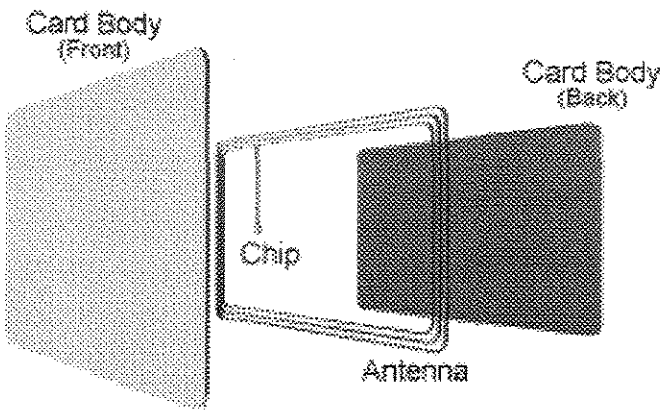


Figura 2.2: Smart card sem contato.

Há também cartões híbridos, que empregam tanto a interface com contato como uma antena embutida. Estes cartões não podem se comunicar através das duas interfaces simultaneamente, mas possibilitam uma grande flexibilidade de aplicações. Esta tecnologia, no entanto, não é muito difundida por ter maior custo de produção.

2.2 Arquitetura Física

Os smart cards são compostos de um *chip* de silício embutido num cartão de plástico que se comunica com os dispositivos externos através de uma placa em seu exterior ou uma antena interna ligada a ele, de acordo com o tipo de cartão em questão (veja seção 2.1.2). No interior do *chip* são encontrados os componentes principais de um smart card, mos-

trados na Figura 2.3, sendo eles uma unidade central de processamento (CPU)[1], uma memória persistente que não pode ser alterada (ROM), uma memória para o armazenamento persistente das informações processadas pelo cartão (tipicamente EEPROM), uma memória mais veloz para ser usada temporariamente durante a execução das instruções (RAM) e um co-processador criptográfico opcional.

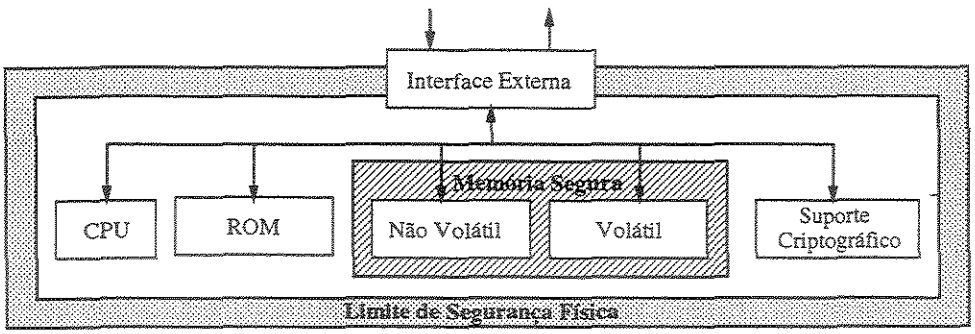


Figura 2.3: Estrutura interna de um smart card.

Uma vez que os smart cards com contato são mais utilizados atualmente, é dado, aqui, maior enfoque a esta tecnologia de cartões. As características físicas destes cartões são definidas por uma série de padrões, entre eles o ISO 7816, que estabelece, entre outras propriedades, as dimensões que os cartões devem possuir, os locais onde o *chip* e a placa de contato devem estar posicionados, qual a funcionalidade de cada ponto de contato, resistência a torções e a outras provações, tanto de natureza física quanto de ordem magnética, que o cartão deve possuir. Apresentamos a seguir uma visão geral de cada um dos aspectos físicos mais importantes.

Apesar de algumas das tecnologias de maior destaque, como o Java Card, ocultarem dos desenvolvedores os detalhes físicos dos smart cards, o seu conhecimento pode indicar as escolhas mais apropriadas das aplicações. Inicialmente, é abordada a especificação de cada um dos pontos de contato do cartão, por onde é realizada a comunicação serial com o mundo externo. Em seguida, mencionamos algumas observações sobre a unidade central de processamento comumente utilizada, e posteriormente descrevemos o uso de

[1]Somente os microprocessor cards possuem CPU, os memory cards possuem uma lógica não-programável para atender aos requisitos de segurança impostos pelas aplicações.

co-processadores e o sistema de memória, um dos itens mais importantes no desempenho das aplicações.

2.2.1 Especificação dos Pontos de Contato

Um smart card possui oito contatos; suas atribuições funcionais são mostradas na Figura 2.4. As dimensões e a localização dos contatos são especificadas na parte 2 do padrão ISO 7816; descrevemos a seguir, brevemente, a funcionalidade de cada um deles.

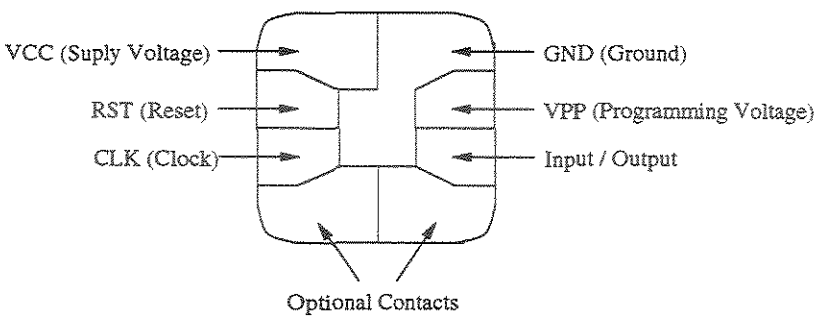


Figura 2.4: Especificação dos contatos de um smart card.

- O contato VCC é responsável pelo suprimento de energia ao *chip*. A voltagem deve ser em torno de 3 a 5 volts, com uma variação máxima de 10 %. Os smart cards em telefones móveis normalmente tem a voltagem ajustada para 3 volts [12].

- O contato RST é usado para o recebimento de um sinal para o cartão restaurar o estado inicial do microprocessador, ou seja, para o estado do cartão no momento em que ele é ativado. Este procedimento é chamado de reinicialização quente (*warm reset*).

- O contato CLK é um sinal de relógio (*clock*) inserido no cartão pelo dispositivo de leitura, uma vez que o smart card não possui um mecanismo de sincronismo interno.

- O contato GND é usado como voltagem de referência, sendo o seu valor considerado como zero volts.

- O contato VPP é opcional e está presente apenas para compatibilidade com sistemas antigos de smart cards, que empregavam memória EPROM. Esta memória requeria uma grande quantidade de energia para a gravação de informações; assim, este contato era utilizado para se obter a voltagem necessária para a gravação de informações e das aplicações na memória do cartão. Nos sistemas atuais este contato não é mais empregado.

- O contato de Input/Output é a interface do smart card com o mundo exterior, por onde é realizada comunicação half-duplex.

- Os contato RFU são reservados para uso futuro, sendo desta forma considerados opcionais.

Como observado, tanto o suprimento de energia do cartão como os sinais de sincronismo são fornecidos por uma fonte externa ao cartão, o que expõe o cartão a um tipo de ataque conhecido como DPA (*Differential Power Analisys*) [33]. Esta técnica induz o cartão a realizar diversas operações com grandes variações de tensões e analisa suas respostas através de técnicas estatísticas para a dedução do conteúdo de seu interior.

Apesar desta técnica ser de difícil execução, uma vez que são necessários equipamentos especiais e de conhecimentos matemáticos e estatísticos sofisticados, diversas empresas de smart cards já têm incluído em seus cartões mecanismos que impedem a introdução de tensões fora das especificações, prevenindo este tipo de ataque.

2.2.2 Unidade Central de Processamento

A unidade central de processamento usada atualmente na maioria dos smart cards é um microcontrolador de 8 bits, com conjuntos de instruções do Motorola 6805 ou do Intel 8051 [65], e com freqüência de clock em torno de 5MHz. No entanto, alguns cartões possuem multiplicadores de clock que elevam a freqüência num fator de 2, 4 ou 8 vezes, chegando assim a operar em até 40MHz.

Recentemente, têm sido produzidos *chips* para smart cards com microcontroladores de 16 e 32 bits, existindo também vários trabalhos que visam empregar a arquitetura

RISC nestes dispositivos. Estes avanços ainda são alternativas muito caras para serem empregados na prática, porém, num futuro próximo eles viabilizarão o surgimento de uma grande gama de aplicações que a tecnologia de smart cards atual não permite.

2.2.3 Co-Processadores

Uma vez que a segurança é uma das principais motivações para a utilização de smart cards, sua capacidade de realizar operações criptográficas é uma característica fundamental. Devido às grandes limitações de poder de processamento (ver Seção 2.2.2), por muitas vezes é incluído no *chip* um co-processador capaz de realizar operações básicas de algoritmos criptográficos, como a aritmética modular para números inteiros muito grandes.

Como o co-processador tem que fazer parte do *chip*, por medida de segurança, ele ocupa uma parcela considerável de espaço dentro da pastilha de silício e introduz uma maior complexidade no projeto do *chip*, podendo representar de 20% a 30% de seu custo total [10].

2.2.4 Sistema de Memória

Os smart cards usualmente contêm três tipos de memória: memória persistente que não pode ser alterada, memória persistente que pode ser alterada e memória volátil. As tecnologias de memória comumente utilizadas são ROM (*read-only memory*), EEPROM (*Electrical Erasable Programmable Read-Only Memory*) e RAM (*Random Access Memory*), respectivamente. Dos três tipos de memória empregados, a ROM é a mais barata; as células de memória EEPROM ocupam cerca de quatro vezes mais espaço no *chip* que as células de ROM e as células de memória RAM são cerca de quatro vezes maiores que as células de EEPROM, sendo desta forma um recurso muito caro.

A memória ROM é usada para armazenar programas fixos no cartão, tais como o sistema operacional e alguns dados que não devem ser alterados durante sua vida útil, como informações sobre o fabricante. Ela é gravada durante o processo de fabricação do cartão, onde é feito um molde com o conteúdo da memória e todos os *chips* produzidos com aquele molde conterão a programação definida pelo fabricante de cartão; este pro-

cesso é conhecido como *masking*. Esta memória tem como vantagem sua capacidade de armazenamento persistente com baixo custo. Por outro lado, não permite alterações após sua produção.

A memória EEPROM, de forma semelhate à memória ROM, tem a capacidade de preservar o seu estado após o fornecimento de energia ao cartão ser interrompido. No entanto, esta tecnologia permite que seu conteúdo seja alterado durante a utilização do cartão. Dois dos parâmetros elétricos mais importantes deste tipo de memória são o número máximo de ciclos de escrita e o tempo máximo de retenção das informações. Na maioria dos cartões, são permitidas pelo menos 100.000 ciclos de escrita e as informações ficam armazenadas por cerca de 10 anos. Apesar de apresentar um tempo de leitura muito próximo ao da memória RAM, o tempo de escrita na EEPROM é cerca de 1.000 vezes maior.

Uma alternativa à memória EEPROM que recentemente tem obtido grande destaque é a memória *flash*. Esse tipo de memória ocupa uma quantidade menor de espaço físico que a EEPROM e consome menos energia, uma das grandes preocupações da utilização de smart cards em telefones celulares. No entanto, apesar de permitir a leitura bit a bit, as alterações só podem ser realizadas em blocos, limitando seu uso ao armazenamento de programas adicionais e atualização de grandes porções de informação.

Por fim, a memória RAM é usada para o armazenamento temporário de informações durante o processamento, uma vez que, diferentemente das memórias citadas anteriormente, não são capazes de manter suas informações após a interrupção do fornecimento de energia ao cartão. A RAM pode ser acessada um número ilimitado de vezes e possui tempo de escrita muito menor que o da EEPROM, tornando-se uma excelente alternativa para melhorar o desempenho das aplicações. Por ocupar grande parte do *chip*, um smart card não pode dispor de grande quantidade desta memória, sendo seu tamanho, em geral, limitado em 512 a 1K bytes.

2.3 Comunicação com o Mundo Externo

A interface dos smart cards com o mundo externo é feita através dos contatos em sua superfície. Para que a comunicação ocorra, o cartão deve ser inserido em um dispositivo que reconheça a funcionalidade de cada contato (descritas na Seção 2.2.1). Estes dispositivos são comumente conhecidos como CAD (*Card Acceptance Device*) e são classificados em dois tipos: leitoras e terminais.

As leitoras são conectadas na porta paralela, serial ou USB de um computador, com a qual a comunicação é realizada. Uma vez que normalmente as leitoras não possuem poder de processamento, elas servem apenas como um canal de comunicação entre o smart card e o computador. Mecanismos adicionais de detecção e recuperação de falhas na transmissão podem estar presentes.

Já os terminais, por sua vez, são computadores completos. Um terminal tem uma leitora de smart card como parte de seus componentes, realizando desta forma o processamento das informações obtidas do cartão. Exemplos deste tipo de CAD são os caixas automáticos de bancos, que possuem locais especiais onde os cartões devem ser inseridos para o acesso aos serviços disponibilizados pela instituição bancária a seus clientes.

Por razões de simplicidade, no restante do texto os dispositivos de comunicação serão referenciados apenas por CAD, uma vez que tanto as leitoras quanto os terminais se encaixam nesta denominação. A seguir, apresentamos outros aspectos da forma como a comunicação com os cartões ocorrem, tais como o modelo de comunicação, o formato dos pacotes de dados definidos pelo ISO 7816, e a configuração inicial dos parâmetros de comunicação através do sinal de ATR (*Answer to Reset*).

2.3.1 Modelo de Comunicação

O cartão se comunica com o mundo exterior através de um canal de comunicações half-duplex, ou seja, os dados podem ser transmitidos em ambas direções, porém em um dado momento só podem estar sendo transmitidos dados em uma única direção.

O protocolo de comunicação empregado usa pacotes de dados com um formato especificado pelo padrão ISO 7816 (ver Seção 2.3.2). O modelo mestre-escravo é empregado

para determinar o relacionamento do cartão com os sistemas externos. Um smart card fica em estado de espera até que um comando lhe seja enviado; o comando enviado é validado, executado e uma resposta é enviada ao requerente com o resultado da operação realizada. Este modelo é ilustrado na Figura 2.5.

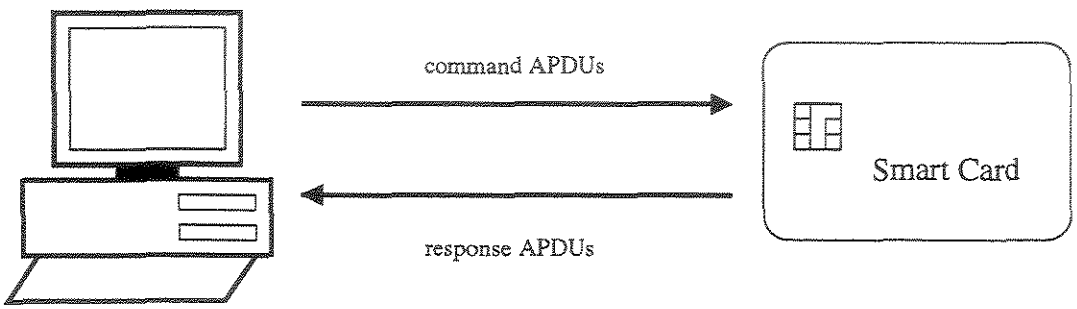


Figura 2.5: Modelo de comunicação dos smart cards.

2.3.2 Protocolo APDU

A parte 4 do ISO 7816 (Seção 2.4) especifica o protocolo de comunicação da camada de aplicação para a interação entre o smart card e uma aplicação em um sistema externo, o protocolo APDU (*Application Protocol Data Units*) [21]. Este protocolo é composto de dois tipos de pacotes, as APDUs de comando (C-APDU) e as APDUs de resposta (R-APDU). As primeiras contêm uma instrução completa para ser executada e são enviadas ao smart card pelo sistema externo, conforme o modelo de comunicação mostrado na Seção 2.3.1. Já o segundo tipo contém uma resposta completa do cartão.

O formato especificado da C-APDU é mostrado na Figura 2.6, sendo o cabeçalho do comando constituído por 4 bytes:

1. **CLA** (*Class Byte*) – identifica qual classe de instrução deve ser executada (por exemplo, se a instrução é para a aplicação ou diretamente para o sistema operacional do cartão);

2. **INS** (*Instruction Byte*) – contém um identificador para a instrução que deve ser executada;

3. **P1** – parâmetro da instrução;

4. **P2** – parâmetro da instrução.

Header				Conditional Body		
CLA	INS	P1	P2	Lc	Data Field	Le

Figura 2.6: Formato da APDU de comando.

Os bytes seguintes são compostos por um corpo opcional da mensagem e têm tamanho variável. Caso a aplicação no dispositivo com o qual o smart card está se comunicando necessite enviar um conjunto de dados no pacote (desde que não seja adequado usar os campos P1 e P2), ela deverá incluir um campo do tipo Lc que contém o número de bytes a ser enviado ao cartão seguido dos dados propriamente ditos. Também é previsto o caso de ser necessária a obtenção de um conjunto de dados provenientes do cartão, caso em que o campo Le especifica o número máximo de bytes que são esperados como resposta ao comando contido na C-APDU.

A R-APDU, mostrada na Figura 2.7, é composta por um corpo opcional e por um trailer obrigatório. No corpo do pacote são inseridos os dados resultantes da execução da última C-APDU recebida pelo cartão, cujo tamanho não deve ultrapassar o máximo estabelecido pelo comando, ou seja, o valor enviado no campo Le da C-APDU recebida. Já o trailer obrigatório, formado pelos bytes SW1 e SW2, é usado para informar à aplicação como o comando foi executado; ou seja, ele traz em seu interior o código de possíveis erros que possam ter ocorrido ou o valor "0x9000", que por convenção é usado para informar que o comando foi processado com sucesso.

Conditional Body	Status word	
DataField	SW1	SW2

Figura 2.7: Formato da APDU de resposta.

2.3.3 Protocolo de Handshake

Imediatamente após o smart card ser acionado, é enviado um sinal, conhecido por ATR (*Answer to Reset*), pelo cartão para a máquina com a qual ele está conectado. Esta mensagem contém as informações requeridas pelo cartão para que seja estabelecida a comunicação inicial. O sinal de ATR contém 33 bytes e neles estão os parâmetros de transmissão, tais como os protocolos de transporte aceitos pelo cartão[2], a taxa de transmissão de dados, parâmetros de hardware do cartão, tais como o número serial do *chip* e o número da máscara, além de outras informações empregadas pelo dispositivo externo para que este possa fazer uso do cartão.

2.4 Padrões e Especificações

A padronização possui um papel chave na aceitação e crescimento da indústria de smart cards. Somente os padrões internacionais apropriados podem assegurar que smart cards operem corretamente com diferentes leitoras e terminais, tais como caixas eletrônicos, em diferentes lugares do mundo [27].

Dessa forma, nos últimos 15 anos houve um grande trabalho para a especificação de diversos padões para que os smart cards, CADs e aplicações desenvolvidas trabalhassem em conjunto, mesmo que produzidos por diferentes fabricantes. O principal padrão foi o ISO 7816, sendo os outros construídos por diferentes segmentos da indústria para que suas necessiades específicas fossem atendidas. Nesta seção serão apresentados alguns dos principais padrões e especificações encontrados na indústria de smart cards.

O principal padrão para os smart cards, o ISO 7816, mostrado na Sessão 2.4.1, serve como base para todos os outros.

[2]Os protocolos de comunicação estabelecidos pelo ISO 7816 são dois, sendo o primeiro orientado a bytes, também conhecido por protocolo T=0, e o segundo orientado a blocos, conhecido por protocolo T=1.

2.4.1 ISO 7816

A ISO (*International Standards Organization*) lançou uma especificação para smart cards com contato, o ISO 7816, o padrão mais importante na definição das características físicas e elétricas dos cartões [29]. Este padrão é dividido em sete partes, cobrindo desde as dimensões que um smart card deve apresentar até os protocolos que devem ser empregados para a comunicação com os cartões:

- **Parte 1 – Características Físicas**: Define as dimensões físicas dos smart cards com contato e sua resistência à eletricidade estática, radiação eletromagnética e impactos físicos. Também especifica a localização física de uma possível tarja magnética e da área para a gravação das informações de personalização na superfície do cartão.

- **Parte 2 – Dimensões e Localização dos Contatos**: Define a localização, o propósito e as características elétricas de cada contato metálico do cartão.

- **Parte 3 – Sinais Eletrônicos e Protocolos de Transmissão**: Define a tensão e as exigências atuais para os contatos elétricos definidos na Parte 2, incluindo também um protocolo de transmissão assíncrona de caracteres (T=0). Há duas extensões que incluem um protocolo de transmissão assíncrona de blocos (T=1) e uma revisão da seleção do tipo de protocolo.

- **Parte 4 – Comandos entre Indústrias para Intercâmbio**: Estabelece um conjunto de comandos através dos quais todas as indústrias provêem acesso, segurança e transmissão dos dados contidos no cartão. Há comandos para leitura, escrita e atualização de registros, entre outros.

- **Parte 5 – Sistema de Numeração e Procedimentos de Registro para Identificadores de Aplicações**: Estabelece padrões para AIDs (*Application Identifiers*). Um AID possui duas partes, sendo a primeira um identificador de provedor de aplicação registrado (RID) de cinco bytes que é único para cada fabricante. A segunda parte é um campo de tamanho variável de no máximo 11 bytes, os quais são utilizados pelos fabricantes para diferenciar suas aplicações.

- **Parte 6 – Elementos de Dados entre Indústrias**: Descreve regras de codificação para os dados necessários em muitas aplicações, como o nome e a fotografia do proprietário do cartão, sua preferência de língua, etc.

2.4.2 EMV

Em 1996 a EMV, um consórcio formado pelas empresas Europay, MasterCard e Visa, lançou um padrão conhecido por Especificações de Cartões com Circuitos Integrados para Sistemas de Pagamento, que combina um subconjunto das partes 1 a 6 da ISO 7816 com características adicionais em um projeto destinado a atender as necessidades específicas das aplicações destas empresas [57]. A versão mais recente desta especificação, a EMV 96 versão 3.1.1, foi publicada em maio de 1998 e conta com três partes:

- Especificação de cartões com circuito integrado;
- Especificação de terminais para cartões com circuito integrado;
- Especificação de aplicações para cartões com circuito integrado.

2.4.3 GSM

O Instituto de Padrões em Telecomunicações Europeu (ETSI) publicou um conjunto de padrões para a utilização de smart cards no sistema público de telefonia celular [19]. O GSM (*Global System for Mobile Communications*) foi desenvolvido pelo ETSI como um padrão internacional para o sistema de telefones celulares. Originalmente estabelecido para cobrir apenas um pequeno conjunto de países do mercado europeu, este sistema tem ganho muita força em diversos mercados, com grande investimento das grandes empresas de desenvolvimento em telecomunicações.

Entre as tecnologias envolvidas neste sistema de telefonia celular, está previsto o uso de smart cards para armazenar as informações personalizadas dos usuários e para prover um conjunto de funcionalidades criptográficas que permitam a verificação mútua de autenticidade entre a operadora e o aparelho, dificultando assim o surgimento de fraudes no sistema. Estes cartões são conhecidos como SIM cards (*Subscriber Identification Module*)

e, entre os diversos padrões que compõem o GSM, os que se destacam no contexto de smart cards são:

- GSM 11.11 – especificação da interface entre o SIM e o equipamento de comunicação.
- GSM 11.14 – especificação da interface entre as ferramentas disponibilizadas no SIM com o equipamento de comunicação.
- GSM 03.48 – especificação dos mecanismos de segurança disponibilizados pelas ferramentas do SIM.
- GSM 03.19 – SIM API para a plataforma Java Card. Este padrão, baseado no GSM 11.11 e no GSM 11.14, define a API Java para o desenvolvimento de aplicações GSM que devem ser executadas na plataforma Java Card. Esta API é uma extensão da API 2.1 do Java Card.

2.4.4 Open Platform

A Open Platform define um ambiente integrado para o desenvolvimento e operação de smart cards que suportem múltiplas aplicações [22]. Esta especificação é constituída de um conjunto mínimo de requerimentos que os fabricantes de smart cards devem seguir para prover funcionalidades que tornem as aplicações interoperáveis entre os diversos cartões em conformidade com a plataforma; um dos principais pontos é a compatibilidade dos smart cards com os padrões ISO e EMV.

Esta especificação foi inicialmente desenvolvida pela Visa e atualmente foi transferida para a GlobalPlatform, uma organização com finalidade de promover uma infra-estrutura global para implementação de smart cards entre várias indústrias.

2.4.5 OpenCard Framework

O OCF (*OpenCard Framework*) foi introduzido pela IBM, sendo atualmente administrado pelo OpenCard consortium, que é formado pelas maiores empresas da indústria de smart cards [45]. O OCF é um framework utilizado em aplicações no lado dos dispositivos que

se comunicam com smart cards e que provê uma interface padronizada para a interação com CADs e aplicações nos cartões.

A arquitetura do OCF é um modelo estruturado de forma a dividir funcionalidades entre fabricates de CADs, desenvolvedores de sistemas operacionais para smart cards e empresas que realizam a emissão destes cartões. O objetivo principal deste framework é reduzir a dependência entre cada uma destas partes, além de também buscar a independência ente os fabricantes de cartões.

2.4.6 PC/SC

A especificação PC/SC é definida como a interoperabilidade entre smart cards e computadores pessoais (PCs), daí o seu nome. Sua especificação e manutenção é de responsabilidade do PC/SC Workgroup, um consórcio entre as principais indústrias de smart cards [46].

Nesta arquitetura, as aplicações situadas nos computadores pessoais são construídas sobre um ou mais *service providers* e um *resource manager*. Um *service provider* encapsula a funcionalidade de um smart card específico tornando-a acessível através de interfaces de programação disponibilizadas para a aplicação. Um *resource manager* gerencia os acessos aos dispositivos de comunicação com os cartões.

O PC/SC e o OCF possuem conceitos bastante similares: quando executando sobre a plataforma Windows, o OCF pode acessar os dispositivos de comunicação com os cartões através de um resource manager de PC/SC instalado.

Capítulo 3

Tecnologia Java para Smart Cards

O desenvolvimento das primeiras aplicações para smart cards foi um grande desafio, dada a natureza repleta de restrições deste dispositivo. Além disso, era uma prática comum o uso de linguagens de programação proprietárias, o que aumentava o custo de desenvolvimento, uma vez que não era possível encontrar desenvolvedores experientes para o desenvolvimento no mercado, o que acarretava um custo grande no treinamento de pessoal. Assim, as indústrias de smart cards procuraram compensar essa deficiência investindo na disponibilização de arquiteturas de programação que fossem baseadas em linguagens de programação bem conhecidas e menos complexas. Uma das primeiras empresas a tomar tal atitude foi a Schlumberger, que em 1996 lançou o primeiro Java Card, um cartão capaz de processar aplicações escritas em linguagem Java.

Com o cartão lançado pela Schlumberger, a Sun Microssystems, detentora oficial da tecnologia Java, se viu obrigada a escrever uma especificação para esta tecnologia, sendo lançado em novembro de 1996 a versão 1.0 da API Java Card[1].

No entanto, foi constatado que esta versão da API não estava muito bem definida, principalmente em relação ao ambiente de execução. Assim, após um ano foi lançada a versão 2.0 da API [56], sendo esta apoiada na experiência obtida na primeira versão e

[1]Uma API (*Application Programming Interface*) da tecnologia Java é uma especificação definida pela Sun Microssystems com um conjunto mínimo de funcionalidades que devem ser disponibilizadas pelos fabricantes nos produtos que empreguem a tecnologia, além de oferecer uma implementação de referência (JDK) da arquitetura. Nas versões mais recentes do Java Card (a partir da 2.0), além da API básica, a Sun também especificou o subconjunto da linguagem Java permitido (*Java Card Language Subset Specification*) e o comportamento da máquina virtual (*Java Card Runtime Environment Specification*).

com o auxílio do Java Card Forum, que é um consórcio formado por grandes empresas do ramo de smart cards, tais como a própria Schlumberger, a Bull e a Gemplus. Vários pontos que haviam sido deixados de lado na primeira versão foram cobertos, tais como a compatibilização com o ISO 7816 (Seção 2.4.1) e EMV (Seção 2.4.2), além de uma melhor especificação do ambiente de execução das aplicações (as aplicações desenvolvidas para a plataforma Java Card são conhecidas por *applets*). Entretanto, a forma como os *applets* deveriam ser introduzidos nos cartões ficou descoberta, prejudicando assim a interoperabilidade entre os diferentes fabricantes.

A API 2.1 foi lançada em março de 1999, com pequenas alterações em relação à API 2.0 [58]. O grande avanço atingido nesta especificação foi a formalização da arquitetura da máquina virtual e do processo de carga dos *applets* no cartão, permitindo assim a interoperabilidade entre as plataformas de Java Card disponíveis [12]. A versão mais recente da API é a 2.1.1 [59], que foi lançada em maio de 2000 e apresenta como característica marcante uma melhor definição da geração de chaves criptográficas pelo cartão [60]. Com esta evolução da tecnologia, o Java Card foi amplamente aceito e já foi licenciada por mais de 90% dos fabricantes de smart cards [28].

Neste capítulo é descrito como a tecnologia Java foi portada para o ambiente restrito dos smart cards. Para tanto, na Seção 3.1 é mostrada uma visão geral da arquitetura definida pela Sun para o Java Card, com uma breve discussão sobre seus principais aspectos. Na Seção 3.2 são apresentadas as funcionalidades disponíveis na linguagem Java que foram portadas para a plataforma Java Card e são especificadas, também, quais não foram. Um detalhamento das partes que compõem a máquina virtual é dado na Seção 3.3. O ambiente de execução é mostrado na Seção 3.4, sendo abordada a forma como a máquina virtual se integra a este ambiente e seus demais componentes. Na Seção 3.5 são descritos os principais serviços criptográficos requeridos pelas aplicações que utilizam o Java Card e como a especificação fez o tratamento de cada um deles.

3.1 Visão Geral da Arquitetura

Atualmente, os smart cards representam uma das menores plataformas computacionais em uso. Tipicamente, a configuração da memória fica em torno de 1K de RAM, 16K de EEPROM e 24K de ROM. O maior desafio de se portar a plataforma Java para estes dispositivos é inserir todas as funcionalidades requeridas no cartão reservando, ainda, uma quantidade de espaço livre suficiente para que os *applets* possam ser inseridos e possam armazenar os seus dados. A solução encontrada foi empregar um subconjunto reduzido da linguagem Java e implementar um modelo de máquina virtual Java dividido.

A máquina virtual do Java Card é dividida em duas partes: uma que executa fora do cartão, num terminal ou em um computador conectado ao CAD, e uma que executa dentro do cartão. A parte externa da máquina virtual executa as tarefas prévias à carga de uma aplicação e que requeiram uma quantidade grande de recursos computacionais, tais como a verificação de *bytecodes*, a resolução de endereços e a otimização do código. Uma vez que, em geral, o sistema externo possui uma quantidade muito maior de recursos que o cartão, estas tarefas são delegadas a ele. A parte interna é a que efetivamente executa os *applets*, sendo então responsável por disponibilizar as funcionalidades previstas na API e implementar as políticas de segurança definidas pela arquitetura, por exemplo o firewall entre os *applets* (ver Seção 3.5).

Além desta separação da máquina virtual, a disponibilização da plataforma Java para os smart cards teve o requisito de se definir um ambiente de execução que desse suporte a suas características peculiares de gerenciamento de memória, dos protocolos de comunicação, segurança e o modelo de execução de *applets*, atendendo principalmente ao padrão ISO 7816. Além disso, este ambiente de execução oferece uma clara separação entre o sistema de smart cards e as aplicações, ocultando dos desenvolvedores toda a complexidade e os detalhes de cada sistema de smart card. As aplicações simplesmente requerem os recursos e serviços do sistema via uma interface de alto nível bem definida e padronizada e a máquina virtual é que se encarrega de interagir com o sistema do smart card.

3.2 Subconjunto da Linguagem Java

Devido a escassez de memória, a plataforma Java Card aceita somente um subconjunto da linguagem Java escolhido de forma muito cuidadosa. Esse subconjunto inclui características que são especialmente úteis para o desenvolvimento de programas para smart cards e outros dispositivos limitados, sendo mantidas as principais capacidades de orientação a objetos da linguagem de programação Java.

Entre as características de Java disponíveis, destacam-se:

- Tipos primitivos pequenos – suporte somente aos tipos primitivos de Java com tamanho reduzido. Os únicos tipos primitivos disponíveis são: `boolean`, `byte` e `short`. O suporte ao tipo `int` é opcional e depende unicamente da plataforma de smart card usada, uma vez que já há cartões com processadores de 32 bits disponíveis no mercado.

- Arrays unidimensionais – aceita somente arrays de uma única dimensão. Para se trabalhar com arrays multidimensionais é necessário construir a lógica necessária no applet, ou seja, criar arrays cujos elementos são arrays e assim por diante.

- Conceitos de orientação a objetos – aceita o conceito de pacotes, classes, interfaces e exceções da linguagem Java.

Dentre as cacterísticas que não foram disponibilizadas, as mais marcantes são as seguintes:

- Tipos primitivos grandes – os tipos primitivos `long`, `double` e `float` não estão presentes na arquitetura Java Card.

- Caracteres e strings – uso de caracteres e da classe String (`java.lang.String` não permitido.

- Arrays multidimensionais – somente é permitido o uso de arrays de uma única dimensão, como já citado anteriormente.

- Carga dinâmica de classes – não é permitido transferir classes de um sistema externo ao cartão durante a execução de um applet; assim as únicas classes disponíveis são aquelas que já foram disponibilizadas no cartão ou aquelas que foram carregadas previamente pelo processo de instalação.

- Security Manager – o gerenciamento de segurança na plataforma Java Card difere significativamente daquele empregado na plataforma Java. Na segunda plataforma, há uma classe de gerenciamento de segurança (`java.lang.SecurityManager`) responsável por implementar as características de segurança. Já na plataforma Java Card, as políticas de segurança são implementadas pela própria máquina virtual, não existindo assim uma classe com esta responsabilidade.

- Threads – a máquina virtual do Java Card não provê suporte a múltiplas threads e consequentemente qualquer das funcionalidades associadas a este recurso.

- Clonagem – a classe Object da API não implementa o método `clone` e não é provida uma interface `Cloneable`.

O mecanismo coletor de lixo (*garbage collection*) é opcional. Assim, é recomendável que seja feita a reutilização dos objetos criados, uma vez que se forem criados novos objetos durante o ciclo de vida da aplicação não há a garantia de que a memória ocupada pelos novos objetos será liberada, podendo assim o cartão ficar rapidamente sem memória. Uma visão mais detalhada das características de Java suportadas, ou não, no Java Card pode ser encontrada na especificação [61].

3.3 Máquina Virtual

A principal diferença entre a máquina virtual do Java Card (JCVM) e a máquina virtual Java (JVM) é que a JCVM é composta de duas partes separadas. A parte externa, chamada de *conversor*, é executada em um PC ou numa workstation, sendo responsável por carregar e pré-processar os arquivos de classes que compõem um pacote Java e produzir um arquivo CAP (*converted applet*). O pacote é então carregado dentro do Java Card e

executado pela parte da JCVM que executa dentro do cartão, o *interpretador*. Além de gerar o arquivo CAP, o conversor também gera um arquivo de exportação com as APIs públicas do pacote sendo convertido, para permitir sua utilização por outros pacotes.

3.3.1 Conversor do Java Card

Diferente da máquina virtual Java, que processa uma classe por vez, a unidade de conversão do Java Card é um pacote. No desenvolvimento de *applets* para o Java Card, são produzidos arquivos com os códigos fonte das classes que devem ser compilados usando um compilador Java (sendo incluídas as APIs específicas do Java Card no classpath). Este processo produz um conjunto de arquivos, cada arquivo representando uma classe. Estes arquivos são então processados pelo conversor, sendo produzido um único arquivo com a estrutura do pacote formado pelas classes convertidas.

Durante a conversão, são realizadas tarefas que a máquina virtual Java deveria realizar no momento da carga de classes Java, tais como:

- Verificar se as imagens das classes Java são bem formadas.
- Verificar se houve alguma violação do subconjunto da linguagem do Java Card.
- Realizar a inicialização das variáveis estáticas.
- Resolver as referências simbólicas de classes, métodos e campos, produzindo um formato mais compacto que pode ser manipulado de forma mais eficiente no cartão.
- Otimizar o bytecode, utilizando as informações obtidas durante a carga das classes e a resolução de símbolos.
- Reservar espaço e criar as estruturas de dados para representar as classes.

O conversor tem como entrada não só os arquivos de classes a serem convertidos como também um ou mais arquivos de exportação, caso o pacote que está sendo convertido importe classes de outros pacotes carregados previamente. Além de produzir o arquivo CAP, é gerado um arquivo de exportação do pacote convertido, que por sua vez é usado

para disponibilizar as classes do pacote para outros *applets*. O processo como um todo é mostrado na Figura 3.1.

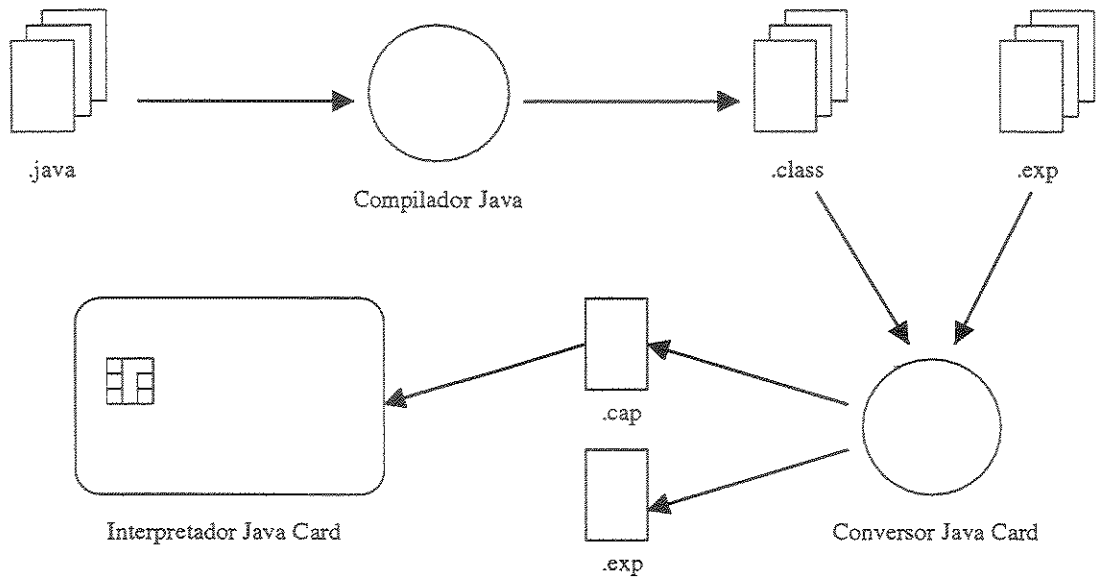


Figura 3.1: Ciclo de desenvolvimento de *applets* para Java Card.

3.3.2 Interpretador do Java Card

O interpretador é o componente da JCVM presente dentro do cartão e com a responsabilidade de executar os *bytecodes* dos *applets*. Este componente provê um ambiente de execução para os elementos do subconjunto da linguagem Java, definidos na especificação da tecnologia Java Card. É de responsabilidade do interpretador realizar as seguintes tarefas:

- Executar as instruções dos *bytecodes* e, conseqüentemente, executar os *applets*.
- Controlar a alocação de memória e a criação de objetos.
- Implementar as políticas de segurança em tempo de execução.

Assim, a máquina virtual do Java Card foi descrita como sendo composta pelo conversor e pelo interpretador. Informalmente, entretanto, nas definições mais recentes a JCVM

é definida somente como sendo a parte que fica no cartão, ou seja o interpretador. A convenção adotada nas primeiras publicações sobre a plataforma ainda é empregada em sua documentação formal; porém, a partir deste ponto o termo JCVM e o interpretador são usados como sinônimos, a menos que seja especificado explicitamente o contrário. Vale ressaltar que a definição clássica da JCVM composta de dois componentes é bastante útil quando é feita a comparação com a plataforma Java, uma vez que as funcionalidades presentes na execução de uma classe Java são obtidas pela combinação do conversor com o interpretador.

3.4 Ambiente de Execução do Java Card

Seguindo o modelo definido pela tecnologia Java, o JCRE (*Java Card Runtime Environment*) é capaz de isolar as aplicações dos detalhes específicos de cada cartão, tais como o gerenciamento de recursos e a comunicação com o meio externo, provendo a independência de hardware e servindo como um verdadeiro sistema operacional de smart card.

A estrutura do JCRE é mostrada na Figura 3.2, na qual pode-se observar que todo o ambiente de execução é executado sobre o sistema operacional nativo do smart card e seu hardware específico. O JCRE é constituído pela máquina virtual do Java Card (o interpretador de *bytecodes*), a API padrão definida para a arquitetura, extensões das funcionalidades da API padrão para atender requisitos específicos das indústrias onde o smart card será empregado e as classes de sistema do JCRE. No topo desta arquitetura são executados os *applets*, que acessam as funcionalidades do sistema através das APIs oferecidas.

3.4.1 JCVM e Métodos Nativos

A camada inferior da JCRE contém a JCVM e os métodos nativos. A JCVM executa os *bytecodes*, controla a alocação de memória, gerencia os objetos e implementa as políticas de segurança definidas para o tempo de execução. Os métodos nativos provêem o suporte à JCVM e às classes de sistema dos níveis superiores. Eles são responsáveis pela manipulação dos protocolos de comunicação de baixo nível, gerenciamento de memória,

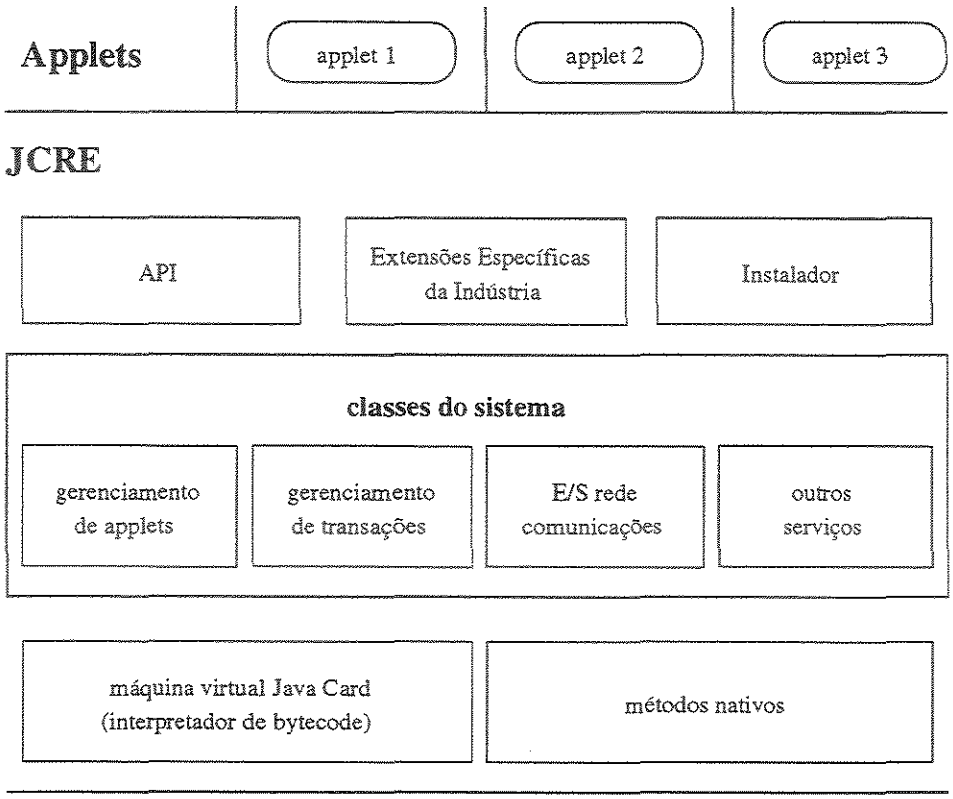


Figura 3.2: Ambiente de Execução do Java Card.

suporte criptográfico, entre outros.

3.4.2 Classes de Sistema

As classes de sistema são análogas ao núcleo do sistema operacional. Elas têm a responsabilidade de gerenciar transações, gerenciar a comunicação entre as aplicações no terminal com os *applets* do Java Card e controlar a criação, seleção e desativação dos *applets*. Para realizar estas tarefas, as classes de sistema tipicamente utilizam os métodos nativos diretamente.

3.4.3 API e Extensões Específicas da Indústria

O framework definido pela especificação do Java Card é constituído das APIs principais e dos pacotes de extensão. As classes da API são compactas e adaptadas para o desenvolvimento de *applets* para smart cards. A maior vantagem deste framework é que ele torna a tarefa de se criar um applet relativamente simples. Os desenvolvedores das aplicações podem se concentrar principalmente nos detalhes das regras de negócio a serem implementadas sem se preocupar com os detalhes da infra-estrutura dos sistemas de smart cards. Os *applets* devem acessar os serviços oferecidos pelo JCRE somente através das classes da API.

Uma indústria específica pode oferecer bibliotecas adicionais de modo a disponibilizar serviços específicos ou um refinamento do modelo de segurança do sistema. Como exemplo, o Open Platform (ver Seção 2.4.4) estende os serviços do JCRE para atender aos requisitos de segurança específicos da indústria financeira. Entre as características incluídas, esta plataforma assegura o controle dos cartões pelas empresas que os expedem e especificam um conjunto de comandos padronizados para a personalização dos smart cards.

3.4.4 Instalador

O instalador permite a carga segura de software e *applets* dentro do cartão após sua entrega ao usuário final, tendo uma grande cooperação com a parte da JCVM que executa externamente ao smart card. Juntos, eles realizam a tarefa de carregar o conteúdo binário de um arquivo CAP na memória do Java Card, podendo oferecer funcionalidades como a verificação de assinaturas criptográficas para atestar a autenticidade do applet sendo carregado, realizar a alocação inicial de recursos para o pacote contido no arquivo CAP, entre outras. Este componente é um item opcional do JCRE, sendo que em sua ausência todo o software de um Java Card, incluindo os *applets*, devem ser inseridos na memória dos cartões durante seu processo de fabricação.

3.4.5 Applets

Os *applets* do Java Card são as aplicações desenvolvidas pelos usuários para serem executadas na plataforma Java Card. Eles são escritos utilizando o subconjunto da linguagem Java definido pela arquitetura Java Card e são executados e gerenciados pela JCRE. Caso o cartão possua um instalador, os *applets* podem ser inseridos após o Java Card ser entregue ao usuário final. Observando a Figura 3.2, nota-se que há uma separação entre os diversos *applets* presentes no smart card. Esta separação é um firewall que impede que um applet de uma determinada fonte acesse funcionalidades de outros *applets* sem a devida autorização. Na plataforma Java Card há a definição de como o compartilhamento entre os *applets* pode ser feito, porém este assunto foge do escopo deste texto. Maiores detalhes do mecanismo de compartilhamento de funcionalidades entre os *applets* podem ser encontrados em [61, 12].

3.4.6 Funcionalidades do JCRE

Além do suporte ao modelo de execução da linguagem Java, o JCRE provê três funcionalidades de tempo de execução:

- *Objetos persistentes e transientes* — Por default, os objetos do Java Card são persistentes e por consequência são criados na memória persistente, sendo desta forma seu estado mantido entre as sessões de uso dos *applets*. Por questões de segurança e de desempenho, os *applets* podem criar objetos em memória RAM, sendo portanto chamados de objetos transientes.

- *Operações atômicas e transações* — A máquina virtual do Java Card assegura que qualquer operação de escrita em um único campo de um objeto de uma classe seja atômica, ou seja, ou o campo sendo atualizado fica com o valor novo ou permanece com o valor antigo, mesmo que o fornecimento de energia seja interrompido durante a operação de escrita. Adicionalmente, o JCRE provê funcionalidades transacionais na API; por exemplo, um applet pode incluir diversas operações de escrita dentro de uma transação, ficando o próprio JCRE encarregado de assegurar que ou todas as alterações são realizadas ou nenhuma, seguindo o princípio da atomicidade.

Porém, esta funcionalidade deve ser usada com bastante cautela, uma vez que requer uma quantidade muito grande de recursos do sistema, que é por natureza bastante limitado.

- *Firewall de applets e mecanismos de compartilhamento* — O firewall de *applets* fornece um isolamento que impossibilita que um applet possa acessar as informações ou mesmo as funcionalidades oferecidas por outras aplicações dentro de um mesmo Java Card. Este mecanismo foi incluído para assegurar que aplicações mal intencionadas tenham acesso a informações não autorizadas contidas no cartão; por exemplo, de um applet que gerencia a conta bancária do usuário. No entanto, o compartilhamento de funcionalidades às vezes se faz necessário, sendo assim definido também um mecanismo que permite o compartilhamento entre as funcionalidades dos *applets*, desde que de forma autorizada.

3.5 Segurança do Java Card

Um dos aspectos mais importantes da linguagem Java é a segurança, tendo como principal enfoque a integridade e segurança dos dados contra aplicações maliciosas. De forma similar a outras linguagens de programação que empregam os conceitos de orientação a objetos, Java provê mecanismos para que sejam especificados os níveis de acesso aos métodos e variáveis de instância das classes, que podem ser públicos, de pacotes, privados ou protegidos. O primeiro nível permite a utilização do método por qualquer objeto do sistema, enquanto os métodos do segundo nível podem ser acessados somente pelos membros do pacote a qual a classe pertence; os demais níveis restringem o acesso aos métodos e variáveis somente aos membros da própria classe, com a diferença que o nível protegido permite o acesso aos membros das extensões da classe [2], enquanto o nível privado não fornece tal permissão [3].

Java é uma linguagem fortemente tipada, sendo desta forma realizada uma verificação em tempo de compilação para evitar problemas potenciais de incompatibilidade de tipos.

[2] Extensões de funcionalidades através do mecanismo de herança de classes, segundo os princípios de orientação a objetos.

Todas as referências para métodos e variáveis são checadas para assegurar que os objetos são do tipo apropriado, prevenindo que seja forjado o uso de um objeto se passando por outro tipo incompatível na tentativa de se contornar os controles de acesso citados anteriormente. O compilador não permite o uso de variáveis locais sem sua devida inicialização, evitando assim que sejam utilizadas referências a posições indeterminadas da memória. A proteção atingida por estes mecanismos também é verificada nos smart cards.

Uma vez que a classe Security Manager não é aceita no Java Card, as políticas de segurança da linguagem são implementadas pela própria máquina virtual. A JCVM verifica cada bytecode toda vez que ele é executado [11], assegurando desta forma que o código é bem formado, ou seja, não causa overflow ou underflow da pilha ou contém *bytecodes* ilegais, entre outros possíveis problemas. Outra funcionalidade da JCVM é implementar o mecanismo de *sandbox*, que impõe um conjunto de restrições nas ações que os *applets* podem realizar. As principais restrições estão no mecanismo usado para o compartilhamento de recursos entre os *applets*, que passa por uma série de verificações, e no isolamento dos dados relativos às aplicações, sendo assegurado que uma porção de memória usada por um aplicativo não será acessada por outro aplicativo. Estas restrições fazem com que o mecanismo também seja conhecido por applet firewall.

3.5.1 Funcionalidades de Segurança

As funcionalidades de segurança previstas nos smart cards em geral englobam a verificação da identidade de seu portador, autenticação mútua do cartão e o CAD, o ciframento e deciframento de dados, a geração e verificação de assinaturas digitais. O suporte às duas últimas funcionalidades (ciframento e geração de assinaturas digitais) é especialmente usado por aplicações que envolvem transações monetárias por meios eletrônicos, que possuem o requisito de integridade, confidencialidade e não-repúdio das informações. Uma visão mais detalhada de cada funcionalidade é dada a seguir.

Verificação da Identidade do Portador do Cartão

O acesso aos dados e as funcionalidades do cartão só é permitido ao seu portador legítmo. Os smart cards em geral empregam um esquema de autenticação do portador através de um PIN (*Personal Identification Number*), que só é conhecido pelo dono do cartão. Outros esquemas também podem ser usados, como os de biometria, onde fica armazenado no cartão alguma característica física única do portador do cartão, como por exemplo sua impressão digital, e esta é verificada antes do acesso ser permitido. Este tipo de verificação não é muito usual pois requer um CAD capaz de realizar a leitura da característica física usada pelo sistema de autenticação.

Autenticação

O primeiro passo de toda sessão envolvendo o smart card e o CAD é a autenticação. Em geral, é empregado um protocolo de desafio e resposta (*challenge and response*) usando um algoritmo criptográfico disponível no cartão, como o DES ou o RSA, previstos na API do Java Card. Para autenticar o cartão perante o CAD, o sistema externo gera um número aleatório e o envia para o smart card, que usa sua chave secreta para criptografar o número recebido e envia o resultado de volta ao sistema externo, que decifra a mensagem e verifica se foi recebido o mesmo número. Um procedimento similar é usado para autenticar o sistema externo perante o smart card.

Integridade dos Dados

Muitas situações demandam o serviço de integridade dos dados, entre elas, as transações monetárias por meios eletrônicos e comunicações de e-mail. A integridade consiste na garantia de que o conteúdo de uma mensagem não foi alterado durante sua transmissão por um meio eletrônico. Uma forma de se obter esta funcionalidade é usar um MAC (*Message Authentication Code*). O MAC é um checksum criptográfico, calculado com base na chave secreta de criptografia, a mensagem e um número aleatório, que é anexado ao final da mensagem durante sua transmissão. O recepiente da mensagem recebe, além da mesma, o MAC e o número aleatório usado. Ele então usa a mesma chave secreta de

criptografia e o número aleatório recebido para gerar novamente o MAC da mensagem, que é comparado com o valor recebido. Caso os dois valores de MAC sejam idênticos, a mensagem não foi alterada durante a transmissão.

Uma vez que nesse esquema o recepiente da mensagem tem os recursos necessários para a geração de um MAC válido (recurso usado na verificação do mesmo), não é obtido o não-repúdio do envio das informações. Este recurso é alcançado por meio de outra técnica de autenticação mais sofisticada, o uso de assinaturas digitais, que será discutido posteriormente.

Confidencialidade dos Dados

Quando é necessária a transmissão de informações por uma rede insegura, é importante que dados confidenciais, tais como números de cartões de crédito ou senhas sejam protegidas por algum esquema para não serem vistos por pessoas não autorizadas. Para assegurar a confidencialidade dos dados, o Java Card oferece alternativas de ciframento das mensagens enviadas. Na API padrão, há um pacote de criptografia que contém algoritmos criptográficos simétricos (DES, 3DES) e algorítmos assimétricos (RSA).

Assinaturas Digitais

As assinaturas digitais são capazes de atender aos requerimentos de integridade, autenticidade e não-repúdio das aplicações. Para gerar a assinatura digital de uma mensagem M, primeiramente caucula-se o *hashing* criptográfico $h(M)$ de M. Em seguida, um procedimento envolvendo a chave privada do assinante e $h(M)$ é executado, gerando a assinatura s. Ao receber o par (M, s), um usuário qualquer verifica a validade da assinatura s executando um procedimento envolvendo M, s e a chave pública do assinante. Em alguns casos, como no RSA, a geração e verificação da assinatura são processos idênticos aos de deciframento e ciframento, respectivamente. Em outros, como no caso do DSA (*Digital Signature Algorithm*), são processos distintos e específicos para esse fim. Dada a importância do uso de assinaturas digitais em aplicações envolvendo smart cards, a especificação da API do Java Card inclui o suporte aos esquemas de assinaturas digitais baseados em RSA e em DSA.

Capítulo 4

Criptografia de Curvas Elípticas

A idéia de criptografia de chave pública foi proposta por Whitfield Diffie e Martin Hellman em 1976 com a especificação de um protocolo para o estabelecimento de chave secreta através de um meio de transmissão sem proteção contra escutas [14]. Eles lançaram, também, a idéia de que as chaves de ciframento e deciframento poderiam ser diferentes, a primeira pública e a segunda secreta. Um ano depois, as idéias revolucionárias de Diffie e Hellman foram postas em prática com o surgimento do primeiro algoritmo criptográfico de chaves públicas, o RSA [48], cujo nome é formado pelas iniciais de seus autores, Rivest, Shamir e Adleman.

Desde então, diversos sistemas criptográficos de chave pública foram propostos, cada um deles tendo sua segurança baseada na dificuldade da resolução de um problema matemático bem conhecido. Apesar de ainda não ter sido apresentada uma prova formal da intratabilidade de qualquer desses problemas, as inúmeras tentativas de resolução sem sucesso, feitas por pesquisadores renomados e grandes empresas durante vários anos, servem como forte evidência da inexistência de soluções eficientes que venham a inviabilizar a utilização dos algoritmos. O problema da fatoração de números inteiros muito grandes, que já vinha sendo estudado há vários séculos e de forma mais intensa nos últimos vinte anos, é um destes problemas, sendo a base do RSA.

Um problema matemático que tem se mostrado interessante para a utilização em criptografia é o do logaritmo discreto sobre determinados grupos cíclicos. Taher ElGamal [16]

foi quem primeiro descreveu como este problema poderia ser empregado em esquemas de chave pública e na construção de assinaturas digitais. Sua idéia é a base para o conhecido algoritmo de assinaturas digitais do governo dos Estados Unidos, o DSA (*Digital Signature Algorithm*), que se vale do grupo multiplicativo $\mathbb{Z}_p^*$, sendo p um número primo muito grande.

Uma classe de grupos algébricos cíclicos onde o problema do logaritmo discreto se mostrou difícil é o grupo aditivo de pontos sobre uma curva elíptica definida sobre um corpo finito [26]. Apesar de serem estruturas matemáticas que vem sendo estudadas há mais de 150 anos, somente em 1985 Neal Koblitz [32] e Victor Miller [39], de forma independente, propuseram a utilização de curvas elípticas para a construção de sistemas criptográficos de chave pública.

Desde então, diversos pesquisadores têm trabalhado intensamente no aprimoramento de técnicas para tornar estes sistemas mais seguros e sua implementação mais eficiente. Atualmente, a utilização de curvas elípticas oferece sistemas criptográficos de chave pública rápidos, seguros e que requerem poucos recursos, sendo uma boa opção para dispositivos limitados, tais como smart cards, handhelds e telefones celulares.

Neste capítulo são abordados os principais conceitos da criptografia de curvas elípticas. Na Seção 4.1 são apresentados os conceitos matemáticos básicos para a construção dos sistemas criptográficos de curvas elípticas, com uma descrição dos corpos finitos recomendados pelos padrões internacionais. A Seção 4.2 contempla a definição das curvas elípticas e da operação necessária para a constituição do grupo usado em criptografia. Por fim, na Seção 4.3.2, é mostrado como o grupo de pontos sobre a curva elíptica foi usado para a construção de sistemas criptográficos, com a descrição dos principais protocolos.

4.1 Fundamentos Matemáticos

São apresentados, nesta seção, alguns dos fundamentos matemáticos mais importantes que servem como base para os sistemas criptográficos baseados em curvas elípticas. Na Seção 4.1.1 são introduzidos os conceitos de corpos e grupos, sendo então nas seções 4.1.2 e 4.1.3 apresentados os dois tipos de corpos finitos comumente empregados em algorimos

criptográficos. Em seguida, é apresentada a definição do grupo cíclico dos pontos sobre a curva elíptica e suas principais características.

4.1.1 Grupos e Corpos finitos

Definição 4.1 *Um* grupo $(G, \diamond)$ *é um sistema algébrico formado por um conjunto* G *e uma operação binária* $\diamond : G \times G \mapsto G$ *com as seguintes propriedades:*

1. fecho*: para todos* $P, Q \in G$*, temos que* $P \diamond Q \in G$*;*
2. associatividade*: para todos* $P, Q, R \in G$*, temos que* $(P \diamond Q) \diamond R = P \diamond (Q \diamond R)$*;*
3. existência de identidade*: existe um elemento* $I \in G$ *tal que, para todo* $P \in G$*, temos que* $P \diamond I = I \diamond P = P$*;*
4. existência de inversos*: para todo elemento* $P \in G$*, existe um elemento* $\overline{P} \in G$ *tal que* $P \diamond \overline{P} = \overline{P} \diamond P = I$*.*

Uma representação comum para a operação $\diamond$ é o símbolo $+$ (notação aditiva) ou o símbolo $\times$ (notação multiplicativa). Na primeira, o inverso de um elemento $P \in G$ é escrito como $-P$, tendo como identidade o elemento 0; na segunda, o inverso de $P \in G$ é denotado por P^{-1} e a identidade é representada por 1.

Definição 4.2 *Um determinado grupo* $(G, \diamond)$ *é chamado de* abeliano, *ou* comutativo, *caso sua operação binária* $\diamond$ *possua a seguinte propriedade adicional:*

5. comutatividade*: para todos elementos* $P, Q \in G$*, temos que* $P \diamond Q = Q \diamond P$*.*

Daqui em diante será usado o termo "grupo" significando grupo abeliano.

Definição 4.3 *O número de elementos de um grupo é chamado de* ordem *do grupo.*

Uma vez que é possível adicionar um elemento P de um grupo G a si mesmo um número arbitrário de vezes, a partir da notação aditiva do grupo é possível definir a *multiplicação escalar* como sendo $k \cdot P = P + P + P + \cdots + P$ (k termos), onde k é um

número inteiro não negativo. Por extensão, define-se também $0 \cdot P = 0$ e, para coeficientes negativos, temos $(-k) \cdot P = k \cdot (-P)$. Já na notação multiplicativa, a mesma operação é chamada de *exponenciação*, sendo representada por $P^k = P \times P \times \cdots \times P$ (k fatores), $P^0 = 1$ e $P^{-k} = (P^{-1})^k$. Dado $Q = kP$ (ou $Q = P^k$), o fator (ou expoente) k é chamado *índice* (ou *logaritmo discreto*) de Q com relação a P.

A multiplicação escalar assim definida satisfaz as seguintes propriedades, semelhantes às de um espaço vetorial, para $P, Q \in G$ e $m, n \in \mathbb{Z}$:

1. identidade: $1 \cdot P = P$;
2. distributividade vetorial: $m \cdot (P + Q) = m \cdot P + m \cdot Q$;
3. distributividade escalar: $(m + n) \cdot P = m \cdot P + n \cdot P$;
4. associatividade: $m \cdot (n \cdot P) = (m \cdot n) \cdot P$;
5. comutatividade: $m \cdot (n \cdot P) = n \cdot (m \cdot P)$.

Para cada elemento P de um grupo finito G, sempre é possível encontrar um inteiro positivo r tal que $r \cdot P = 0$, uma vez que, caso a ordem de G seja igual a t, uma sequência qualquer de $t + 1$ ou mais múltiplos de P necessariamente contém algum valor repetido, ou seja, dois escalares x e y (onde $x < y$) tais que $x \cdot P = y \cdot P$, implicando que $r \cdot P = 0$ para $r = y - x > 0$.

Definição 4.4 *O menor inteiro positivo r para o qual $r \cdot P = 0$ é chamado de* ordem *de P.*

É possível verificar que os coeficientes de uma multiplicação escalar podem ser reduzidos módulo r antes de se calcular efetivamente o produto, uma vez que qualquer inteiro m pode ser escrito como $m = \alpha r + \beta$, onde $0 \leq \beta < r$, e portanto temos $m \cdot P = \alpha \cdot (r \cdot P) + \beta \cdot P = \beta \cdot P$.

Definição 4.5 *Um grupo G é* cíclico *se existe um elemento $P \in G$ tal que qualquer elemento $X \in G$ pode ser escrito como $X = k \cdot P$ (notação aditiva) ou $X = P^k$ (notação multiplicativa), para algum k. Neste caso, P é chamado* gerador *de G.*

Evidentemente, a ordem do gerador de um grupo é sempre igual à ordem do grupo.

Definição 4.6 *Um* corpo finito $(C, +, \times)$ *é um sistema algébrico composto de um conjunto C e duas operações binárias $+$ e $\times$ definidas sobre os elementos de C e que possuem as seguintes propriedades:*

- $(C, +)$ *é um grupo aditivo;*
- $(C \setminus \{0\}, \times)$ *é um grupo multiplicativo;*
- distributividade*: para todos $a, b, c \in C$, temos que:*

$$\begin{aligned} a \times (b + c) &= (a \times b) + (a \times c) \\ (a + b) \times c &= (a \times c) + (b \times c). \end{aligned}$$

Um resultado fundamental da Teoria dos Números diz que existe um corpo finito de ordem q se e somente se q é uma potência de um número primo (veja [37]). Adicionalmente, se q é uma potência de um número primo, então existe somente um corpo de ordem q, salvo o caso de isomorfismos; este corpo é denotado por $\mathbb{F}_q$ ou $GF(q)$. Sendo $q = p^m$, onde p é um número primo e m é um número inteiro positivo, então p é chamado de *característica* e m é chamado de *grau de extensão* de $\mathbb{F}_q$.

Existem também diversas formas de representação dos elementos do corpo, sendo que a escolha correta deste quesito pode levar a implementações mais eficientes que outras, dependendo da arquitetura alvo. Para o uso em ECC, os corpos finitos são divididos em duas categorias, onde a primeira é aquela onde é usado um número primo muito grande ($q = p$). Outra abordagem é a utilização de um número primo pequeno e um expoente ($q = p^m$, com p e m pequenos) de modo que a ordem do corpo fique grande, condição necessária para a construção de sistemas criptográficos. Os dois tipos de corpos são abordados nas próximas seções.

4.1.2 Corpo Finito $\mathbb{F}_p$

Corpos finitos com característica prima $\mathbb{F}_p$, sendo p um número muito grande, são muito populares. Isto ocorre uma vez que podem ser implementados de forma eficiente usando

técnicas herdadas dos corpos finitos usados em sistemas criptográficos já existentes há muito tempo, tais como o RSA e o DSA.

A redução por um módulo muito grande é o principal problema encontrado na implementação desta categoria de corpos finitos. O algoritmo padrão, descrito em [31], utiliza o algoritmo de divisão modular que tem um custo proibitivo para módulos muito grandes. Uma forma encontrada para se obter a redução de forma eficiente é a mudança dos números para uma base onde as operações se tornam mais simples. Esta alternativa foi proposta por Montgomery [41], sendo possível, através da conversão numérica proposta, transformar a operação de redução num simples conjunto de operações de deslocamentos de bits. Já a redução de Barret [7], que possui um tempo comparável à de Montgomery [54], também é uma técnica muito atraente quando o módulo dos cálculos é fixo, sendo empregada uma tabela de valores pré-calculados. Uma discussão mais detalhada sobre os dois métodos e sua implementação pode ser encontrada em [8, 38].

Outra abordagem é a utilização de números primos com propriedades que permitem reduzir significativamente o custo (em termos de tempo) das operações modulares. Os números primos de forma mais simples são conhecidos por primos de Mersenne, que são da forma $p = 2^k - 1$. Entretanto, o número de primos de Mersenne com tamanho adequado para criptografia é limitado, levando diversos autores a proporem generalizações destes números primos. Uma das propostas, feita por Crandall [13], faz uso de primos na forma $p = 2^k - c$, onde c é um número inteiro pequeno, que usualmente pode ser representado em uma única palavra de computador. Os números primos na forma $p = 2^k \pm c$, para um valor pequeno de c (em relação a 2^m), são frequentemente chamados de primos pseudo-Mersenne.

Em outra direção, Solinas [55] introduziu o conceito de primos de Mersenne generalizados, que são dados pela forma:

$$p = f(2^k)$$

onde f é um polinômio de grau pequeno e k é um múltiplo do tamanho de uma palavra de computador. Em sua proposta, Solinas apresenta um algoritmo para a realização da redução através de três adições módulo p, que são operações menos custosas que a divisão modular.

O uso dos primos de Mersenne generalizados se tornou muito popular devido a adoção destes números nas curvas recomendadas em padrões de diversas instituições, tais como a ANSI [2], NIST [43], SECG [52] e WAP [64].

4.1.3 Corpo Finito $\mathbb{F}_{p^m}$

Os corpos finitos desta categoria que têm sido usados em aplicações criptográficas são aqueles onde o valor de p é igual a 2 ou aqueles onde p é um número primo que pode ser representado em uma palavra de computador. A primeira opção é mais antiga, sendo usada em diversos sistemas disponíveis atualmente. Já a segunda foi proposta recentemente e tem se mostrado uma boa escolha. A seguir os dois tipos de corpos finitos são mostrados com mais detalhes.

Corpo Finito Binário ($\mathbb{F}_{2^m}$)

O corpo finito mais comumente usado na implementação de curvas elípticas é o $\mathbb{F}_{2^m}$, chamado de *corpo finito binário*. Este corpo pode ser visto como um espaço vetorial de dimensão m sobre $\mathbb{F}_2$, ou seja, existe um conjunto de m elementos $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ em $\mathbb{F}_{2^m}$ tal que cada elemento $a \in \mathbb{F}_{2^m}$ pode ser escrito unicamente na forma:

$$a = \sum_{i=0}^{m-1} a_i \alpha_i, \text{ onde } a_i \in \{0, 1\}.$$

O conjunto $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ é chamado *base* de $\mathbb{F}_{2^m}$ sobre $\mathbb{F}_2$, sendo o elemento a representado pelo vetor binário $(a_0, a_1, \ldots, a_{m-1})$. Para o uso em sistemas criptográficos, os corpos finitos de característica 2 são representados por bases polinomiais, que são usadas em implementações em software, ou por bases normais, mais adequadas para a implementação em hardware. Uma vez que este trabalho tem como objetivo a implementação em software de sistemas criptográficos baseados em curvas elípticas, uma discussão detalhada sobre bases normais está fora do escopo deste texto. Uma discussão detalhada do uso de bases normais pode ser encontrada em [4, 42].

Bases polinomiais. Seja $f(x) = x^m + \sum_{i=0}^{m-1} f_i x^i$, onde $f_i \in \{0, 1\}$, para $i = 0, 1, \ldots, m-1$, um polinômio irredutível de grau m sobre $\mathbb{F}_2$; $f(x)$ é chamado de *polinômio redutor*.

Para cada polinômio redutor, existe uma representação em base polinomial [36]. Nesta representação, cada elemento de $\mathbb{F}_{2^m}$ corresponde a um polinômio binário de grau menor que m.

O uso de bases polinomiais possui a vantagem de sua aritmética ser constituída de operações sem *carry*, recurso que auxilia as implementações em linguagens de alto nível. Também diversas de suas operações podem ser feitas com instruções básicas disponíveis na maioria das arquiteturas, tais como operações de ou-exclusivo ou de deslocamento de bits.

Corpos de Extensão Ótima (OEF)

Os Corpos de Extensão Ótima foram propostos por Bailey e Paar em [5] e [6] e possuem características que permitem a construção de aritméticas eficientes. Para ser classificado por OEF, um corpo deve estar na forma:

- $p = 2^k \pm c$ é um primo pseudo-Mersenne, com $\log_2 c \leq k/2$ e p podendo ser armazenado em uma palavra de computador.

- $f(x) = x^n - \omega$ é irredutível.

Por questões de eficiência, em geral assume-se $\omega = 2$.

Em corpos OEF, a adição de elementos em $\mathbb{F}_q$ é relativamente simples e pode ser feita sem propagação de *carries*, uma vez que os elementos de $\mathbb{F}_q$ são implementados como polinômios módulo $f(x)$. A operação de multiplicação também apresenta vantagens, principalmente devido ao fato do polinômio irredutível permitir a redução de forma mais simplificada. Uma vez que o corpo, ao qual os coeficientes do polinômio pertencem, é definido sobre um número primo pequeno o suficiente para poder ser usada uma palavra de computador para armazená-lo, as operações sobre os coeficientes e a redução são feitas por instruções nativas da máquina, o que garante um bom desempenho.

4.2 Curvas elípticas sobre corpos finitos

Seja F um corpo. Como exemplo, F pode ser o corpo finito $\mathbb{F}_q$, o corpo primo $\mathbb{Z}_p$, onde p é um primo grande, o corpo dos números reais $\mathbb{R}$, o corpo dos números racionais $\mathbb{Q}$ ou o corpo dos números complexos $\mathbb{C}$.

Definição 4.7 *Uma curva elíptica E, definida sobre um corpo F, é um conjunto de pontos (x, y), com $x, y \in F$, que satisfazem a equação de Weierstrass:*

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

onde $a, b, c, d, e \in F$.

A curva elíptica E definida sobre F é denotada por $E(F)$. O número de pontos de E (a cardinalidade) é denotada por $\#E(K)$, ou somente $\#E$.

Para corpos de várias características, a equação de Weierstrass pode ser transformada (e simplificada) em diferentes formas por uma mudança linear das variáveis. Primeiramente, são apresentadas, aqui, as equações para corpos de características diferentes de 2 e 3, e posteriosmente as equações para os corpos com característica 2. A equação para os corpos de característica 3 foi omitida pois estes corpos não têm aplicação definida em criptografia.

[**Corpos de característica $\neq 2, 3$**] Seja F um corpo finito de característica $\neq 2, 3$. Uma curva elíptica E sobre F é definida pela equação

$$y^2 = x^3 + ax + b,$$

onde $a, b \in F$ e $4a^3 + 27b^2 \neq 0$ (condição que assegura que o polinômio não possui raízes múltiplas). O conjunto $E(F)$ consiste de todos os pontos (x, y), com $x, y \in F$, que satisfazem a equação que define a curva, junto a um ponto $\mathcal{O}$, chamado de *ponto no infinito.*

[**Corpos de característica 2**] Seja F um corpo finito de característica 2. Uma curva elíptica E sobre F é definida pela equação equação

$$y^2 + xy = x^3 + ax^2 + b,$$

onde $a, b \in F$ e $b \neq 0$. O conjunto $E(F)$ consiste de todos os pontos (x, y), com $x, y \in F$, que satisfazem a equação que define a curva, junto com um ponto $\mathcal{O}$, chamado de *ponto no infinito*.

4.2.1 Definição do grupo aditivo

Para uma curva elíptica E qualquer definida sobre um corpo F, é possível definir uma operação de adição entre os pontos da curva de modo a ser formado um grupo abeliano, como é mostrado a seguir.

Dados dois pontos $P, Q \in E(F)$, é definido um terceiro ponto, $P + Q$, segundo as regras abaixo:

1. Se $P \neq Q$, então a interseção da linha que conecta P e Q com $E(F)$ é dada em único ponto, que será denotado por PQ.

2. Se $P = Q$, então a interseção da tangente de $E(K)$ em P com a curva será o ponto PQ.

Seria tentador usar o ponto PQ como resultado da soma entre P e Q, porém isto resultaria numa operação sem a definição de um elemento neutro, o que iria ferir a definição de um grupo. Um meio de assegurar a existência do elemento neutro é utilizar o inverso do ponto PQ como resultado da operação. Na Figura 4.1 é ilustrado este procedimento, chamado de *método da corda e da tangente*, tomando a curva $y^2 = x^3 - x$ definida sobre os números reais ($\mathbb{R}$).

De acordo com a característica do corpo sobre o qual a curva elíptica está definida, a soma é definida de forma diferente. No caso de corpos de característica $\neq 2, 3$, o inverso de um ponto $P = (x_1, y_1) \in E$ é $-P = (x_1, -y_1)$. Assim, se $Q \neq -P$, então $P + Q = (x_3, y_3)$, sendo

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

onde o valor de λ é dado por:

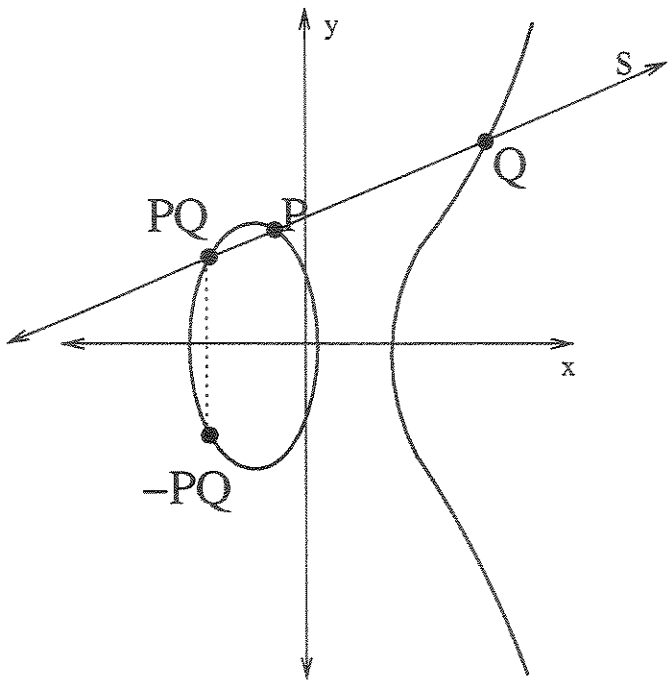


Figura 4.1: Soma elíptica de dois pontos distintos P e Q.

Se $P \neq Q$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

Se $P = Q$

$$\lambda = \frac{3x_1^2 + a}{2y_1}$$

Já para os corpos de característica 2, o inverso de um ponto $P = (x_1, y_1) \in E$ é dado por $-P = (x_1, y_1 + x_1)$. Assim, se $Q \neq -P$, então $P + Q = (x_3, y_3)$, sendo:

Se $P \neq Q$

$$\begin{aligned} x_3 &= \left(\tfrac{y_1+y_2}{x_1+x_2}\right)^2 + \left(\tfrac{y_1+y_2}{x_1+x_2}\right) + x_1 + x_2 + a \\ y_3 &= \left(\tfrac{y_1+y_2}{x_1+x_2}\right)(x_1 + x_3) + x_3 + y_1 \end{aligned}$$

Se $P = Q$

$$\begin{aligned} x_3 &= \left(\tfrac{b}{x_1^2}\right) + x_1^2 \\ y_3 &= x_1^2 + \left(x_1 + \tfrac{y_1}{x_1}\right) x_3 + x_3 \end{aligned}$$

O ponto no infinito ($\mathcal{O}$) é o elemento neutro da operação de adição definida acima, ou seja, as seguintes propriedades são verificadas:

1. $P + \mathcal{O} = \mathcal{O} + P = P$.
2. $P - P = \mathcal{O}$.

A partir destas propriedades, é possível provar que os pontos sobre uma curva elíptica junto com o ponto $\mathcal{O}$ e a operação de adição formam um grupo abeliano tendo $\mathcal{O}$ como identidade aditiva, como descrito em [53].

4.3 Criptossistemas de curvas elípticas

A operação de adição em ECC é usada para formar um grupo sobre o qual é possível construir um sistema criptográfico, de forma semelhante à operação de multiplicação modular relativa ao RSA. Assim, a aplicação da adição de um elemento sobre ele mesmo múltiplas vezes é o equivalente a operação de exponenciação. Para formar um sistema criptográfico usando curvas elípticas, é necessário encontrar um "problema difícil" correspondente a fatorar o produto de dois primos ou obter o logaritmo discreto. Na Seção 4.3.1 são apresentados os parâmetros usados na definição de sistemas criptográficos baseados em curvas elípticas e na Seção 4.3.2 são mostrados os protocolos criptográficos comumente usados com esta tecnologia criptográfica.

4.3.1 Parâmetros de ECC

As operações de esquemas criptográficos de chave pública envolvem operações aritméticas onde uma curva elíptica é determinada por alguns parâmetros. Estes parâmetros são dados pela sétupla:

$$T = (q, FR, a, b, G, n, h)$$

consistindo do número q que define a potência prima ($q = p^m$), uma indicação FR do método usado para representar os elementos do corpo, os coeficientes da equação da curva $a, b \in \mathbb{F}_q$ ($y^2 = x^3 + ax + b$ para corpos de característica $\neq 2, 3$ ou $y^2 + xy = x^3 + ax^2 + b$

para corpos de característica 2), o ponto base $G = (x_G, y_G) \in E(\mathbb{F}_q)$, um primo n que é a ordem do elemento G e um inteiro h que é o cofator $h = \#E(\mathbb{F}_q)/n$.

Vários algoritmos para a geração e validação de parâmetros de curvas elípticas foram propostos, tais como os apresentados em [52] e [43]. Uma vez que o principal parâmetro de segurança é n, o tamanho da chave de ECC é definido como sendo o tamanho em bits de n. Na Tabela 4.1 está a comparação do tamanho de chaves de diferentes sistemas criptográficos com um nível de segurança comparável (sendo considerados os ataques conhecidos).

Tamanho de chave cifrador simétrico	Algoritmo Exemplo	Tamanho de chave ECC	Tamanho de chave DSA/RSA
80	SKIPJACK	160	1024
112	Triple-DES	224	2048
128	128-bit AES	256	3072
192	192-bit AES	384	7068
256	256-bit AES	512	15360

Tabela 4.1: Comparativo de tamanhos de chave entre sistemas simétricos, ECC, DSA e RSA (Fonte: Certicom Corporation).

4.3.2 Protocolos de Curvas Elípticas

Nesta seção é dada uma breve descrição de três protocolos fundamentais baseados em curvas elípticas: o ECDH (*Elliptic Curve Diffie-Hellman*), o ECDSA (*Elliptic Curve Digital Signature Algorithm*) e o ECAES (*Elliptic Curve Authenticated Encryption Scheme*). O ECDH é a versão elíptica do bem conhecido método de estabelecimento de chaves de Diffie-Hellman; o ECDSA é uma extensão do DSA com o uso de curvas elípticas, que foi proposto por Scott Vanstone em 1992 [63]. Já o ECAES é uma variante do esquema de ciframento de chave pública de ElGamal, proposto em 1998 por Abdala, Bellare e Rogaway [1].

Geração das Chaves

O par de chaves (pública e privada) de uma entidade está associado com um conjunto particular de parâmetros de uma curva elíptica (q, FR, a, b, G, n, h). Para assegurar a integridade do sistema, ou seja, que um determinado par de chaves realmente está associado a um conjunto de parâmetros, é possível utilizar certificados digitais ou garantir que todas as entidades utilizem os mesmos parâmetros, ficando desta forma a associação estabelecida pelo contexto.

A geração do par de chaves criptográficas de A deve ser feita da seguinte maneira:

1. Obtenha um número aleatório d no intervalo $[1, n-1]$.
2. Calcule $Q = dG$.
3. A chave pública de A é Q, a chave secreta é d.

Validação da Chave Pública

Este processo assegura que uma chave pública satisfaz os requisitos aritméticos de criptografia baseada em curvas elípticas (veja em [51]). Uma chave pública $Q = (x_Q, y_Q)$ associada com os parâmetros (q, FR, a, b, G, n, h) é validada através do seguinte procedimento (chamado de validação explícita):

1. Verifique se $Q \neq \mathcal{O}$.
2. Verifique se x_Q e y_Q são elementos de $\mathbb{F}_q$.
3. Verifique se Q pertence à curva elíptica definida por a e b.
4. Verifique se $nQ = \mathcal{O}$.

A validação dos parâmetros públicos sem o passo 4 é chamada de validação *parcial* de chave pública.

ECDH

A idéia básica do protocolo ECDH é gerar um valor secreto compartilhado por duas partes a partir de uma chave secreta pertencente a uma entidade A e uma chave pública pertencente a uma entidade B. Assim, caso ambas as partes executem o procedimento simultaneamente com suas respectivas chaves, elas irão recuperar o mesmo valor secreto. Será assumido que a entidade A possui os parâmetros da curva dados por $D = (q, FR, a, b, G, n, h)$ e uma chave privada d_A. Também será assumido que a entidade B possui uma chave pública Q_B associada com D, que deve ser pelo menos *parcialmente* válida.

A entidade A deve então seguir o seguinte procedimento para calcular o valor secreto com B:

1. Calcule $P = d_A Q_B = (x_P, y_P)$.
2. Verifique se $P \neq \mathcal{O}$.
3. O valor secreto é $z = x_P$.

Caso o passo 1 seja calculado como $P = hd_A Q_B = (x_P, y_P)$, então este procedimento é chamado de *Diffie-Hellman baseado em curvas elípticas com cofator*. A incorporação do cofator h no cálculo do valor secreto fornece uma resistência eficiente a ataques tal como os que aproveitam subgrupos pequenos (veja em [51]).

ECAES

Este é um protocolo de ciframento e deciframento para o envio de mensagens de A para B. Suporemos que B possui os parâmetros $D = (q, FR, a, b, G, n, h)$ e a chave pública Q_B. Suporemos também que A possui cópias autênticas de D e Q_B. A seguir, o MAC denota um algoritmo de código de autenticação de mensagens, tal como o HMAC [34], ENC denota um sistema de ciframento simétrico, tal como o Triple-DES, e KDF é uma função de derivação de chaves que deriva chaves criptográficas a partir de um ponto secreto.

Para cifrar uma mensagem m para B, A deve realizar os seguintes passos:

1. Obter um número inteiro aleatório r que esteja no intervalo $[1, n-1]$.

2. Calcular $R = rG$.

3. Calcular $K = hrQ_B = (x_K, y_K)$. Verificar se $K \neq \mathcal{O}$.

4. Calcular $k_1||k_2 = \mathrm{KDF}(x_k)$.

5. Calcular $c = \mathrm{ENC}_{k_1}(m)$.

6. Calcular $t = \mathrm{MAC}_{k_2}(c)$.

7. Enviar (R, c, t) para B.

Para decifrar o texto (R, c, t), B deve fazer:

8. Realizar a validação parcial de R.

9. Calcular $K = hd_BR = (x_K, y_K)$. Verificar se $K \neq \mathcal{O}$.

10. Calcular $k_1||k_2 = \mathrm{KDF}(x_K)$.

11. Verificar se $t = \mathrm{MAC}_{k_2}(c)$.

12. Calcular $m = \mathrm{ENC}_{k_1}^{-1}(c)$.

As operações que requerem mais recursos no processo de cifrамento e decifrамento são as multiplicações escalares nos passos 3 e 9.

ECDSA

O algoritmo é usado para assinar digitalmente as mensagens enviadas de uma entidade A para uma entidade B. Suponha que A possua os parâmetros $D = (q, FR, a, b, G, n, h)$ e a chave pública Q_A. Também supomos que B possui cópias autênticas de D e Q_A. A seguir, o SHA-1 denota uma função de hash de 160 bits [44].

Para assinar uma mensagem m, A deve realizar os seguintes passos:

1. Obter um número inteiro aleatório r que esteja no intervalo $[1, n-1]$.

2. Calcular $kG = (x_1, y_1)$ e $r = x_1 \bmod n$.
   Caso $r = 0$, volte ao passo 1.

3. Calcular $k^{-1} \bmod n$.

4. Calcular $e = \text{SHA-1}(m)$.

5. Calcular $s = k^{-1}\{e + d_A \cdot r\} \bmod n$.
   Caso $s = 0$, volte ao passo 1.

6. A assinatura de A para a mensagem m é (r, s).

Para verificar a assinatura digital (r, s) de A na mensagem m, B deve realizar os seguintes passos:

7. Verificar se r e s são inteiros no intervalo $[1, n-1]$.

8. Calcular $e = \text{SHA-1}(m)$.

9. Calcular $w = s^{-1} \bmod n$.

10. Calcular $u_1 = ew \bmod n$ e $u_2 = rw \bmod n$.

11. Calcular $u_1 G + u_2 Q_A = (x_1, y_1)$.

12. Calcular $v = x_1 \bmod n$.

13. A assinatura está correta se e somente se $v = r$.

As operações que mais consomem tempo nos processos de geração e verificação de assinaturas são as multiplicações escalares feitas nos passos 2 e 11.

Capítulo 5

Implementação

Os smart cards são dispositivos que têm diversas restrições de recursos, principalmente com relação ao espaço de armazenamento e ao poder de processamento. Como grande parte das aplicações que fazem uso de smart cards necessitam de serviços criptográficos, faz-se necessário que sejam disponibilizadas boas implementações dos mesmos de modo que os requisitos de segurança e de desempenho sejam atendidos.

Na API do Java Card, o algoritmo criptográfico de chave assimétrica usado é o RSA [56, 58]. Uma vez que o RSA requer a realização de operações modulares com números inteiros muito grandes, em especial a exponenciação modular, torna-se necessário o uso de co-processadores capazes de realizar tais operações, tornando assim o custo unitário dos cartões significativamente elevado, conforme mostrado na Seção 2.2.3.

Uma alternativa ao uso do RSA que tem sido explorada são os criptossistemas baseados em curvas elípticas. Estes criptossistemas tem apresentado diversas vantagens em relação ao RSA, entre elas o uso de chaves criptográficas menores e a necessidade de menor poder de processamento, considerando-se níveis de segurança equivalentes [35]. Desta forma, o uso de tal técnica criptográfica é freqüentemente citada como a mais adequada, na atualidade, para o uso em dispositivos com recursos limitados, como é o caso dos smart cards [10, 65].

Assim, o foco deste trabalho foi voltado à implementação de algoritmos criptográficos baseados em curvas elípticas no Java Card, uma vez que esta técnica pode significar

uma redução no espaço necessário para o armazenamento de chaves criptográficas e pode remover a necessidade da utilização de um co-processador criptográfico no chip. Uma vez que a aritmética em curvas elípticas está baseada na aritmética do corpo sobre o qual é definida, daremos um enfoque maior nesta parte, com um estudo mais aprofundado destes algoritmos. Assim, na Seção 5.1 são descritas as escolhas com relação à base do sistema criptográfico, discutindo desde o tipo de corpo a ser usado até a forma de representação dos elementos do corpo e os melhores algoritmos a serem usados em cada caso.

Já na Seção 5.2 é abordada a implementação da operação de adição dos pontos sobre a curva elíptica, que define o grupo aditivo usado em operações criptográficas. Além da operação do grupo, nessa seção, são descritas as extensões da adição, tais como a multiplicação escalar. Por fim, na Seção 5.3 descreveremos a construção de um sistema criptográfico baseado em curvas elípticas usando operações discutidas.

5.1 Construção da Aritmética do Corpo Finito

O desempenho da aritmética sobre o corpo finito é um dos principais fatores no sistema como um todo, uma vez que para cada operação no nível superior (aritmética no grupo de pontos da curva elíptica), em geral, são necessárias várias operações da aritmética básica. Os padrões de criptografia têm recomendado o uso de corpos finitos de característica ímpar ($\mathbb{F}_p$, onde p é um número primo grande) e de corpos finitos de característica par ($\mathbb{F}_{2^m}$). Outra alternativa que tem obtido destaque é a utilização de corpos de extensão ótima, com melhor performance em relação aos outros dois.

A aritmética sobre $\mathbb{F}_p$ já é alvo de estudos há muito tempo, principalmente devido à proximidade de sua natureza de suas operações com a aritmética moldular. Assim, o uso deste corpo permite que sejam aproveitados todos os estudos que vêm sendo feitos por mais de 20 anos e toda a abrangência de algoritmos disponíveis, bem como a grande quantidade de co-processadores criptográficos com capacidade de realizar operações modulares sobre números muito grandes. A vantagem do uso de curvas elípticas neste caso está no fato de ser possível a utilização de elementos do corpo de tamanhos menores, ou seja, representados por um número menor de bits, para um nível equivalente de segurança.

Com isto, os resultados gerados pelos protocolos criptográficos, em especial as assinaturas digitais, têm tamanhos menores e conseqüentemente há uma economia de espaço de armazenamento. Outra conseqüência da diminuição do tamanho das assinaturas está na redução do tempo de transmissão de dados entre o cartão e o CAD, um dos principais gargalos de aplicações para smart cards.

No entanto, o uso deste corpo apresenta uma desvantagem na implementação em linguagens de alto nível. Uma vez que grande parte das operações aritméticas sobre este corpo necessita que o *carry bit* seja acessado, as implementações que não fazem uso do *assembly* do processador devem simular ou prever o *carry*, gerando um *overhead* considerável de processamento.

Num trabalho recente [17], foi feita uma implementação em Java Card de uma aritmética modular sem que bons resultados tivessem sido atingidos. Apesar do autor do trabalho citar que sua implementação não estivesse totalmente otimizada, fizemos uma análise das possíveis melhorias nesta abordagem e concluimos que a redução no tempo de processamento não iria atingir a ordem de grandeza suficiente para a implementação poder ser usada de forma prática em sistemas criptográficos. Apesar de ter sido encontrado na literatura um trabalho reportando a implementação desta aritmética para um microprocessador comumente usado em smart cards [25], vale ressaltar que não era uma plataforma de Java Card e que foi usada a linguagem *assembly* do processador, possibilitando o acesso direto ao *carry bit*, o que não se aplica em nosso caso.

Tendo em vista os fatos apresentados acima, decidimos então construir somente a aritmética para os corpos $\mathbb{F}_{2^m}$ e de extensão ótima usando bases polinomiais, por apresentarem uma aritmética sem *carry* e por serem encontrados na literatura diversos trabalhos mostrando que sua implementação em software tem apresentado bons resultados em dispositivos com poucos recursos. Na Seção 5.1.1 será discutida uma importante decisão de projeto a ser tomada, que é a de se fazer uma implementação genérica ou específica, sendo apresentadas suas respectivas vantagens e desvantagens e uma solução intermediária adotada. A Seção 5.1.2 apresenta a modelagem da solução, tanto do ponto de vista do gerador de código quanto do esqueleto básico do código gerado. As seções seguintes dizem respeito aos algoritmos empregados na duas abordagens, ou seja, a Seção 5.1.3 trata da

implementação sobre $\mathbb{F}_{2^m}$ e a Seção 5.1.4 mostra como foi feita a implementação em OEF.

5.1.1 Gerador de Código

A implementação da aritmética sobre um determinado corpo finito pode ser feita de tal forma que seja possível mudar os parâmetros que definem o corpo e os algoritmos sejam capazes de continuar funcionando corretamente ou então fazer uma implementação específica para um determinado conjunto de parâmetros para o corpo. A primeira abordagem tem como vantagem a grande facilidade em se alterar as características do corpo sem um grande esforço de programação. Em geral, é escolhida a característica do corpo que se deseja trabalhar e a base de representação, onde a parte configurável fica nos parâmetros que definem a base e o número de bits usados nos elementos do corpo.

O ganho com flexibilidade da implementação usando algoritmos genéricos é compensado pela perda de desempenho que poderia ser obtida com a exploração de características peculiares de alguns corpos usados. Ou seja, é possível que sejam escolhidos os parâmetros do corpo e só então feita a implementação, com a adaptação dos algoritmos para que todas as possíveis otimizações baseadas em características únicas do corpo sejam exploradas. Esta abordagem resulta em melhor desempenho, tendo um custo muito elevado caso seja necessário modificar algum dos parâmetros do corpo.

Uma abordagem que visa a união das vantagens das estratégias citadas acima é a da construção de um gerador de código capaz de explorar as características dos parâmetros do corpo e prover boas implementações para os algoritmos para as operações básicas, mas oferecendo a flexibilidade de mudança de corpo através de uma simples execução do gerador com novos parâmetros. A idéia inicial, surgida em [40], apresentou bons resultados para implementação em C++ em máquinas sem limitações de recursos. Nosso trabalho foi a adaptação desta idéia para a construção de um gerador de código com algoritmos escolhidos de forma a atender as limitações de memória e de poder de processamento impostos pelo Java Card.

A construção do gerador usou bases polinomiais. Uma vez que a implementação apresentava um risco elevado devido ao *overhead* imposto pela máquina virtual, decidimos restringir um pouco mais o escopo do problema em relação ao gerador de código original

para que fossem usados somente trinômios irredutíveis como base do corpo finito. Esta decisão levou a uma perda de generalidade no gerador, já que nos padrões de curvas elípticas também está previsto o uso de pentanômios. No entanto, os pentanômios têm se mostrado menos eficientes que os trinômios e este escopo mais restrito permitiu que fosse obtido um maior nível de otimizações.

5.1.2 Modelo de Objetos da Aritmética Básica

Apesar da estratégia de utilizar um gerador de código prover flexibilidade na mudança da aritmética sobre o corpo finito, é importante que a modelagem dos elementos do corpo seja feita de forma que uma mudança tenha o menor impacto possível em camadas superiores que façam uso desta aritmética. Tendo em vista que a linguagem do Java Card é orientada a objetos, foi possível empregar os conceitos básicos de se obter este isolamento entre as camadas da implementação. O uso de interfaces é uma das abordagens que permitem um menor acoplamento entre camadas de software [9], sendo definido somente um contrato geral com o comportamento esperado pelos objetos que implementam a interface. Assim, as operações básicas sobre um elemento de um corpo finito foram agrupadas em uma interface, com o gerador configurado de modo a produzir classes que implementam a interface base.

Um diagrama UML da base dos elementos dos corpos finitos implementados é mostrado na Figura 5.1. Podemos observar primeiramente que na interface foram incluídos, além das operações básicas, métodos que visam uma troca de informações facilitada entre o elemento do corpo e o meio externo através de *arrays* de bytes. Esta abordagem foi escolhida pois verificamos que na API do Java Card esta estratégia está muito presente, tornando assim nossa implementação mais integrada à forma da API trabalhar. Abaixo é apresentado o comportamento esperado de cada um dos métodos da interface básica:

public short byteLength() Retorna o número de bytes que podem ser usados para preencher um elemento do corpo.

public short fromByteArray(byte[] array, short offset, short len) Preenche o elemento do corpo com as informações contidas no vetor de bytes, a partir da

posição indicada por `offset`. O parâmetro `len` indica o número máximo de bytes que pode ser carregado. Retorna a quantidade de bytes que efetivamente foram carregados.

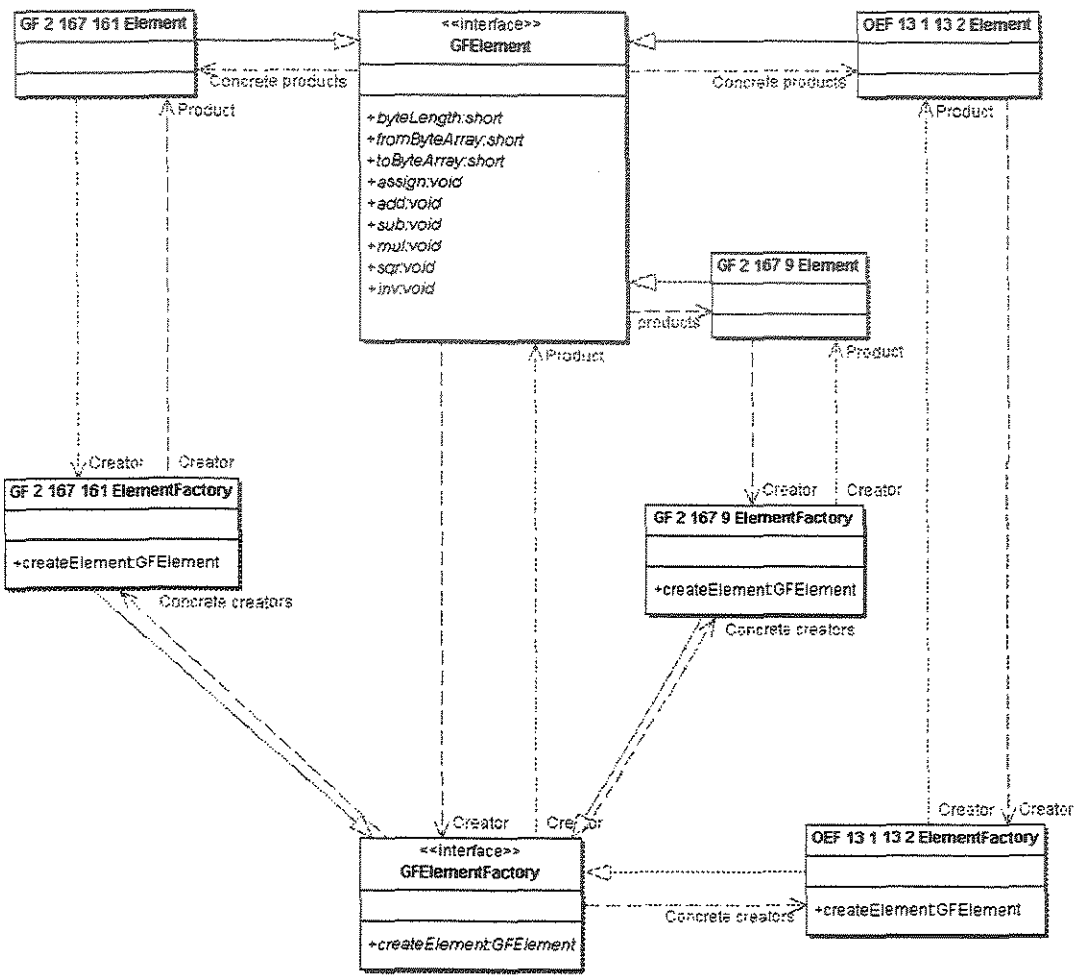


Figura 5.1: Diagrama de classes UML para a aritmética no corpo finito.

public short toByteArray(byte[] array, short offset) Armazena o conteúdo do elemento em um *array* de bytes, começando a partir da posição indicada por `offset`. Retorna o número de bytes usados para armazenar o elemento do corpo.

public void assign(GFElement element) Faz a cópia do conteúdo de outro elemento do corpo finito sobre o elemento representado no objeto. Este método foi incluído principalmente devido à natureza de reutilização de objetos das aplicações desenvolvidas para o Java Card.

public void add(GFElement B, GFElement C)
Define a soma, no corpo finito, do elemento representado no objeto de outro elemento do mesmo corpo **B**. O resultado deve ser armazenado no elemento **C**. Seria a operação equivalente a $C = A + B$, com $A, B, C \in \mathbb{F}_q$ e A representando o elemento contido no objeto.

public void sub(GFElement B, GFElement C)
Define a subtração do elemento representado no objeto por outro elemento do mesmo corpo **B**. O resultado deve ser armazenado no elemento **C**. Seria a operação equivalente a $C = A - B$, com $A, B, C \in \mathbb{F}_q$ e A representando o elemento contido no objeto.

public void mul(GFElement B, GFElement C)
Define a multiplicação no corpo finito do elemento representado no objeto por outro elemento do mesmo corpo **B**. O resultado deve ser armazenado no elemento **C**. Seria a operação equivalente a $C = AB$, com $A, B, C \in \mathbb{F}_q$ e A representando o elemento contido no objeto.

public void sqr(GFElement C)
Define a operação de elevar ao quadrado, no corpo finito, o elemento representado no objeto. O resultado deve ser armazenado no elemento **C**. Seria a operação equivalente a $C = A^2$, com $A, C \in \mathbb{F}_q$ e A representando o elemento contido no objeto.

public void inv(GFElement C)
Define a operação de obtenção de inverso multiplicativo do objeto no corpo finito. O resultado deve ser armazenado no elemento **C**. Seria a operação equivalente

ao cálculo de um elemento $C = A^{-1}$ com a propriedade de que $AC = 1$, com $A, A^{-1}, C \in \mathbb{F}_q$ e A representando o elemento contido no objeto.

Ainda observando a Figura 5.1, temos o exemplo de duas classes baseadas em corpos de característica par, as classes `GF2_167_161_Element` e `GF2_167_9_Element`, que correspondem aos corpos definidos pelos trinômios $x^{167}+x^{161}+1$ e $x^{167}+x^{9}+1$, respectivamente. Já a representante de OEF é dada pela classe `OEF_13_1_13_2` para a qual n é igual a 13, c é igual a -1, m é igual a 13 e w é igual a 2. Associada a cada implementação está sua respectiva fábrica, segundo o padrão de projeto *Factory Method* [20], que visa promover principalmente a facilidade em se modificar a implementação concreta de um contrato estabelecido seja por interface ou por classe abstrata.

Dada a natureza distinta entre os corpos de característica par e os OEF, foi construída uma implementação do gerador de código para cada uma destas abordagens. Nas seções seguintes serão detalhados os algoritmos e otimizações feitas em cada um dos geradores construídos, sendo apontados os pontos mais críticos em cada um dos casos.

5.1.3 Implementação da Aritmética Básica em GF(2^m)

A implementação da aritmética do corpo $\mathbb{F}_{2^m}$ foi feita com o uso de bases polinomiais para a representação dos elementos do corpo, ou seja, os elementos do corpo são polinômios reduzidos por um polinômio irredutível. Para ser atingido um nível maior de otimizações, adotamos a restrição de que só seria permitido o uso de trinômios irredutíveis, com os dois expoentes iniciais servindo de parametrização para o gerador. O terceiro termo, a constante, sempre é não-nula, pois o trinômio é irredutível.

A seguir será mostrado como os elementos do corpo foram armazenados fisicamente no cartão e em seguida serão apresentados os detalhes de implementação das operações definidas na interface base para os elementos do corpo finito.

Representação dos Elementos do Corpo

A representação física dos elementos do corpo foi feita em um *array* de elementos do tipo *short*, que é o maior disponível no cartão. Cada bit do *array* representa um coeficiente do

polinômio, que é um valor em $\mathbb{F}_2$, ou seja, pode assumir apenas o valor 0 ou 1. O expoente associado ao coeficiente é dado pela posição de cada bit dentro do *array* visto como um todo. Portanto, o número de posições usado por um polinômio de grau máximo $m-1$ é dado pela expressão $\lceil m/16 \rceil$, onde 16 é o número de bits usado pelo tipo *short*. Caso em alguma implementação seja adotado algum outro tipo, como por exemplo em cartões que suportem o tipo *int*, tanto o tipo usado como o seu tamanho são configuráveis no gerador, ou seja, o valor 16 deve ser substituído pelo tamanho do tipo usado na constante apropriada do gerador.

A Figura 5.2 apresenta o armazenamento do polinômio

$$x^{14} + x^{12} + x^{11} + x^9 + x^6 + x^4 + x^3 + 1$$

em um *array* composto por elementos do tipo *byte*. Como observado, o número de palavras usado é $\lceil 15/8 \rceil = 2$, sendo o coeficiente de índice 0 armazenado na primeira posição da palavra 0.

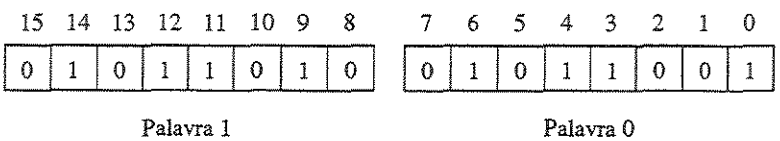


Figura 5.2: Representação polinomial de um elemento de $\mathbb{F}_{2^m}$.

Uso de Elementos Temporários

O uso de elementos temporários durante o cálculo de algumas das operações é inevitável e pode acarretar em grande utilização de memória, que é um dos recursos mais críticos dos smart cards. Outra característica importante do Java Card é a não obrigatoriedade do *garbage collector*, sendo um risco a alocação constante de memória do sistema. Assim, adotamos o conceito do uso de registradores de trabalho, que são elementos do corpo finito mantidos estáticos na classe, ou seja, são únicos para todos os elementos do corpo. Desta forma, garantimos o reuso das variáveis temporárias da classe, uma das recomendações fortemente ressaltadas na literatura de implementações para Java Card.

Para determinarmos a quantidade de registradores criados, fizemos uma análise preliminar dos algoritmos e verificamos que seria necessário um registrador com o dobro do tamanho de um elemento do corpo, requisito dos algorimtos de multiplicação e de quadrado. Além deste elemento especial, foi constatado que o algoritmo de inversão requeria o uso de três elementos do corpo adicionais. As necessidades dos demais algoritmos poderiam ser atendidas plenamente com a reutilização dos registradores criados.

Adição e Subtração

A operação de adição é realizada usando apenas uma das operações nativas da arquitetura. Uma vez que em polinômios não há propagação de *carry* entre os elementos, basta a adição em $\mathbb{F}_2$ entre os coeficientes de mesmo expoente. Uma vez que os coeficientes são binários, os resultados possíveis para a soma de dois elementos $a, b \in \mathbb{F}_2$ são

a	b	$a+b$	$a-b$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

que é idêntico ao resultado da função **xor** (representada por $\oplus$). Observamos também que a subtração (colocada na tabela) é idêntica à adição, sendo possível que os dois algoritmos compartilhem o mesmo código para evitar desperdício de espaço. Assim, a soma é executada como um **xor** bit a bit entre as palavras de mesmo índice. O código gerado para a soma é mostrado no Algoritmo 5.1. O *array* usado para armazenar os polinômios contidos nos elementos do corpo é chamado de *polynomial*, sendo este uma variável de instância da classe que representa um elemento no corpo finito.

Algoritmo 5.1 Exemplo de código gerado para a adição em $\mathbb{F}_{2^m}$.

ENTRADA: Um elemento $B \in \mathbb{F}_{2^m}$ a ser somado com o elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this + B$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

for(short i=0; i < ⌈m/16⌉; i++)
   auxC[i] = (short) (auxA[i] ^ auxB[i]);
```

Note que foram usadas variáveis temporárias para armazenar os endereços dos *arrays* onde os polinômios do elementos são representados, inclusive do polinômio interno ao próprio objeto sobre o qual a operação de adição foi realizada. Esta estratégia foi adotada pois as indireções se mostraram operações caras no Java Card, tendo sido atingida uma redução de tempo em torno de 50%. O valor $\lceil m/16 \rceil$ é calculado pelo gerador em tempo de construção do código e em seu lugar é colocado o resultado do cálculo, ficando assim o algoritmo específico para o tamanho do corpo em questão.

Uma alternativa de otimização que poderia ser empregada seria a expansão do laço, evitando o custo de verificação da condição e do incremento do contador. No entanto, a redução de tempo seria muito pequena se comparada com o aumento do tamanho do código para a realização desta operação, fato que nos fez desconsiderar esta alternativa.

Redução

A redução é feita sempre que o resultado de um cálculo tenha coeficientes não-nulos em expoentes maiores ou iguais a m. Este método é usado como auxiliar para o cálculo da multiplicação e do quadrado, as únicas operações em que esta situação é verificada. Desta forma, o algoritmo de redução espera elementos com maior expoente de até $2m-2$. O algoritmo usado foi proposto por López e Dahab em [35] que utiliza a relação de equivalência

$$x^{-m}(x^t + 1) \equiv 1 \bmod (x^m + x^t + 1) \tag{5.1}$$

onde $x^m + x^t + 1$ é o trinômio irredutível que serve como base para o corpo finito e $m > t$. O problema da redução consiste então em usar a relação mostrada acima para fazer com que os termos maiores ou iguais a m deixem de existir.

Uma vez que cada elemento do *array* contém 16 termos do polinômio, é possível representar um elemento A por

$$A_k x^{16k} + \cdots + A_2 x^{16 \cdot 2} + A_1 \cdot x^{16} + A_0 x^0$$

onde $16k < 2m - 2 < 16(k+1)$, ou seja, o elemento de expoente $2m-2$ está na última posição do *array*. Para que o elemento k seja reduzido, basta multiplicá-lo pela relação de equivalência da Equação 5.1, obtendo:

$$A_k x^{16k} x^{-m}(x^t + 1) = A_k x^{16k+t-m} + A_k x^{16k-m}.$$

Avaliando o segundo termo da equação, verificamos a existência de uma soma entre dois termos do coeficiente A_k. Na segunda parte da soma, percebemos que o valor $16k - m < m - 2$, uma vez que por hipótese $16k < 2m - 2$. Desta forma, este valor é reduzido de forma adequada. Já a primeira parte da soma tem a adição do valor de t, o que pode gerar duas situações distintas:

1. A diferença entre m e t ser menor que 16, o que pode fazer com que o termo A_k não fique nulo. Caso seja verificado que o termo não ficou nulo, deve ser feita uma nova aplicação da Equação 5.1 até que o elemento A_k fique nulo. A garantia de que isto irá acontecer está no fato de que $t - m$ sempre será um valor negativo, o que acaba fazendo com que o valor da posição A_k seja "deslocado" de $|t - m|$ posições para a direita a cada aplicação da equação.

2. A diferença entre m e t ser maior que 16. Neste caso, o termo A_k é deslocado para uma posição anterior à sua, o que automaticamente torna nula sua posição.

O gerador de código é capaz de detectar em qual das situações o corpo que está sendo gerado se enquadra e verifica a necessidade de se incluir a redução de um termo em um laço ou não. Adicionalmente, é verificado o lugar onde os termos reduzidos devem ser inseridos, uma vez que o posicionamento é dado por $(16k + t - m)$ e $(16k - m)$, que

dependem de k. Uma vez reduzido o termo k, deve ser reduzido o termo $k-1$ de forma idêntica, até que seja reduzido o termo $\lceil k/2 \rceil$.

A redução da posição do *array* onde está o termo m é um caso especial, uma vez que os coeficientes das posições menores que m não precisam ser alterados. Desta forma, nós optamos por remover do elemento do *array* a porção relativa aos coeficientes maiores ou iguais a m e colocá-los em uma variável temporária B. Como o primeiro coeficiente da variável possui expoente m, a aplicação da equação de redução resulta em

$$(Bx^m)\, x^{-m} \left(x^t + 1\right) = Bx^t + B.$$

Assim, a reintegração dos coeficientes da variável B com o polinômio consiste na incorporação de B aos termos x^t e x^0. Uma vez que o valor de t é fixo, a forma como os termos serão incorporados poderá ser prevista em tempo de geração de código, permitindo que sejam criadas instruções específicas para tratar este caso, o que pode acarretar num ganho de desempenho.

Multiplicação

A multiplicação é uma das operações mais custosas do desenvolvimento, perdendo apenas para a inversão. Ela também é uma operação crítica, pois é muito usada. Desta forma, procuramos testar duas abordagens diferentes, uma com o uso do algoritmo de multiplicação padrão, que é amplamente usado, e outra com o uso de um algoritmo de multiplicação proposto por López e Dahab [35] que tem se mostrado mais eficiente. Há outros algoritmos na literatura, porém muitos deles utilizam a pré-computação de alguns termos ou mesmo alguns pontos, o que acarretaria num custo excessivo de memória, não respeitando assim um de nossos requisitos básicos.

Multiplicação Padrão (*Standard*)

O Algoritmo 5.2, conhecido por algoritmo de multiplicação padrão (*standard*), é o mais intuitivo, uma vez que ele visa a implementação da forma básica de multiplicação de polinômios. Tem como principal vantagem a possibilidade de dispensar um procedimento à parte para realizar a redução módulo o polinômio irredutível que define o corpo.

Algoritmo 5.2 Algoritmo de multiplicação padrão em $\mathbb{F}_{2^m}$.

ENTRADA: Um elemento $B \in \mathbb{F}_{2^m}$ a ser multiplicado pelo elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this * B \pmod{x^m + x^t + 1}$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

auxC = 0; // coloca 0 em todas as posições de auxC

for(short i=(⌈m/16⌉ − 1); i >=0; i--) {
   for(short j=15; j >=0; j--) {

      //esta operação de shift na verdade é um conjunto
      // de operações que é expandida na geração de código
      auxC = auxC << 1;
      // caso tenha ocorrido carry da operação de shift,
      // irá surgir um termo x^m em C.  a redução é feita pela
      // substituição de x^m por x^t + 1, uma vez que é verificado:
      // x^m ≡ x^t + 1 mod (x^m + x^t + 1)
      if (carry != 0)
         auxC = auxC + (x^t + 1);
      // caso o bit j de auxA[i] não for zero
      if(auxA_j[i] != 0) {
         // esta soma na verdade é uma adição no corpo,
         // como o código da adição é pequeno, ele é
         // expandido aqui
         auxC = auxC + auxB;
      }
   }
}
```

A implementação mais eficiente realiza a redução durante as iterações do algoritmo, toda vez em que um termo x^m surgir no resultado. A redução deste termo utiliza a equivalência

$$x^m \equiv x^t + 1 \bmod (x^m + x^t + 1)$$

com a substituição do termo x^m pelo termo $x^t + 1$.

Um fator importante é a inicialização da variável de resultado, quando deve ser atribuído o valor zero a todas as suas posições. Na API padrão está prevista uma funcionalidade de preenchimento do conteúdo de um *array* com um determinado valor. No entanto, o cartão que utilizamos não disponibiliza esta funcionalidade de forma satisfatória, com o risco desta operação não apresentar resultados corretos. É então recomendado pelo fabricante substituir a chamada ao método de inicialização por um laço, o que evidentemente é um processo mais lento do que a chamada a um método nativo do cartão.

Multiplicação de López e Dahab

O algoritmo proposto por López e Dahab baseia-se em um conjunto de propriedades obtidas devido à forma de armazenamento dos elementos do corpo finito. A representação dos elementos em *arrays* que permitem o acesso a qualquer valor em tempo constante motivou a observação de propriedades semelhantes em termos dos polinômios que compartilham a mesma posição dentro da palavra de memória, ou seja, foi verificada uma relação entre os termos do polinômio cujos coeficientes estavam armazenados na mesma posição mas em elementos diferentes do *array*.

Como exemplo da relação, o termo do polinômio do primeiro bit de cada palavra é dado pelo produto

$$c_{16i}x^{16i},$$

onde c_k é o k-ésimo coeficiente do polinômio e i é a i-ésima palavra do *array*, que vai de 0 a $\lceil m/16 \rceil$. Desta forma, tendo em mente o algoritmo de multiplicação padrão, a parcela da multiplicação de um polinômio B pelas primeiras posições do *array* e acumulando em C seria algo na forma

$$C = C + (Bc_{16i}x^{16i}),$$

onde, caso c_{16i} for zero, não é necessário realizar o produto pois a parcela entre parênteses seria igual a zero. O que é interessante observar é que o produto por um múltiplo do tamanho dos elementos do *array*, no caso 16, permitiria que a soma fosse feita sem a necessidade de deslocar fisicamente o polinômio B em $16i$ posições para a direita, o que seria a operação normal de ser feita. Como o deslocamento seria de uma palavra inteira, basta que a soma seja feita com um desalinhamento entre B e C de modo a simular este

deslocamento. A Figura 5.3 ilustra este processo para o caso de $i = 1$.

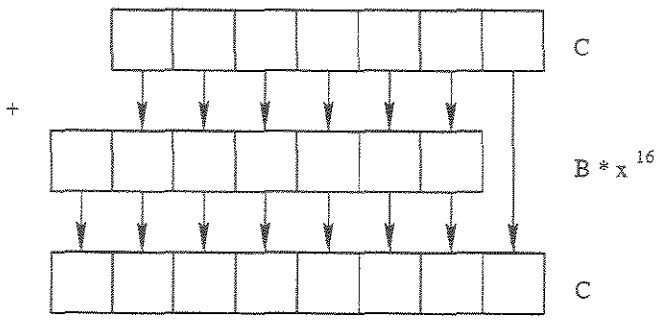


Figura 5.3: Exemplo de processo de soma entre $C = C + Bx^{16}$.

O processo de deslocamento mostrado acima tem como principal atrativo ser livre de *carry*, o que torna o processo mais rápido. No entanto, foi observado que este processo só serviria para as primeiras posições de cada elemento do *array*, fazendo-se necessário então promover uma adaptação no processamento dos demais bits para a utlização deste método.

A estratégia adotada então foi a de iniciar o cálculo da esquerda para a direita, ou seja, do bit de maior posição para o bit de menor posição. A cada iteração, a posição sendo processada tinha um tratamento idêntico ao dado à primeira posição da palavra. Em toda iteração o resultado era multiplicado por x, ou seja, era feito um deslocamento para a esquerda com a propagação de *carry*. Ao final do processo, o resultado de um bit i havia sido multiplicado por x um total de i vezes, fazendo com que o resultado ficasse em seu devido lugar.

O que deve ser questionado é o ganho que esta estratégia provê, uma vez que há o deslocamento com *carry* de qualquer forma. O ganho está no fato de, em uma iteração, serem tratados os bits de mesma posição de todas as palavras, ou seja, é processado o bit i de todas as $\lceil m/16 \rceil$ palavras usadas para armazenar o polinômio usando o deslocamento de bits livre de *carry* e seria feito somente um deslocamento complexo de bits. Assim, os deslocamentos de bits com propagação de *carry* são reduzidos de m do método padrão para 16 neste método. Os demais $m - 16$ deslocamentos seriam os deslocamentos simples, conforme mostrado. O Algoritmo 5.3 mostra o processo como um todo.

Algoritmo 5.3 Algoritmo de multiplicação de López e Dahab em $\mathbb{F}_{2^m}$.

ENTRADA: Um elemento $B \in \mathbb{F}_{2^m}$ a ser multiplicado pelo elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this * B \pmod{x^m + x^t + 1}$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

// usa um registrador de trabalho da classe com o dobro do tamanho
// de um elemento do corpo
reg = 0; // coloca 0 em todas as posições de reg

for(short j=15; j >=0; j--) {
      //esta operação de shift na verdade é um conjunto
      // de operações que é expandida na geração de código
      reg = reg << 1;
   for(short i=(⌈m/16⌉ - 1); i >=0; i--) {

      // caso o bit j de auxA[i] não for zero
      if(auxA_j[i] != 0) {
         // esta soma na verdade é uma adição no corpo,
         // como o código da adição é pequeno, ele é
         // expandido aqui
         // a multiplicação por x^16i na verdade é
         // é apenas um desalinhamento entre os laços de
         // soma, ou seja, é feito reg[k] = reg[k] ⊕ auxB[k+i]
         reg = reg + (auxB * x^16i);
      }
   }
}
// redução usando o algoritmo descrito anteriormente
reg = reg   (mod x^m + x^t + 1);

// copia o resultado para C
auxC = reg;
```

O penúltimo passo, a redução pelo polinômio irredutível, é a principal desvantagem deste algoritmo em relação ao padrão. Não é possível aplicar a mesma idéia que é usada no algoritmo padrão pois a cada iteração do laço mais externo o subproduto gerado em `reg`

não é bem comportado como é o caso do deslocamento de bits verificado no algoritmo padrão. Note que este algoritmo requer um registrador de trabalho com o dobro do tamanho de um elemento do corpo finito. Este registrador é usado para armazenar o resultado do cálculo antes da redução, outra diferença em relação ao algoritmo padrão, uma vez que este não requer registradores extras. No entanto, este algoritmo apresenta um ganho significativo de performance em relação ao algoritmo padrão, o que poderia justificar sua escolha.

Quadrado

A operação de quadrado, à primeira vista, não seria especial, uma vez que seria equivalente a apenas multiplicar um elemento por si mesmo para realizá-la. Entretanto, a informação de que os elementos multiplicados são idênticos permite que sejam exploradas algumas particularidades que tornam a execução deste procedimento extremamente mais rápida que a solução óbvia.

Para se obter o quadrado de um polinômio qualquer representado por

$$c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + \cdots + c_2x^2 + c_1x + c_0$$

os expoentes de cada coeficiente devem ser multiplicados por 2, resultando em

$$c_{m-1}x^{2(m-1)} + c_{m-2}x^{2(m-2)} + \cdots + c_2x^4 + c_1x^2 + c_0.$$

Uma vez que a posição onde cada coeficiente do polinômio é fixa, a construção de uma tabela com o posicionamento correto de cada elemento é a melhor estratégia. Assim, para cada elemento do *array* que armazena o polinômio são calculados os dois elementos resultantes de seu quadrado. Esta estratégia leva a uma boa performance, mas foi necessário primeiramente realizar o dimensionamento do tamanho da tabela para que as restrições de espaço impostas pelos smart cards não fossem violadas.

A tabela de valores pré-calculados deve conter um mapeamento entre um pequeno polinômio e seu quadrado. O correto posicionamento do valor obtido na tabela é de responsabilidade do algoritmo de cálculo do quadrado. Para que o mapeamento seja útil, ele deve cobrir todos os valores possíveis para os coeficientes do polinômio. Desta forma,

como usamos polinômios com coeficientes em $\mathbb{F}_2$, o número de elementos na tabela para um polinômio de grau w é 2^w. Um tamanho razoável para o polinômio usado na tabela foi de $w = 4$, ou seja, foram pré-calculados os quadrados de todos os polinômios com grau menor que 4. A tabela foi construída em um *array* onde o quadrado do polinômio $c_3x^3 + c_2x^2 + c_1x + c_0$ foi armazenado na posição $(c_3c_2c_1c_0)_2$, o que permitiu um acesso rápido dos valores.

O algoritmo de quadrado então consistiu na divisão do *array* que armazena o polinômio em partes com 4 bits cada e na busca destas partes na tabela de pré-computação, com o correto posicionamento dos resultados. Este procedimento resulta em um polinômio de grau $2m - 1$, o que faz necessária a aplicação do procedimento de redução ao final do processo.

Inversão

O cálculo do inverso de um elemento do corpo finito é a operação mais custosa da aritmética. Apesar de apresentar tempos na mesma ordem de grandeza que a multiplicação, a inversão de um elemento é cerca de quatro vezes mais lenta. Desta forma, a escolha do algoritmo a ser usado deve ser muito criteriosa, pois o sucesso de todo o sistema criptográfico pode ser comprometido por uma inversão muito lenta no corpo finito.

O algoritmo clássico para a inversão em $\mathbb{F}_{2^m}$ é o **Euclidiano Extendido**, que tem como base o algoritmo de cálculo do máximo divisor comum de Euclides. Em 1995 foi proposto um novo algoritmo de inversão, conhecido como **AIA** (*Almost Inverse Algorithm*) [50]. Como o nome sugere, o resultado do AIA é um valor muito próximo do inverso do polinômio, o que torna necessário um passo adicional para que o resultado esperado seja obtido. Apesar do passo adicional, o AIA tem apresentado melhor desempenho que o Euclidiano Extendido. Uma proposta feita em 2000 sugere uma modificação no AIA de modo que o passo adicional não fosse necessário e ainda assim o número de passos sua realização permanecesse o mesmo. Esta modificação, chamada de **MAIA** (*Modified Almost Inverse Algorithm*) [24], foi escolhida para nossa implementação de inversão.

Algoritmo 5.4 Algoritmo de inversão MAIA em $\mathbb{F}_{2^m}$.

ENTRADA: Um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this^{-1} \pmod{x^m + x^t + 1}$

```
short[] inv = C.polynomial;
short[] B = reg1;
short[] F = reg2;
short[] G = reg3;

// inicializa os registradores de trabalho
inv = 1; B = 0; F = this; // faz uma cópia de this.polynomial em F
G = x^m + x^t + 1

// laço principal
while (true) {
   // enquanto F for divisível por x (testa o bit 0)
   while (F[0]_0 == 0) {  // passo 1
      // divida F por x
      F >>>= 1;
      inv >>>= 1;
      // se o bit 0 de inv não era nulo, deve surgir o termo x^-1, ajustado
      // usando a relação x^-1 ≡ x^(m-1) + x^(t-1)  (mod x^m + x^t + 1)
      if (x^-1)
         inv = inv + (x^(m-1) + x^(t-1));
   }
   // caso o valor de F tenha ficado igual a 1,
   //o conteúdo de inv é o resultado procurado
   if (F == 1)
      break; // quebra o laço principal

   // coloca o maior dos valores em F, para garantir que
   // sempre irá ocorrer reduções no passo 1
   if (deg(F) < deg(G)) {
      F ⇔ G;
      inv ⇔ B;
   }
   F = F + G;
   inv = inv + B;
}
```

O Algoritmo 5.4 apresenta algumas das escolhas que foram feitas na implementação da inversão. A priori, foi necessário o uso de três registradores auxiliares para a implementação. Uma vez que o Java Card não prevê o multiprocessamento, o uso de variáveis estáticas comuns a todos os objetos como registradores de trabalho não causaria problemas de concorrência. Assim, o uso de apenas três elementos do corpo como registradores de trabalho não penalizou de forma contundente nossa implementação, com relação ao espaço ocupado.

O relacionamento entre as variáveis usadas (inv, B, F, G) é dado pelas equações

$$\begin{aligned} inv \cdot F + k \cdot G &= F \\ B \cdot F + l \cdot G &= G \end{aligned}$$

onde $inv = 1$, $B = 0$, $G = (x^m + x^t + 1)$ e F é o polinômio de que desejamos obter o inverso. No algoritmo, o valor de F do lado direito da equação é continuamente dividido por x, até que se torne 1. No entanto, a cada redução o valor de B é ajustado de tal maneira que a equação é mantida. Quando F atinge 1, o valor de B está ajustado como sendo o inverso de F.

5.1.4 Implementação da Aritmética Básica em OEF

A aritmética dos corpos de extensão ótima foi feita com o uso de uma base polinomial, onde cada coeficiente do polinômio era um número inteiro limitado ao tamanho de uma palavra de memória. Isto significa que, para o corpo $\mathbb{F}_{p^m}$, p é um número primo que pode ser representado por uma única palavra de memória. A condição para a existência do corpo para um dado valor de m é a existência de um valor $w \in \mathbb{F}_p$ tal que $x^m - w$ seja um polinômio irredutível. Devem então ser fornecidos ao gerador os valores de p, m e w.

A vantagem da utilização de OEF está no fato de ser possível utilizar os recursos de aritmética do próprio hardware para a realização das operações em $\mathbb{F}_p$. Assim, diferentemente de $\mathbb{F}_{2^m}$, onde a unidade de operação estava em um bit, é possível realizar operações usando palavras de memória de forma eficiente.

A seguir será mostrada a forma como os elementos do corpo foram representados na implementação e os algoritmos utilizados para a realização das operações no corpo finito.

Representação dos Elementos do Corpo

A representação física dos elementos do corpo foi feita em um *array* de elementos do tipo *short*. Cada posição do *array* representa um coeficiente do polinômio, que é um valor em $\mathbb{F}_p$. O expoente associado ao coeficiente é dado pelo índice do elemento no *array*. Desta forma, o número de posições usadas é igual a m.

Uso de Elementos Temporários

De forma semelhante a $\mathbb{F}_{2^m}$, o uso de elementos temporários é necessário para a realização de algumas das operações do corpo finito. Como a criação de tais elementos em cada operação poderia penalizar o desempenho do algoritmo e promover um desperdício de memória, uma vez que a existência de um mecanismo de *garbage collector* não é assegurada, os elementos temporários, que foram chamados de registradores de trabalho, são variáveis estáticas da classe que representa o elemento do corpo.

A análise dos algoritmos implementados mostrou uma nova necessidade de um registrador com o dobro do tamanho de um elemento do corpo, uma necessidade da operação de multiplicação. A operação de redução foi então construída para trabalhar sobre este registrador.

Adição

A adição entre os polinômios é feita através da soma modular entre os coeficientes de mesmo expoente. Uma vez que a aritmética não prevê *carry* entre operações feitas de um expoente para outro, a adição apresenta um bom desempenho em relação a esta operação em $\mathbb{F}_p$. O Algoritmo 5.5 mostra o procedimento de adição entre dois elementos de OEF.

Aqui também podem ser observadas as variáveis temporárias usadas para armazenar as referências dos polinômios contidos nos operandos e assim evitar um número grande de indireções no acesso aos elementos dos *arrays*. Vale ressaltar que é necessário que p seja escolhido de tal forma que seja pelo menos dois bits menor que o tamanho dos elementos usados para representar os elementos, ou seja, que possa ser representado por 14 bits. Esta restrição é imposta para que a verificação se houve *overflow* possa ser feita

diretamente, como mostrado no Algoritmo 5.5. A necessidade de dois bits, ao invés de um, está no fato de não estarem previstas na linguagem operações não sinalizadas, sendo então evitado o uso do bit de sinal para o armazenamento de informações.

Algoritmo 5.5 Adição em OEF.

ENTRADA: Um elemento $B \in \mathbb{F}_{p^m}$ a ser somado com o elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this + B$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

for(short i=0; i < m; i++) {
   auxC[i] = auxA[i] + auxB[i];
   if (auxC[i] >= p)
      auxC[i] = auxC[i] - p;
}
```

Subtração

A subtração entre os elementos do corpo finito é feita de forma semelhante à operação de adição. A diferença, além de obviamente ser usada a operação de subtração ao invés da soma, está no fato da possibilidade de ocorrência de *underflow* ao invés de *overflow*, ou seja, caso a operação dê um valor negativo, deve ser feito um ajuste para que os coeficientes sejam elementos de $\mathbb{F}_p$. O Algoritmo 5.6 mostra os passos necessários para a realização da subtração.

Redução

Esta operação tem como objetivo manter os polinômios que representam os elementos do corpo finito usado em OEF reduzidos módulo o binômio irredutível $x^m - w$, onde $w \in \mathbb{F}_p$, que define o OEF. Uma vez que esta operação é necessária principalmente na multiplicação, que por sua vez produz elementos com expoentes de valor máximo $2m$,

basta que seja usada a seguinte relação nos termos com expoentes maiores ou iguais a m

$$x^m \equiv w \pmod{x^m - w}.$$

Note que, se o valor de w for pequeno, multiplicações podem ser substituídas por somas. Para o caso particular de $w = 2$, um deslocamento (*shift*) pode ser utilizado. O Algoritmo 5.7 mostra como a equação deve ser aplicada no procedimento de redução.

Algoritmo 5.6 Subtração em OEF.

ENTRADA: Um elemento $B \in \mathbb{F}_{p^m}$ a ser subtraído do elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this - B$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

for(short i=0; i < m; i++) {
   auxC[i] = auxA[i] - auxB[i];
   if (auxC[i] < 0)
      auxC[i] = auxC[i] + p;
}
```

Algoritmo 5.7 Redução em OEF.

ENTRADA: Um *array* A com $2m$ elementos.

SAÍDA: $A = A \pmod{x^m - w}$.

```
short aux;
for(short i=(2m-1); i >= m; i--) {
   aux = A[i];
   A[i] = 0;
   aux = aux * w; // caso w = 2, pode ser usado (<< 1)
   A[i-m] = A[i-m] + aux;
   // reduz o valor de A[i-m] em F_p
   // supondo um valor de w pequeno
   while(A[i-m] >= p)
      A[i-m] = A[i-m] - p;
}
```

Multiplicação

A multiplicação de dois elementos A e B em OEF pode ser feita com o algoritmo clássico de multiplicação de polinômios. Este método consiste em multiplicar cada um dos termos de A por todos os termos de B, somar os resultados intermediários e reduzir o polinômio resultante com o Algoritmo 5.7.

Algoritmo 5.8 Multiplicação em OEF.

ENTRADA: Um elemento $B \in \mathbb{F}_{p^m}$ a ser multiplicado pelo elemento contido no objeto e um elemento C que será usado para armazenar o resultado da operação.

SAÍDA: $C = this * B \pmod{x^m - w}$

```
short[] auxA = this.polynomial;
short[] auxB = B.polynomial;
short[] auxC = C.polynomial;

reg = 0; // array com 2m posições

for (short i=(m − 1); i >= 0; i--) {
   k = i;
   // particiona A[i] em dois, fazendo A[i] = a1·2^8 + a0
   a0 = auxA[i] & 0x0FF;
   a1 = auxA[i] >>> 8;

   for (short j=0; j < m; j++) {
      //particiona B[i] em dois
      b0 = auxB[j] & 0x0FF;
      b1 = auxB[j] >>> 8;
      reg[k++] = (a1·b1)·2^16 + (a1·b0 + a0·b1)·2^8 + (a0·b0)  (mod p);
   }
}
auxC = reg  (mod x^m − w);
```

O Algoritmo 5.8 mostra o procedimento de multiplicação implementado. Nele podemos observar que o número de multiplicações em $\mathbb{F}_p$ são de ordem quadrática sobre m. As multiplicações em $\mathbb{F}_p$ são realizadas com a divisão dos coeficientes em duas partes, uma vez que o produto de dois números de k bits gera um número de $2k$ bits. Assim, as partes com metade do tamanho da palavra de memória podem ser multiplicadas diretamente

usando a instrução nativa disponível na arquitetura. A cada multiplicação realizada, é possível que seja necessária a redução módulo p. Esta necessidade pode ser levantada em tempo de geração de código, onde o número de bits obtidos com a operação é confrontado com o número de bits de p.

Quadrado

O cálculo do quadrado em OEF consiste em aplicar o algoritmo de multiplicação do elemento sobre ele mesmo. É possível modificar o Algoritmo 5.8 de tal modo que o fato dos elementos serem iguais seja usado para evitar que alguns cálculos sejam feitos duas vezes, como $(a_1b_0+a_0b_1) = 2(a_0a_1)$ caso $a_0 = b_0$ e $a_1 = b_1$. No entanto, os ganhos com este método não seriam significativos o suficiente para justificar o espaço que seria consumido por este método.

Inversão

A inversão implementada em EOF é baseada no algoritmo AIA, que trabalha com duas equações,

$$\begin{aligned} AB + XG &= F \\ AC + YG &= G \end{aligned} \tag{5.2}$$

onde $A, B, X, Y, G, F \in \mathbb{F}_{p^m}$. A é o polinômio que desejamos inverter, F contém inicialmente o valor de A, enquanto G é inicializado com o polinômio irredutível que determina o corpo. Assim, os valores iniciais de B e C para que as equações sejam verdadeiras devem ser 1 e 0, respectivamente.

O algoritmo aplica então transformações em B, C, F, G nas equações de tal forma que o valor de F fique igual a x^k, para $k < 2m$. Desta forma, a equação fica sendo $AB' + X'G = x^k$, onde pode-se notar claramente que $B' = x^kA^{-1}$. Um passo adicional para calcular $B'x^{-k}$ deve ser feito para que seja obtido o valor de A^{-1}. O Algoritmo 5.9 mostra as transformações usadas para o cálculo do inverso pelo AIA.

Algoritmo 5.9 Algoritmo de inversão AIA em OEF.

ENTRADA: Um elemento *res* que será usado para armazenar o resultado da operação.

SAÍDA: $res = this^{-1} \pmod{x^m - w}$

```
short[] B = inv.polynomial, C = reg1; F = reg2; G = reg3;

B = 1; C = 0; F = this; G = (x^m - w);

// laço principal
while (true) {
   // enquanto F for divisível por x
   while (F[0] == 0) {  // passo 1
      // divida F por x
      F[i] = F[i+1], para 0 ≤ i < m - 1;
      F[m - 1] = 0;
      // multiplique B por x
      aux = B[0]; // armazena o valor de B para verificar se houve borrow
      B[i] = B[i+1], para 0 ≤ i < m - 1;
      B[12] = 0;
      // caso tenha ocorrido borrow, ajusta o termo x^-1 que surgiu
      if (aux != 0)
         B[12] = aux * w^-1  (mod p);
   }
   // caso o grau máximo de F seja 0, ou seja, F possui somente o termo constante
   if (deg(F) == 0) {
      B[i] = B[i] * F[0]^-1  (mod p), para 0 ≤ i < m - 1;
      break; // quebra o laço principal
   }
   // coloca o maior dos valores em F, para garantir que
   // sempre irá ocorrer reduções no passo 1
   if (deg(F) < deg(G)) {
      F ⇔ G;
      B ⇔ C;
   }
   aux = G[0] * F[0]^-1  (mod p);
   F = F * aux   (mod p), para 0 ≤ i < m - 1;
   B = B * aux   (mod p), para 0 ≤ i < m - 1;
   F = F + G;
   B = B + C;
}
res = B;
```

Uma outra alternativa proposta por Bailey [65] é uma modificação no algoritmo de Itoh-Tsujii [30] apresentada em [6]. No entanto, a utilização de uma tabela para o armazenamento do mapa de Frobenius torna esta alternativa custosa em relação ao espaço. É necessário, portanto, avaliar se o ganho em desempenho justifica a perda em espaço.

5.2 Aritmética no Grupo Elíptico

A aritmética no grupo formado por pontos sobre uma curva elíptica é constituída da operação de adição de pontos. Como mostrado na Seção 4.2, a equação usada na adição dos pontos pode variar dependendo da característica do corpo sobre o qual a curva foi definida. Além disto, há também a possibilidade de escolha de um sistema de coordenadas onde as operações se tornem mais eficientes. É comum o uso de coordenadas projetivas para reduzir o número de inversões no corpo finito. No entanto, estas inversões são substituídas por várias operações de multiplicação. Assim, esta alternativa somente se torna vantajosa quando a multiplicação tem um desempenho muito superior a operação de inversão, o que não foi verificada em nossa implementação.

A seguir será apresentada a modelagem da representação dos pontos da curva elíptica, onde foi utilizada uma interface que define o comportamento esperado por um ponto elíptico. A seguir será dada a implementação da operação de soma nos corpos implementados e uma extensão desta operação, a multiplicação escalar.

5.2.1 Modelo de Objetos dos Pontos Elípticos

Os pontos de uma curva elíptica definida em coordenadas afins possuem o mesmo comportamento básico, que foi agrupado em uma interface que define o contrato geral dos pontos, como mostrado na Figura 5.4. Podemos observar que foi delegado ao ponto a responsabilidade de realizar a operação de soma com outro ponto e a multiplicação escalar. Os parâmetros da curva são mantidos em uma implementação da interface que define o comportamento de uma curva elíptica, que basicamente provê os valores de a e b que definem a equação da curva.

Para garantir que todos os pontos compartilham os mesmos parâmetros da curva e

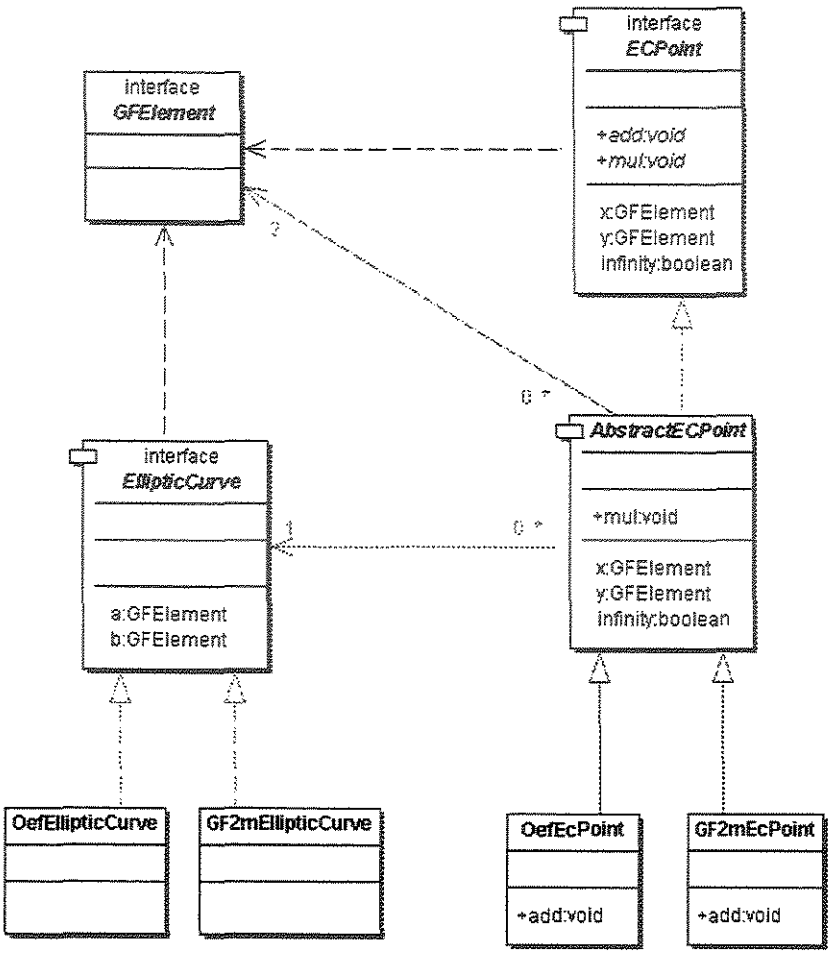


Figura 5.4: Diagrama de classes UML para a aritmética no corpo finito.

para promover uma economia de espaço físico, as implementações concretas da interface `EllipticCurve` são construídas de acordo com o *pattern* **Singleton** [20]. Além da interface de pontos elípticos, foi disponibilizada uma classe abstrata que mantém a estrutura de um ponto definido sobre coordenadas afins. Uma vez que o algoritmo de multiplicação escalar não depende da característica do corpo finito sobre o qual a curva está definida, a classe abstrata provê a implementação do algoritmo clássico para a realização desta operação.

5.2.2 Operação de Adição

A operação de soma entre dois elementos pertencentes a curva elíptica é definida de tal forma que os pontos formam um grupo abeliano, condição necessária para a construção dos sistemas criptográficos. Conforme discutido na Seção 4.2, a equação da curva, e por consequência a operação de adição, são diferentes de acordo com a característica do corpo sobre o qual a curva é definida. Desta forma, foram construídas duas implementações concretas para a classe abstrata básica dos pontos elípticos, uma para os corpos de característica 2, com a aritmética sobre $\mathbb{F}_{2^m}$, e uma implementação para OEF. A seguir a forma como estas operações foram implementadas será mostrada com mais detalhes.

Adição na Curva Elíptica sobre $\mathbb{F}_{2^m}$

O Algoritmo 5.10 mostra a implementação do método de soma de dois pontos elípticos definidos sobre $\mathbb{F}_{2_m}$ em coordenadas afins. Neste procedimento, são usadas variáveis locais para o armazenamento das referências dos parâmetros da curva elíptica, das coordenadas dos operandos e das coordenadas que definem o resultado. Esta estratégia é necessária pois a indireção à memória é uma operação com custo elevado no Java Card e deve ser evitada.

As variáveis que armazenam os valores temporários foram extraídas de um dos pontos elípticos que servem como registradores de trabalho para o algoritmo de multiplicação escalar na classe abstrata de ponto elíptico. Além destas variáveis, a estrutura do ponto que recebe o resultado também foi aproveitada durante os cálculos, sendo desta forma promovida a reutilização de objetos já criados, um dos princípios fundamentais do Java Card. No entanto, esta estratégia impões a restrição de que o objeto onde o resultado é colocado não pode ser usado como operando da adição.

Algoritmo 5.10 Algoritmo de Adição Elíptica sobre $\mathbb{F}_{2^m}$.

ENTRADA: Um ponto elíptico P a ser somado com o objeto e um ponto elíptico R para armazenar o resultado.

SAÍDA: $R = this + P$

```
// caso algum dos valores corresponda ao ponto no infinito, o resultado
// é o outro valor
if (this.infinity) { >R.assign(P); return; }
if (P.isInfinity()) { R.assign(this); return; }
if (P.isInverseOf(this)) { R.setInfinity(true); return; }
// armazena as referências das coodenadas dos pontos em variáveis
// locais para evitar o custo de indireções a memória
GFElement a,b; // parâmetros da curva elíptica
GFElement x1,y1; // coordenadas de this
GFElement x2,y2; // coordenadas de P
GFElement x3,y3; // coordenadas de R (resultado)
GFElement regx, regy; // coordenadas do registrador da classe

if (x1.equals(x2)) {
   if (y2.equals(x2)) {
      // reg = x1 + y1/x1
      x1.inv(reg); y1.mul(reg, x3); x1.add(x3, reg);
      // x3 = reg^2 + reg + a
      reg.sqr(x3); regx.add(regx,y3); a.add(y3,x3);
   }
   else { R.setInfinity(true); return; }
}
else {
   // regx = (y2 + y1)/(x2 + x1)
   x1.add(x2,regx); regx.inv(x3); // x3 = (x2 + x1)^-1
   y1.add(y2,y3); x3.mul(y3,regx);
   // x3 = regx^2 + regx + x1 + x2 + a
   regx.sqr(x3); x3.add(regx,x3); x3.sum(x1,x3);
   x3.sum(x2,x3); x3.sum(a,x3);
}
// y3 = regx * (x1 + x3) + x3 + y1
x1.add(x3,regy); regx.mul(regy,y3);
y3.add(x3,y3); y3.add(y1,y3);
```

Adição na Curva Elíptica sobre OEF

A adição de dois pontos elípticos definidos sobre o corpo finito OEF é mostrada no Algoritmo 5.11. Como pode ser observado, a estratégia de usar variáveis locais com referências para as variáveis de classe utilizadas foi novamente empregada, de forma semelhante aos algoritmos de aritmética na curva e de soma em $\mathbb{F}_{2^m}$.

As coordenadas de um dos pontos usados como registradores de trabalho foram usados como variáveis temporárias no procedimento, seguindo assim a linha de reaproveitamento de objetos requerida pela arquitetura do Java Card. Uma vez que os elementos do corpo finito pertencentes ao objeto que recebe o resultado são usados durantes os cálculos, há o requisito de que o ponto que recebe o resultado não seja o mesmo objeto que algum dos operandos do método.

5.2.3 Operação de Multiplicação Escalar

Esta é a principal operação de esquemas criptográficos baseados em curvas elípticas. Dado um inteiro k e um ponto elíptico P, a multiplicação escalar kP é o resultado da aplicação da soma do grupo de P por ele mesmo por k vezes.

A implementação básica desta operação fornecida pela classe abstrata prevê o uso do algoritmo clássico, conhecido por *método binário*, que como o nome sugere se baseia na representação binária de k. Assim, se $k = \sum_{j=0}^{l-1} k_j 2^j$, onde $k_j \in \{0,1\}$ para $0 \leq j < l$ e $l = \log_2 k$, então kP pode ser calculado como

$$kP = \sum_{j=0}^{l-1} k_j 2^j P = 2(\cdots 2(2k_{l-1}P + k_{l-2}P) + \cdots) + k_0 P.$$

O Algoritmo 5.12 mostra a implementação do método binário. Note que o número escalar é representado por um *array* de valores do tipo *short*, onde o bit 0 do elemento de índice 0 do *array* é o bit menos significativo. No algoritmo foram necessários dois pontos usados como registradores, uma vez que a operação de soma elíptica não permite que o ponto que recebe o resultado seja o mesmo que algum dos operandos.

Algoritmo 5.11 Algoritmo de Adição Elíptica sobre OEF.

ENTRADA: Um ponto elíptico P a ser somado com o objeto e um ponto elíptico R para armazenar o resultado.

SAÍDA: $R = this + P$

```
// caso algum dos valores corresponda ao ponto no infinito, o resultado
// é o outro valor
if (this.infinity) { >R.assign(P); return; }
if (P.isInfinity()) { R.assign(this); return; }
if (P.isInverseOf(this)) { R.setInfinity(true); return; }
// armazena as referências das coodenadas dos pontos em variáveis
// locais para evitar o custo de indireções a memória
GFElement a,b; // parâmetros da curva elíptica
GFElement x1,y1; // coordenadas em this
GFElement x2,y2; // coordenadas de P
GFElement x3,y3; // coordenadas de R (resultado)
GFElement regx, regy; // coordenadas do registrador da classe

if (x1.equals(x2)) {
   if (y2.equals(x2)) {
      // reg_x = (3x_1^2 + a)/(2y_1)
      y1.add(y1,regx); regx.inv(y3); // y_3 = (2y_1)^-1
      x1.sqr(regx); regx.add(regx,x3); regx.add(x3,x3); // x_3 = 3x_1^2
      x3.add(a, x3); x3.mul(y3, regx); // reg = (x_3 + a) * y_3
   }
   else { R.setInfinity(true); return; }
}
else {
   // reg_x = (y_2 - y_1)/(x_2 - x_1)
   x2.sub(x1,regx); regx.inv(x3); // x_3 = (x_2 - x_1)^-1
   y2.sub(y1,y3); // y_3 = (y_2 - y_1)
   x3.mul(y3,regx);
}
// x_3 = reg_x^2 - x_1 - x_2
regx.sqr(x3); x3.sub(x1,x3); x3.sub(x2,x3);
// y_3 = reg_x * (x_1 - x_3) - y_1
x1.add(x3,regy); regx.mul(regy,y3); y3.sub(y1,y3);
```

Algoritmo 5.12 Algoritmo de Adição Elíptica sobre OEF.

ENTRADA: Um ponto elíptico P, um número *array* de *short* que representa um número escalar *scalar* e um ponto elíptico R para armazenar o resultado.

SAÍDA: $R = scalar * P$

```
if (this.infinity) { R.setInfinity(true); return; }

ECPoint pow = reg;
ECPoint aux = reg1;
ECPoint swap;
pow.assign(this);
R.setInfinity(true);
short len = scalar.length;

for(short i=0; i < len; i++)
   for(short j=0; j < 16; j++) {
      // se o bit j do elemento i do array não for zero
      if (scalar[i]_j == 1) {
         R.add(pow,aux);
         R.assign(aux);
      }
      // pow = 2 * pow
      pow.add(pow, aux);
      swap = pow;
      pow = aux;
      aux = swap;
   }
```

5.3 Protocolo Criptográfico

Dentre os protocolos criptográficos apresentados na Seção 4.3.1, escolhemos implementar protocolo de estabelecimento de uma chave secreta com base em curvas elípticas, o ECDH. Esta escolha foi feita com base na maior simplicidade com deste protocolo, cuja operação consiste somente no cálculo de uma multiplicação escalar. Os outros protocolos mostrados requerem operações modulares sobre números grandes, o que já haviamos constatado não ser uma boa alternativa [17].

A seguir será mostrada a forma como os dados relativos aos parâmetros do algoritmo

foram estruturados. Além disto, a forma como foi feita a integração da arquitetura estabelecida por nós com a API criptográfica disponibilizada no Java Card.

5.3.1 Modelo de Classes do Protocolo ECDH

A Figura 5.5 apresenta o diagrama de classes UML da estrutura do pacote criptográfico da API Java Card `javacardx.crypto` combinado com nossa implementação do protocolo criptográfico ECDH. Nele podemos notar que as classes básicas relativas a criptografia assimétrica estão adaptadas para um esquema de assintauras digitais, onde estão previstos os métodos de criação da assinatura na chave privada e a verificação e assinatura na chave pública.

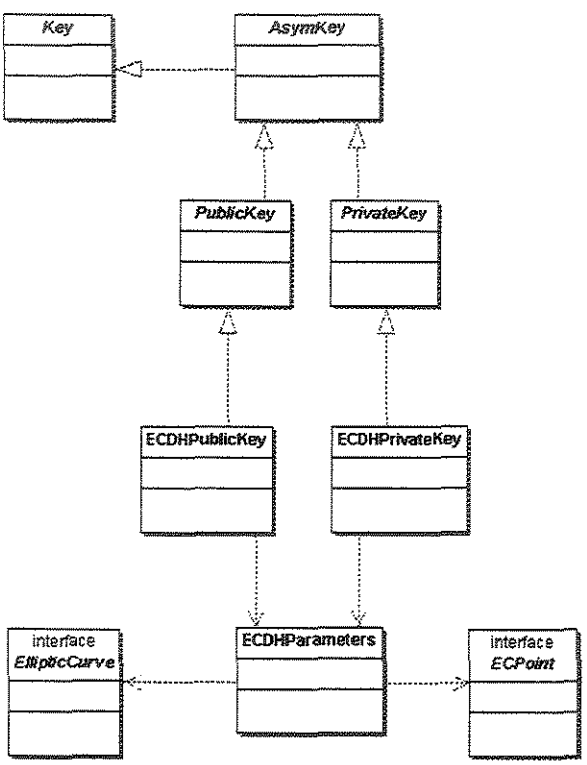


Figura 5.5: Diagrama de classes UML para o protocolo criptográfico ECDH.

A nossa modelagem teve como principal diretriz a criação de um menor número de

objetos o possível. Para tanto, foi implementado o *pattern* **Singleton** [20] na classe que representa os parâmetros do protocolo, garantindo assim que todas as instâncias de chaves de ECDH utilizadas no sistema fariam referência ao mesmo objeto de parametrização.

5.3.2 Geração das Chaves

A geração das chaves é feita pela classe responsável pela parametrização do sistema criptográfico. Primeiramente, é obtido um número a partir da classe de geração de números aleatórios disponibilizado na API (`javacardx.crypto.RandomData`). Este número é então multiplicado pelo elemento base do subgrupo da curva elíptica através do algoritmo de multiplicação escalar. Caso o resultado tenha sido diferente do ponto no infinito, o número aleatório é armazenado como sendo a chave privada e o ponto obtido na multiplicação escalar é armazenado como chave pública.

5.3.3 Estabelecimento de Valor Secreto

Este procedimento diz respeito ao estabelecimento de um valor conhecido somente por duas entidades, A e B, através de um canal de comunicação onde o sigilo não é garantido. Para tanto, uma das partes, A, obtém uma cópia autêntica da chave pública de B e aplica nela o algoritmo de multiplicação escalar usando a chave secreta de A como entrada. A entidade B deve fazer o mesmo procedimento com a chave pública de A. Para que este procedimento seja executado com sucesso é necessário que as duas entidades utilizem os mesmos parâmetros de configuração. Ao final do processo as duas entidades estarão com o mesmo ponto em mãos, onde basta que seja acordado o uso de uma das coordenadas do ponto como valor secreto.

Capítulo 6

Resultados Obtidos

O desenvolvimento da biblioteca de criptografia baseada em curvas elípticas procurou atender dois requisitos principais no desenvolvimento de aplicações para smart cards: não deveria ser utilizada uma quantidade grande de memória e o tempo de execução não deveria ser elevado. A primeira restrição devia-se, principalmente, ao elevado custo de memória e conseqüente impacto na produção em massa de cartões. Já a segunda restrição vinha do fato de que a maioria das aplicações que fazem uso de smart cards são de tempo real, onde há um cliente aguardando as respostas do sistema. O projeto da biblioteca também teve como objetivo promover facilidades de extensão e reuso, principalmente através do uso de *design patterns*.

Para validar nossos resultados, decidimos verificar esses dois requisitos empregando uma plataforma de Java Card consolidada no mercado. Assim, optamos por empregar o produto Cyberflex Access 16k, da Schlumberger, que é uma das empresas líderes no desenvolvimento de aplicações e na fabricação de Java Cards. Esta plataforma, apesar de ser consolidada, contém algumas restrições em sua API que influenciaram significativamente em nossos resultados. Apesar disto, o experimento se mostrou válido no que diz respeito à viabilidade de desempenho da construção de uma biblioteca desta natureza.

A Seção 6.1 mostra uma visão detalhada dos requisitos estabelecidos para o projeto. Já a Seção 6.2 apresenta o Cyberflex Access, que foi usado como nossa plataforma de testes. Uma vez que as limitações de nossa plataforma de testes influenciaram o desenvolvimento

das funcionalidades criptográficas, na Seção 6.3 são tratadas estas deficiências.

A medição de tempos de execução é tratada nas seções 6.4 e 6.5. A primeira descreve a metodologia empregada na obtenção dos resultados e a segunda mostra os tempos obtidos. Na Seção 6.6 são feitas considerações sobre os resultados obtidos de acordo com os requisitos estabelecidos para o projeto.

6.1 Requisitos do Projeto

O objetivo principal do projeto foi a construção de uma biblioteca de funcionalidades criptográficas empregando a tecnologia de ECC em smart cards. Esta biblioteca deveria obedecer os requisitos comumente impostos a aplicações construídas para estes dispositivos, principalmente com relação à memória utilizada. Esta restrição foi imposta principalmente devido ao fato de encontrarmos Java Cards com 8k a 16k de memória EEPROM, compartilhada tanto por dados como por aplicações. Assim, uma biblioteca muito grande iria limitar a possibilidade de utilização do cartão, uma vez que não haveria memória suficiente para que suas aplicações pudessem ser carregadas e para o armazenamento dos dados dos usuários.

Outro requisito importante diz respeito ao tempo de resposta. Uma vez que grande parte das aplicações para smart cards devem apresentar resultados em tempo real, o desempenho da biblioteca se tornou um fator determinante para o sucesso da mesma. O desempenho esperado para uma biblioteca criptográfica nestes dispositivos é fortemente influenciado pela geração de assinaturas digitais, operação mais empregada na prática, com tempo esperado de menos de um segundo. As outras funcionalidades criptográficas apresentam requisitos de desempenho semelhantes.

Além dos requisitos de espaço de armazenamento e tempo de resposta, destacamos também a necessidade de realizar uma modelagem de classes que permita uma boa integração da biblioteca criptográfica com as demais funcionalidades da API do Java Card. A necessidade de uso de mecanismos avançados de modelagem, tais como *design patterns*, advém do fato dos projetistas da API fazerem grande uso deles, provendo assim grande facilidade de extensão e reuso.

6.2 Escolha da Plataforma de Java Card

A escolha da plataforma de Java Card utilizada em nossos testes foi baseada nos seguintes critérios:

- Versão da API.
- Quantidade de memória EEPROM disponível.
- Representatividade no mercado.

Quanto ao primeiro critério, verificamos que a versão mais recente da API é a 2.1.2, já com a publicação do *draft* pela Sun da versão 2.2. No entanto, na época em que o cartão foi adquirido, a versão da API era a 2.1 e no mercado só era possível encontrar cartões com a versão 2.0.

Dentre os diversos fabricantes, dois dominavam o mercado, a Gemplus e a Schlumberger. Uma vez que a iniciativa do desenvolvimento do Java Card partiu do segundo fabricante, optamos por adquirir o seu produto, o Cyberflex Access 16K, também por apresentar um tamanho de EEPROM maior que os outros produtos do mercado.

6.3 Limitações da Plataforma de Testes

A plataforma de Java Card utilizada em nossos continha limitações que, de certa forma, prejudicaram o nosso desenvolvimento. A primeira foi o uso de uma versão antiga da API. Apesar do produto ter sido adquirido no começo de 2001 e a versão 2.1 ter sido lançada cerca de um ano antes, só havia cartões disponíveis com a versão 2.0. Este fato teve impacto principalmente na modelagem dos serviços criptográficos, uma vez que na versão 2.1 houve uma revisão no pacote criptográfico com a aplicação de padrões de projeto para facilitar a extensão das funcionalidades do pacote.

Já com relação ao desempenho de nossa implementação, um problema encontrado na plataforma foi a utilização de *arrays*. O método `JCSystem.makeTransientByteArray` têm como finalidade forçar que o *array* seja criado em memória RAM, que é cerca de 1000 vezes mais rápida que a memória EEPROM. Desta forma, o uso desta função para

a criação dos registradores de trabalho das classes poderia melhorar consideravelmente o desempenho dos algoritmos. No entanto, segundo as notas de *release* do produto [49], este método não tem o funcionamento esperado e deve ser evitado.

Os métodos `JCSystem.arrayCopyNonAtomic` e `JCSystem.arrayFillNonAtomic`, que poderiam ser usados para a cópia e inicialização dos valores, respectivamente, também apresentaram problemas. Uma vez que estes métodos estavam sempre gerando uma exception `arrayIndexOutOfBounds`, a recomendação feita nas notas de *release* é substituir a chamada destes métodos por uma estrutura de laço. Este problema não permitiu o uso de métodos nativos para a inicialização e cópia em memória que estariam disponíveis nos métodos citados, o que resultaria em um ganho adicional de desempenho.

6.4 Arquitetura de Testes

A arquitetura construída para os testes consistiu de duas partes, uma que operava dentro do cartão, fazendo uso das bibliotecas disponibilizadas, e outra num PC conectado a um CAD. Uma vez que os testes foram voltados mais para a aritmética no corpo finito, a parte interna dos testes era constituída de um *applet* com três elementos da aritmética básica obtidos através das fábricas disponibilizadas e oferecia os serviços de obtenção do número de bytes utilizado pelos elementos do corpo, armazenamento e recuperação de informações relativas aos elementos do corpo finito. Além disso, foram disponibilizados métodos para solicitar a realização das operações previstas na interface de elemento do corpo finito com o resultado destas operações sendo retornado em resposta às solicitações.

A parte que executou no PC foi construída com base nas APIs disponibilizadas pelo kit de desenvolvimento da Schlumberger. Optamos por desenvolver a parte PC dos testes usando a linguagem de programação C++ pois os mecanismos de medição de tempos disponíveis são mais precisos que os encontrados em Java. O aplicativo tinha como tarefa inicial estabelecer uma sessão com o *applet* e obter os parâmetros de configuração da aritmética, que consistia basicamente do número de bytes esperados.

O teste de corretude foi feito com o auxílio de uma biblioteca de aritmética em corpos finitos disponibilizada em [40]. As operações aritméticas foram realizadas sobre valores

aleatórios, tanto no cartão quanto na biblioteca.

Já para o teste de desempenho, o primeiro passo foi estimar o tempo de comunicação necessário para o envio e recebimento das informações. Esta estimativa é relevante pois o tempo de transferência de informações é um dos conhecidos gargalos de desempenho dos smart cards. Porém, uma vez que as funcionalidades disponibilizadas têm como alvo *applets* executando dentro do cartão, não há o custo de comunicação. Assim, em nossas medições o tempo médio de comunicação foi subtraído do tempo total de cálculo.

Os tempos necessários para a realização das operações foram então obtidos através do envio de diversos valores obtidos aleatóriamente e a realização de um número determinado de operações. A quantidade de operações realizadas para a medição dos tempos foi determinada pela ordem de grandeza de cada operação.

6.5 Medição dos Tempos da Aritmética no Corpo Finito

Uma vez definida a arquitetura de testes, realizamos o teste das implementações desenvolvidas. O critério utilizado para os casos de testes foi a escolha de corpos finitos com um número de bits onde seria obtido um nível de segurança aceitável para as aplicações. Na atualidade, os requisitos de segurança das aplicações podem ser satisfeitos com o uso de chaves de 160 bits para curvas elípticas, que são equivalentes em termos de segurança a chaves RSA de 1024 bits.

Para o corpo $\mathbb{F}_{2^m}$, foram medidos os tempos para os corpos definidos pelos trinômios

$$x^{167} + x^6 + 1$$

e seu recíproco

$$x^{167} + x^{161} + 1.$$

A Tabela 6.1 mostra as medições de tempo do corpo para as duas abordagens. Nela podemos constatar que o algoritmo de multiplicação de López e Dahab apresentou melhor performance que o algoritmo de multiplicação padrão, atingindo cerca de 50% do tempo.

Corpo Finito	Soma	Mult Lopez & Dahab	Mult. Stand	Quad	Inv
$x^{167}+x^6+1$	0.16209	24.93969	51.86129	1.87029	95.92379
$x^{167}+x^{161}+1$	0.13524	26.00734	50.21509	2.43444	85.87054

Tabela 6.1: Tempos obtidos (em segundos) na aritmética sobre o corpo finito $\mathbb{F}_{2^{167}}$.

Corpo Finito	Soma	Multiplicação	Inversão
OEF	0.21303	8.45663	37.14042
$\mathbb{F}_{2^{167}}$ $(x^{167}+x^6+1)$	0.16209	24.93969	95.92379
$\mathbb{F}_{2^{167}}$ $(x^{167}+x^{161}+1)$	0.13524	26.00734	85.87054

Tabela 6.2: Comparativo entre os tempos obtidos (em segundos) em OEF com $\mathbb{F}_{2^m}$.

Nossa implementação de OEF supunha que o número primo utilizado estaria na forma 2^n+c. Assim, como na arquitetura o maior tipo era o *short*, com 16 bits, os valores escolhidos para n e c foram 13 e -1, respectivamente. Por apresentar vantagens computacionais, o valor de w foi escolhido como sendo 2 e o valor de m igual a 13. O polinômio irredutível de OEF ficou sendo então

$$8191^{13}-2,$$

resultando assim em um corpo com $mn = 169$ bits.

A Tabela 6.2 mostra os tempos obtidos por OEF em comparação aos tempos de $\mathbb{F}_{2^{167}}$ (foram usados os tempos da multiplicação de Lopez e Dahab). Apesar do tempo da soma em OEF ser maior que os tempos em $\mathbb{F}_{2^{167}}$, o tempo obtido na multiplicação foi cerca de 3 vezes menor que os tempos nos outros corpos.

6.6 Avaliação dos Resultados

A construção da biblioteca de criptografia baseada em curvas elípticas teve em mente principalmente os requisitos de utilização de pouca memória e de tempo de execução. Uma vez que o favorecimento de um dos requisitos acaba por prejudicar o outro, buscamos encontrar um ponto de equilíbrio entre eles. Desta forma, evitamos o uso de valores

pré-calculados e o recurso de expansão de laços para prover menor uso de memória, salvo na implementação do algoritmo de cálculo de quadrado em $\mathbb{F}_{2^m}$, onde adotamos um número de valores pré-calculados compatível com nossos requisitos de memória. Assim, conseguimos obter uma biblioteca com cerca de 3K bytes de tamanho, o que representa cerca de 18.75% do total de memória de nossa plataforma de testes. Este tamanho de biblioteca é equiparável a implementações de mesma natureza encontradas na literatura, como em [25].

Os tempos obtidos nas operações feitas na aritmética básica demostraram que não seria possível atender ao requisito da geração de uma assinatura digital em menos de um segundo. Isto devido ao fato de uma operação no grupo elíptico desencadear dezenas ou até mesmo centenas de operações no corpo finito. Como cada operação no corpo finito necessita de mais de um segundo, seriam necessários vários minutos para que um protocolo criptográfico fosse executado completamente. Assim, nossa implementação tornou-se inviável para fins práticos.

Ainda observando os tempos, pudemos constatar que estes resultados de performance foram fortemente influenciados pelas limitações da arquitetura de testes, que dispunha de um processador de 8 bits operando a uma freqüência de 5 MHz. Testes em um desktop mostraram bons resultados em termos de desempenho, o que nos mostra que um aumento no poder de processamento dos cartões poderia viabilizar a utilização desta biblioteca.

Os requisitos de modelagem foram alcançados através do uso de *design patterns* e com o foco em interfaces. Testes mostraram que o advento destes níveis de abstração na modelagem não adicionaram um *overhead* significativo no desempenho da biblioteca. Assim, o uso destas técnicas de modelagem, que promovem a facilidade de extensão da arquitetura, apresentam vantagens na construção de bibliotecas desta natureza.

Capítulo 7

Conclusão

Em nosso projeto, tínhamos como objetivo a construção de uma biblioteca de serviços criptográficos para smart cards. Uma vez que a criptografia baseada em curvas elípticas possui características de economia de memória e de poder de processamento, tornando-as alternativas atraentes para dispositivos limitados, decidimos empregá-la em nossa implementação.

A escolha da plataforma de desenvolvimento foi influenciada pelo mercado. Segundo nossas observações, havia uma tendência (e ainda há) do uso de cartões que suportassem multiplas aplicações e baseados em linguagens de programação conhecidas, ao invés de arquiteturas e linguagens proprietárias com uma única aplicação executando no cartão. Desta forma, havia duas plataformas candidatas, o MultiOS e o Java Card. A primeira é uma arquitetura baseada em linguagem de programação C++ e a segunda é baseada em Java. Os dois têm em comum o uso de um interpretador para executar seus códigos binários dentro do cartão. Uma vez que o mercado estava preferindo o uso do Java Card, que hoje é um padrão desta indústria, decidimos adotar esta tecnologia.

Foi então estabelecido para o projeto a construção de uma biblioteca portável de ECC para o Java Card. A implementação e o sucesso de uma biblioteca desta natureza é fortemente influenciado por dois fatores: o tamanho de memória necessária e o tempo de execução. Assim, buscamos implementar algoritmos que não necessitassem de grandes quantidades de memória (evitamos valores pré-calculados), buscando um equilíbrio entre

os dois requisitos principais. Com relação aos requisitos de desempenho da biblioteca, havia um risco muito grande de não conseguirmos antendê-los satisfatoriamente, principalmente devido a utilização de um interpretador e os diversos mecanismos de segurança definidos na especificação do Java Card, que causam um *overhead* significativo.

A base da implementação de ECC é a aritmética no corpo finito; desta forma, procuramos privilegiar o seu tratamento. Em um trabalho encontrado na literatura [18], verificamos que o uso de corpos de característica prima já haviam sido explorados, sem sucesso. Esta abordagem foi então descartada pois os algoritmos para este tipo de corpo finito fazem uso intensivo do *carry bit* do processador, o que não estaria acessível em uma linguagem de alto nível como Java. Adotamos então os corpos de característica par ($\mathbb{F}_{2^m}$) e os corpos OEF.

O passo seguinte foi a escolha entre o uso de corpos finitos específicos ou o uso de uma estrutura parametrizável. A primeira alternativa permite que características especiais sejam exploradas para a obtenção de melhor desempenho, porém, sem a flexibilidade da segunda alternativa. Uma terceira abordagem, sugerida em [40], era o uso de um gerador de código capaz de analisar as características do corpo a ser usado e adaptar-se a ele. Em nossa implementação, optamos pela construção dos geradores de código.

Procuramos também manter nosso modelo de classes flexível, de modo a possibilitar que a mudança na parametrização dos corpos pudesse ser incorporada às implementações com um custo reduzido. Para isto, foram utilizados alguns dos padrões de projetos para linguagens orientadas a objetos mais populares, tais como fábricas de objetos e *singleton*. Esta abordagem também serviu para acompanharmos a evolução da API, uma vez que as versões mais recentes têm procurado a incorporação destes conceitos. Os princípios utilizados na modelagem da aritmética sobre o corpo finito foram utilizados também na aritmética da curva elíptica. O protocolo criptográfico foi adequado para a forma sugerida na API, apesar da versão 2.0 não apresentar a flexibilidade da versão 2.1.

Os testes nos mostraram que conseguimos atender o requisito de espaço de armazenamento, com uma biblioteca com cerca de 3K bytes de tamanho. No entanto, o desempenho da biblioteca não foi satisfatório o suficiente para que fosse viável sua utilização prática. Verificamos, no entanto, que o problema estava na arquitetura do Java Card, uma vez

que os tempos obtidos pelo algoritmo de soma em $\mathbb{F}_{2^m}$ se mostraram extremamente elevados. Apesar disto, obtemos tempos significativamente melhores com $\mathbb{F}_{2^m}$ e OEF que os da implementação de corpos de característica prima [18], o que nos mostrou que este é o caminho mais indicado para implementações em software. No entanto, será necessário aguardar que ocorram avanços no poder de processamento dos smart cards para que seja possível a construção de bibliotecas que usem operações de baixo nível no Java Card.

7.1 Trabalhos Futuros

- Adequação da implementação da versão 2.0 da API para a versão 2.1.1 ou mesmo para a versão 2.2, que está para ser lançada. Nestas versões, a interface das classes referentes ao criptossistema RSA permitem, aparentemente, uma utilização mais flexível de suas funções. Assim, seria possível aproveitar o co-processador aritmético do RSA, presente na maioria dos produtos Java Card, para implementar aritmética em $\mathbb{F}_p$.

- Adequar os algoritmos para que façam uso dos métodos que permitem a alocação de memória diretamente em RAM, uma vez que em nossa plataforma de testes esta funcionalidade não estava disponível. Com isso, é possível obter ganhos em tempos de acesso a memória da ordem de 1000 vezes. No entanto, deve ser avaliado se o *overhead* de controle a ser realizado nesta estratégia é compensado pelo ganho em desempenho.

- Verificar o desempenho da aritmética nas novas plataformar de Java Card lançadas no mercado, que sofreram um aumento em poder de processamento em relação ao produto empregado em nossos testes.

Glossário

AIA (Almost Inverse Algorithm) – Algoritmo de cálculo do inverso multiplicativo em um corpo finito.

AID (Application Identifier) – Número único atribuído a uma instância de *applet* ou um pacote Java.

APDU (Application Protocol Data Unit) – Pacote de dados trocado entre o smart card e o terminal na camada de aplicação. Um APDU contém ou um comando ou uma resposta.

API (Application Programming Interface) – Define as interfaces e convenções pelas quais uma aplicação acessa funcionalidades disponíveis em bibliotecas de serviços.

ATR (Answer to Reset) – Conjunto de bytes enviados por um smart card após uma condição de reinício.

CAD (Card Acceptance Device) – Dispositivo que é usado para a comunicação com o smart card. Ele também provê energia e o sinal de sincronismo para o smart card.

CAP (Converted Applet) – Padrão de formato de arquivo binário que provê compatibilidade entre diferentes plataformas de Java Card. Um arquivo CAP contém a representação binária executável das classes em um pacote Java.

DES (Data Encryption Standard) – Algoritmo de criptografia simétrico com chave de 56 bits utilizado como padrão pelo governo dos EUA.

DSA (Digital Signature Algorithm) – Algoritmo utilizado pelo governo dos EUA, em conjunto ao SHA-1, na constituição de seu padrão de assinaturas digitais.

ECAES (Elliptic Curve Authenticated Encryption Scheme) – Protocolo de ciframento e deciframento de mensagens construído sobre um sistema de curvas elípticas.

ECC (Elliptic Curve Cryptography) – Sistemas criptográficos de chave pública baseados na dificuldade de cálculo do logaritmo discreto sobre um grupo de pontos definidos sobre uma curva elíptica.

ECDH (Elliptic Curve Diffie Hellman) – Protocolo de estabelecimento de uma chave secreta criado por Diffie e Hellman construído sobre um sistema de curvas elípticas.

ECDLP (Elliptic Curve Discrete Logarithm Problem) – Problema de calcular o logaritmo discreto em um grupo formado sobre um conjunto de pontos de uma curva elíptica.

ECDSA (Elliptic Curve Digital Signature Algorithm) – Variação do algoritmo DSA baseado no ECDLP.

EEPROM (Electrically Erasable Programmable Read-Only Memory) – Meio de armazenamento persistente tipicamente empregado em smart cards.

EMV – Consórcio das empresas Europay, Mastercard e Visa com o objetivo de estabelecer padrões de smart cards específicos para seus ramos de atividades.

GF(2^m) – Corpo finito (de Galois) sobre 2^m. Equivalente à notação $\mathbb{F}_{2^m}$.

GSM (Global System for Mobile communication) – Novo padrão para telefonia celular digital que prevê o uso de smart cards. Este padrão foi adotado inicialmente na Europa e autualmente está sendo implantado em outras regiões do mundo.

ISO 7816 – Principal documento de padronização da indústria de smart cards.

JCVM (Java Card Virtual Machine) – Máquina virtual responsável por executar as aplicações no Java Card. É constituída de um conversor que executa fora do cartão em um PC ou numa *workstation* e o interpretador que executa dentro do cartão.

JCRE (Java Card Runtime Environment) – Ambiente no qual os *applets* do Java Card são executados.

JRE (Java Runtime Environment) – Ambiente no qual as aplicações Java são executadas.

JVM (Java Virtual Machine) – Máquina virtual responsável por executar as aplicações Java.

MAC (Message Autentication Code) – Número anexado à uma mensagem para que seu receptor seja capaz de verificar sua integridade.

MAIA (Modified Almost Inverse Algorithm) – Variação do algoritmo AIA que provê melhor desempenho.

OCF (OpenCard Framework) – Conjunto de funcionalidades que provêem uma interface padronizada para aplicações em PCs ou *workstations* interagirem com os CADs e aplicações nos smart cards.

OEF (Optimal Extension Field) – Corpo do tipo $\mathbb{F}_p$, onde o valor de p é um primo pseudo-Mersene, que possibilita implementações eficientes de sua aritmética.

PIN (Personal Identification Number) – Número usado para autenticar um determinado usuário.

RSA – Algoritmo de criptografia de chave pública mais usado atualmente, onde seu nome é composto pelas iniciais de seus autores: Rivest, Shamir e Adleman. Sua segurança é baseada na dificuldade da fatoração de números inteiros muito grandes.

SHA-1 (Secure Hash Algorithm) – Algoritmo de *hash* criptográfico que foi projetado por NIST e NSA para ser utilizado com o *Digital Signature Standard*. O padrão é o *Secure Hash Standard*; SHA é o algoritmo utilizado pelo padrão. O SHA produz um hash de 160 bits.

UML (Unified Modeling Language) – Linguagem de modelagem que tem se tornado padrão de mercado. Esta linguagem define um conjunto de diagramas e notações para a modelagem de sistemas baseados no paradigma de orientação a objetos.

3DES – Variação do DES em que este algoritmo é aplicado três vezes para aumentar sua segurança.

Bibliografia

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem. IEEE P1363a Submission, Available at `http://grouper.ieee.org/groups/1363/P1363a/Encryption.html#ABR`, 1998.

[2] ANSI X9.62: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). *American National Standards Institute*, 1999.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. The Java series. Addison-Wesley, 1996.

[4] Ash, Blake, and Vanstone. Low complexity normal bases. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 25, 1989.

[5] D. Bailey and C. Paar. Optimal extension feld for fast arithmetic in public key algorithms. In *Advances in Cryptology - CRYPTO '98*, volume 1462 of *LNCS*. Springer-Verlag, 1998.

[6] Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[7] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Ad-*

vances in Cryptology - CRYPTO '86, volume 263 of *LNCS*, pages 311–323, Berlin, Germany, August 1986. Springer-Verlag.

[8] I. Blake, G. Seroussi, and N. Smart. *Elliptic curves in cryptography.* Cambridge University Press, 1999.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language - User Guide.* Object Technology Series. Addison-Wesley, 1998.

[10] Certicom. The Elliptic Curve for Smart Cards. `http://www.certicom.com/ecc/wecc4.htm`, April 1998.

[11] Y. L. Chan and H. Y. Chan. Java Smart Cards. `http://www.iis.ee.ic.ac.uk/~frank/surp98/report/ylc3/report2.html`, 1998.

[12] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* The Java series. Addison-Wesley, 2000.

[13] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent # 5,159,632, October 1992.

[14] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[15] Bull-Personal Transaction Systems Division. Smart Card's World. `http://www.cp8.bull.net/download/scworld.pdf`, 1998.

[16] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[17] Tommi Elo. Lessons learned on implementing ECDSA on a Java smart card. `citeseer.nj.nec.com/elo00lessons.html`.

[18] Tommi Elo. Implementing ECDSA on a Java Smart Card. Master's thesis, Helsinki University of technology, March 2000.

[19] European Telecomunications Standards Institute. `http://www.etsi.org/`.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1994.

[21] Rinaldo Di Giorgio. Smart Cards: A Primer. `http://www.javaworld.com/javaworld/jw-12-1997/jw-12-javadev_p.html`, December 1997.

[22] GlobalPlatform. `http://www.globalplatform.org/`.

[23] Louis Claude Guillou, Michel Ugon, and Jean-Jacques Quisquater. The Smart Card—A Standardized Security Device Dedicated to Public Cryptology. In Gustavus J. Simmons, editor, *Contemporary Cryptology—The Science of Information Integrity*, pages 561–613. IEEE Press, 1992.

[24] Darrel Hankerson, Julio Lopez Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. In *Proceedings of Cryptographic Hardware and Embedded Systems*, pages 1–24, 2000.

[25] Hasegawa, Nakajima, and Matsui. A small and fast software implementation of elliptic curve cryptosystems over GF(p) on a 16-bit microcomputer. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1999.

[26] Julio César López Hernández. *Implementação Eficiente em Software de Criptossistemas de Curvas Elípticas.* Tese de doutorado, IC–Unicamp, Campinas, SP, Março 2000.

[27] IBM. IBM Smart Card Solutions Elements—Technical Overview, July 1997.

[28] Sun Microsystems Inc. Java Card Technology. `http://www.java.sun.com/products/javacard/index.html`, 1999.

[29] International Organization for Standardization. `http://www.iso.ch/`.

[30] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $gf(2^m)$. *Information and Computation*, 78:171–177, 1998.

[31] Donald E. Knuth. *The Art of Computer Programming*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1981.

[32] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[33] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 15–19 August 1999.

[34] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Key hashing for message authentication. RFC 2104, IETF, February 1997.

[35] Júlio López and Ricardo Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Cryptographic Hardware and Embedded Systems—CHES '99*, volume 1717 of *LNCS*, pages 316–327, 1999.

[36] Júlio López and Ricardo Dahab. An Overview of Elliptic Curve Cryptography. Technical Report IC-00-10, IC – Unicamp, May 2000.

[37] R. J. McElience. *Finite Fields for Computer Scientsts and Engineers*. Kluwer Academic Puclishers, 1987.

[38] Alfred J. Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[39] Victor Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology, Crypto '85*, volume 218 of *LNCS*, pages 417–426, New York, 1986. Springer-Verlag.

[40] Rogério Albertoni Miranda. Criptossistemas Baseados em Curvas Elípticas. Dissertação de mestrado, Instituto de Computação, UNICAMP, 2002.

[41] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[42] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal Normal Bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1989.

[43] NIST FIPS PUB 186-2: Digital Signature Standard (DSS). *National Institute for Standards and Technology*, 2000.

[44] National Institute of Standards and Technology. Secure Hash Standard. FIPS Publication 180-1, 1995.

[45] OpenCard consortium. `http://www.opencard.org/`.

[46] PC/SC Workgroup. `http://www.pcscworkgroup.com/`.

[47] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, 1997.

[48] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Comm. ACM*, 21:120–126, 1978.

[49] Schlumberger, Austin, Texas, USA. *Cyberflex Access Software Developer's Kit – Release Notes*, September 2000.

[50] Richard Schroeppel, Hilarie Orman, Sean W. O'Malley, and Oliver Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology – CRYPTO '95*, pages 43–56, Germany, 1995. Springer-Verlag.

[51] SECG SEC 1: Elliptic curve cryptography. *Standards for Efficient Cryptography Group*, 1999.

[52] SECG SEC 2: Recommended Elliptic Curve Domain Parameters. *Standards for Efficient Cryptography Group*, 1999.

[53] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag New York Inc., 1986.

[54] N. P. Smart. A comparison of different finite fields for use in elliptic curve cryptosystems. *Computers and Mathematics with Applications*, 42(?):91–100, October 2001.

[55] J. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Dept. of C& O, University of Waterloo, 1999.

[56] Sun Microsystems Inc. *Java Card 2.0 Application Programming Interfaces*, October 1997. Revision 1.0 Final.

[57] Sun Microsystems Inc. *Java Card 2.0 Programming Concepts*, October 1997. Revision 1.0 Final.

[58] Sun Microsystems Inc. *Java Card 2.1 Application Programming Interfaces*, February 1999. Revision 1.0 Final.

[59] Sun Microsystems Inc. *Java Card 2.1.1 Specifications – Application Programming Interfaces*, May 2000. Revision 1.0 Final.

[60] Sun Microsystems Inc. *Java Card 2.1.1 Specifications – Release Notes*, May 2000. Revision 1.0 Final.

[61] Sun Microsystems Inc. *Java Card 2.1.1 Virtual Machine Specification*, May 2000. Revision 1.0 Final.

[62] Michel Ugon. Smart Card Odyssey. `http://www.cardshow.com/guide/card/odyssey.html`, April 1998.

[63] S. Vanstone. Responses to NIST's proposal. *Communications of the ACM*, 35:50–52, July 1992.

[64] WAP: WTLS: Wireless Transport Layer Security Specification. *Wireless Application Forum Ltd*, 1999.

[65] A. Woodbury, D. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *IFIP CARDIS 2000 – Fourth Smart Card Research and Advanced Application Conference*, pages 20–22, Bristol, UK, September 2000. Kluwer.