FUNDAÇÃO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS ZANCO LADEIRA

UTILIZAÇÃO DE CRIPTOGRAFIA HOMOMÓRFICA PARA AUTENTICAÇÃO EM JAVA CARD

Marília

2015

LUCAS ZANCO LADEIRA

UTILIZAÇÃO DE CRIPTOGRAFIA HOMOMÓRFICA PARA AUTENTICAÇÃO EM JAVA CARD

Trabalho de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília - UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador
Profº Dr. Fábio Dacêncio Pereira

Marília
2015

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
MANTIDO PELA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Lucas Zanco Ladeira

TÍTULO: UTILIZAÇÃO DE CRIPTOGRAFIA HOMOMÓRFICA PARA AUTENTICAÇÃO EM JAVA CARD.

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 9,5 (nove meio)

Orientador: Fábio Dacêncio Pereira

1º.Examinador: Rodolfo Barros Chiaramonte

2º.Examinador: Jussara Mallia Zachi

Marília, 30 de novembro de 2015.

Ladeira, Lucas Zanco

Utilização de Criptografia Homomórfica para Autenticação em Java Card/ Lucas Zanco Ladeira; orientador: Prof. Dr. Fábio Dacêncio Pereira, SP: [s.n.], 2014.

64 folhas

Monografia (Bacharelado em Ciência da Computação): Centro Universitário Eurípedes de Marília.

Dedico este trabalho a minha família, pelo apoio durante o curso, e em toda a minha vida.

AGRADECIMENTOS

Agradeço primeiramente a minha família pelo apoio durante o curso, e por me incentivar mesmo quando foi difícil superar os desafios encontrados. Além disso, foi importante pelo aconselhamento para buscar conhecimento, dentro e fora das aulas. Principalmente os meus pais que me ajudaram a entender a área que desejo seguir, e quais os planos para o meu futuro.

Agradeço a instituição de ensino pelas oportunidades apresentadas de aprendizado seguindo o que é ensinado nas principais faculdade brasileiras, e desenvolvimento de projetos de iniciação de ensino. Também foram apresentadas vagas de emprego frequentemente caso o foco do aluno seja de trabalhar em uma empresa.

É necessário agradecer aos professores que incentivaram a busca por conhecimento, se esforçando para ensinar, e estando a disposição para tirar dúvidas mesmo fora do horário de aula. Em especial ao professor Fábio Dacêncio Pereira, que me orientou durante a iniciação científica e trabalho de conclusão de curso. Onde, foi possível desenvolver trabalho de ponta, com o apoio de bolsa da FAPESP. Essa instituição de apoio à pesquisa científica foi fundamental para que fosse possível a dedicação ao projeto todos os dias, possibilitando fazer o uso dos equipamentos da faculdade para desenvolvimento do mesmo.

Por fim, agradeço a minha namorada Ligia Maria Fantin Salvi, que me apoiou durante quase todo o curso, me ouvindo durante as dificuldades, e me fazendo sorrir. Sei que seria muito diferente sem ela, com certeza não teria crescido tanto, e devo agradecer demais a ela.

“A habilidade suprema não consiste em ganhar cem batalhas, mas sim em vencer o inimigo sem combater.” – Sun Tzu – A Arte da Guerra

RESUMO

O intuito do projeto é utilizar um algoritmo de criptografia homomórfica para autenticar um usuário com uma aplicação em *Java Card*. O dado utilizado na autenticação é o PIN, uma senha de oito caracteres, que deve ser mantida em segredo durante transmissão ao cartão. Com a exposição da senha seria possível o acesso a qualquer serviço implementado dentro do *applet*, podendo acarretar perdas financeiras, acesso a locais prioritários, ou até mesmo vazamento de informações. O esquema escolhido para o projeto é o de Coron et al, onde o mesmo é um esquema totalmente homomórfico. Fazendo uso de apenas algumas das primitivas implementadas: *keygen*, *encrypt*, *decrypt* e *expand*. Dessa maneira, como o intuito do projeto não é de implementar o esquema homomórfico citado, foi feito o uso de um algoritmo em python criado por Santos (2014). Dessa maneira apenas a decodificação do PIN deve ser executada no *applet*, tornando possível autenticar o usuário.

Palavras chave: Java Card, Criptografia Homomórfica, Autenticação, PIN, Decodificação.

ABSTRACT

The purpose of the project is to use homomorphic encryption algorithm to authenticate a user at a Java Card application. The data used to authenticate is the PIN, is a password of eight characters that must be kept secret during transmission to the card. With an exposed password would be possible to access any service implemented within the applet, it may result in financial losses, access to priority places, or event lead to leak of information. The scheme chosen for the project is from Coron et al, and it is a fully homomorphic scheme. Using just some of the primitives implemented: keygen, encrypt, decrypt and expand. The objective of the project is not to implement the full homomorphic scheme, so was used the algorithm created by Santos (2014). This way only a decryption of the PIN shall be executed by the applet, making possible to authenticate the user.

Keywords: Java Card, Homomorphic Cryptography, Autentication, PIN, Decryption.

LISTA DE ILUSTRAÇÕES

Figura 1: Triad CIA. Fonte: ANDRESS, J., 2014, p. 28. Tradução nossa. 18
Figura 2: Processo de compartilhamento e funcionamento de chaves simétricas. Fonte: BILAR (2014). 22
Figura 3: Processo de compartilhamento e funcionamento de chaves assimétricas. Fonte: BILAR (2014). 23
Figura 4: Esquema de Bootstraping. Fonte: SANTOS (2014). 25
Figura 5: Principais esquemas totalmente homomórficos. Fonte: SANTOS, L. C., 2014. 26
Figura 6: Código referente a primitiva Keygen parte 1. Fonte: SANTOS, 2014. 34
Figura 7: Código referente a primitiva Keygen parte 2. Fonte: SANTOS, 2014. 34
Figura 8: Código referente a primitiva Encrypt. Fonte: SANTOS, 2014. 35
Figura 9: Código referente a primitiva Decrypt. Fonte: SANTOS, 2014.......................... 35
Figura 10: Código referente a primitiva Expand. Fonte: SANTOS, 2014. 36
Figura 11: Código referente a primitiva Evaluate. Fonte: SANTOS, 2014. 36
Figura 12 - *Java Card Virtual Machines*. Fonte: CHEN (2000, p.31)............................... 38
Figura 13 - Diagrama da Arquitetura 50
Figura 14 - Gráfico sem homomorfismo... 54
Figura 15 - Gráfico com homomorfismo 55
Figura 16 - VerifyPinSk. Fonte: própria... 61
Figura 18 - SendDecimalPlaces. Fonte: própria... 62
Figura 19 - CountDecimalPlaces. Fonte: própria.... 62
Figura 17 - ExecutePlusSkExpand. Fonte: própria. 63
Figura 20 – ExecuteMenusSum. Fonte: própria.... 64

LISTA DE ABREVIATURAS E SIGLAS

SHE	Criptografia Parcialmente Homomórfica (Inglês: *Somewhat Homomorphic Encryption*)
FHE	Criptografia Totalmente Homomórfica (Inglês: *Fully Homomorphic Encryption*)
LWE	Aprendendo com Erros (Inglês: Learning with Errors)
JCRE	Ambiente de Execução Java Card (Inglês: *Java Card Runtime Environment*)
JDK	Kit de Desenvolvimento Java (Inglês: *Java Development Kit*)
JCDK	Kit de Desenvolvimento Java Card (Inglês: *Java Card Development Kit*)
APDU	Protocolo de Aplicação em Unidade de Dado (Inglês: *Application Protocol Data Unit*)
AID	Identificador de Aplicação (Inglês: *Application Identifier*)
RID	Identificador de Recurso (Inglês: *Resource Identifier*).
PIX	Extensão Identificadora de Proprietário (Inglês: *Proprietary Identifier Extension*)
NFC	Comunicação com Campo Próximo (Inglês: *Near Field Communication*)

SUMÁRIO

Introdução 15
Motivação e Justificativa 15
Objetivos Gerais 16
Organização deste documento 16
1 Contextualização 18
1.1 Segurança da Informação 18
1.2 Criptografia 20
1.2.1 Criptografia Simétrica 21
1.2.2 Criptografia Assimétrica 22
1.2.3 Funções de *Hash* 23
1.2.4 Criptografia Homomórfica 23
1.3 Smart Card 27
1.4 Java Card 27
1.5 Considerações finais do capítulo 28
2 Fundamentação teórica 30
2.1 Criptografia Homomórfica Esquema DGHV 30
2.2 Esquema DGHV com Chave Pública Reduzida 30
2.2.1 Primitiva *KeyGen* $(\mathbf{1\lambda})$ 31
2.2.2 Primitiva *Encrypt* (pk, m∈{0,1}) 32
2.2.3 Primitiva Decrypt (sk, c) 32
2.2.4 Primitiva *Evaluate* (pk, C, c1, ...,ct) 32
2.2.5 Primitiva *Expand* (pk, c) 32
2.2.6 Primitiva *Recrypt* (pk, c, z) 32
2.3 Implementação Esquema DGHV com Chave Reduzida 33
2.3.1 *Keygen* 34
2.3.2 *Encrypt* 35
2.3.3 *Decrypt* 35

2.3.4 *Expand* ... 36
2.3.5 *Evaluate (add, mult)* ... 36
2.3.6 Parâmetros Utilizados ... 36
2.3.7 Análise do Código ... 37
2.4 Tecnologia Java Card ... 37
2.4.1 API *Java Card* ... 38
2.4.2 Protocolo de Comunicação ... 40
2.4.3 Estrutura de um Applet ... 40
2.4.4 Segurança da Plataforma ... 42
3 Materias e metodos ... 44
3.1 Materias Utilizados ... 45
4 Trabalhos Correlatos e Implementações ... 46
4.1 Método Utilizando Algoritmo de Criptografia Homomórfica One-way no Cenário de Comunicação em Redes ... 46
4.2 Modelo de Transação Seguro e de Custo Eficaz para Serviços Financeiros ... 47
5 Desenvolvimento ... 49
5.1 Implementação ... 50
5.2 Testes ... 53
5.2.1 Sem Homomorfismo ... 53
5.2.2 Com Homomorfismo ... 54
5.2.3 Tempo de decodificação no *Smart Card* ... 56
6 Conclusão ... 57
Referências ... 58
APÊNDICE A – Códigos da Decodificação – Applet ... 61
1 VerifyPinSk ... 61
3 SendDecimalPlaces ... 61
2 ExecutePlusSkExpand ... 62

4 ExecuteMenusSum 63

INTRODUÇÃO

A segurança da informação está em constante desenvolvimento devido a necessidade de garantir a confidencialidade durante o envio de dados ou armazenamento dos mesmos. Esses dados podem ser desde anotações em uma agenda, até mesmo dados pessoais e senhas bancárias. Sendo assim, uma das técnicas utilizadas é a conversão de um texto claro para uma cifra fazendo o uso de uma chave. Sendo essa técnica nomeada criptografia (ANDRESS, 2014, p. 93).

A criptografia simétrica faz o uso de uma mesma chave para codificar e decodificar dados. Já a criptografia assimétrica utiliza um par de chaves, pública e privada, sendo uma para codificar e outra para decodificar.

No entanto, fazendo uso do algoritmo de Shor (1994) em computadores quânticos é possível derivar a chave pública e obter a chave privada. Isso torna possível a decodificação de qualquer informação transmitida onde foi feito o uso de uma das chaves. Para resolver esse problema uma classe de algoritmos criptográficos foram criados chamados algoritmos pós quânticos. Nos mesmos não é possível executar o algoritmo de Shor (1994) e obter o resultado correto.

Dentro do grupo de algoritmos pós quânticos é possível citar esquemas de criptografia homomórfica, onde a principal características é a capacidade de executar uma operação matemática em um dado codificado. Esses esquemas podem ser classificados em dois tipos: totalmente homomórficos e parcialmente homomórficos. Os totalmente homomórficos podem executar a operação de adição e multiplicação em um dado codificado, enquanto o parcialmente homomórfico apenas uma das duas operações é possível.

Motivação e Justificativa

Autenticação é algo que está presente em vários *applets* (Aplicações para *Java Cards*), como por exemplo em qualquer cartão de crédito ou débito, e é utilizada para acessar contas bancárias, sendo com a utilização de um PIN ou da biometria. Além disso, *smart cards* podem armazenar dados pessoais de usuários, documentos confidenciais, entre outras informações que devem ser mantidas em segredo. Sendo assim, é necessário garantir a segurança durante a transmissão de qualquer informação de um leitor para o cartão e vice-versa. Para garantir a segurança algoritmos criptográficos são utilizados, podendo também criar sessões seguras, e validar a autenticidade de ambas as entidades que tentam se comunicar.

No entanto, é necessário analisar a complexidade da segurança aplicada nessa tecnologia, pois pela quantidade de memória disponível pode não ser possível fazer o uso da mesma. Um exemplo ocorre no algoritmo pós quântico de Coron (CORON et. al, 2011), com chave pública de grandeza $\boldsymbol{\Theta}(\boldsymbol{\lambda}^7)$. Para exemplificar essa grandeza é levado em consideração o nível *small* de segurança, nesse nível o valor referente ao parâmetro $\boldsymbol{\lambda}$ é 52. Em um cenário onde o *java card* possui 80kb de memória, o mesmo não tem a capacidade de armazenar essa chave, e nem de executar qualquer operação que a utilize. Mesmo assim, existem operações onde não é feito o uso da chave pública, como por exemplo a decodificação.

Dessa maneira, ao associar a autenticação em um *java card* com a criptografia homomórfica, o dado utilizado na autenticação estaria seguro. Mesmo que, durante o envio ao cartão, um atacante obtenha a cifra ele não terá condições de descobrir qual o valor real da mesma, e nem de ambas chaves utilizadas na codificação.

Objetivos Gerais

Os objetivos gerais dessa monografia são: implementar uma aplicação, para um *Java Card*, que tenha a capacidade de inicializar a mesma com uma chave privada homomórfica, deve criar uma sessão segura com um gerenciador, e o mais importante, executar a decodificação homomórfica do PIN para autenticar o usuário. Sendo assim, o dado utilizado na autenticação deve ser mantido seguro, e o tempo de execução da aplicação analisado. O esquema homomórfico escolhido foi o proposto por Coron (CORON et. al, 2011), que se baseia no esquema com o nome de DGHV (DIJIK et. al, 2010). Para envio dos dados ao cartão foi utilizado um gerenciador desenvolvido na linguagem Java. Além disso, um código implementado em Python foi utilizado para gerar as chaves, codificar os dados, e expandir a cifra. Esse código foi criado por Santos (SANTOS, 2014), e apresenta código aberto para uso na monografia.

Organização deste documento

Esse projeto foi dividido em 6 partes: primeiramente contextualização, onde os conceitos mais básicos da área onde o projeto está contido são apresentados. Esses conceitos ditam sobre segurança da informação, criptografia, *smart card*, entre outros. Segundo é feita a apresentação da fundamentação teórica, onde os conceitos sobre criptografia homomórfica e *java card*, são explicados de forma mais aprofundada. Então, são citados autores que utilizam

as tecnologias apresentadas no projeto, parte do trabalho dos mesmos é explicado. Após isso, é apresenta a metodologia utilizada no desenvolvimento do projeto, como também os materiais. Sendo esses: bibliotecas de desenvolvimento, hardware, aplicações, entre outros. Então em quinto lugar é apresentado o desenvolvimento em si, e os resultados obtidos através de testes na aplicação resultante do trabalho. Por último, é feita a conclusão sobre todo o trabalho executado para a finalização da monografia, onde é analisado o objetivo alcançado no projeto.

1 CONTEXTUALIZAÇÃO

O presente capítulo tem como objetivo explicar o básico sobre as tecnologias utilizadas no projeto, e de onde elas se desenvolveram. Isso sem se aprofundar na teoria relativa a cada tópico, como fórmulas matemáticas e explicações técnicas do processo de criptografia de cada algoritmo citado. Dessa forma, pretende-se delimitar o contexto deste projeto.

1.1 Segurança da Informação

Por definição segurança da informação significa "proteger informações e sistemas de informação de acesso não autorizado, uso, divulgação, interrupção, modificação, ou destruição" (*US Government, Legal Information Institute*). Dessa forma, visa a proteção dos dados de um indivíduo, sejam dados bancários, ou anotações. De acordo com Andress (2014, p. 28) a segurança da informação possui três características principais: integridade, confidencialidade, e disponibilidade. As mesmas podem ser observadas na figura 1.

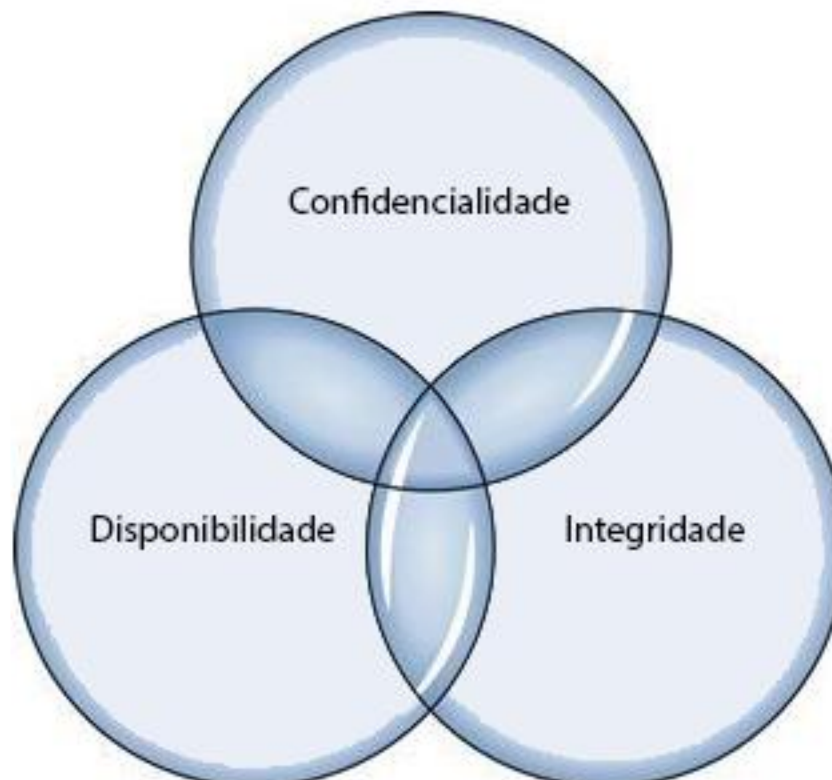

Figura 1: Triad CIA. Fonte: ANDRESS, J., 2014, p. 28. Tradução nossa.

De acordo com Cartilha de Segurança para Internet (Cert.br, 2015), além dos aspectos citados anteriormente, também podem ser mencionadas identificação, autenticação, autorização e não repúdio.

O estudo da segurança da informação teve seu início devido ao crescimento do acesso a computadores, nas casas ou locais de trabalho, conectados à internet (RUIU, 2006). E também devido à crescente quantidade de vulnerabilidades descobertas, algumas delas como *buffer-overflow exploit*, execução remota de *shell-script*, comprometimento de relações confiáveis, entre outras. Alguns desses ataques possuíam a capacidade de roubar informações ou até desmantelar o sistema operacional. Com isso, foi criada a necessidade de constante pesquisa sobre métodos de defesa, mantendo sempre atualizados, e formas de identificar ataques em tempo real e os responsáveis pelos mesmos.

Existem várias maneiras de se proteger contra ataques, por exemplo com a utilização de *firewall*, antivírus, *antispyware*, entre outras. As formas citadas são softwares que analisam o tráfego da rede, os processos do computador, e os arquivos armazenados em disco para verificar se algum atacante está tentando obter informações (PAULI, 2013).

Além disso, uma outra maneira de se conseguir alguma senha ou dado pessoal é comunicando-se diretamente com o alvo, prática que recebe o nome de "Engenharia Social". No meio empresarial é possível a aquisição de dados passando-se por algum superior requisitando acesso a aplicações ou senhas de usuários. Para manter dados seguros é necessária a conscientização popular para reconhecer possíveis falsários (BHAKTA e HARRIS, 2015).

A criptografia é uma maneira de assegurar a confidencialidade de informações durante o envio pela internet ou armazenamento. Algoritmos criptográficos executam a codificação de um dado, utilizando uma ou mais chaves, e apenas quem estiver apto poderá decodificar o texto cifrado e descobrir o texto claro. Dos aspectos da segurança da informação citados anteriormente a criptografia compreende: confidencialidade, integridade, autenticação e não repúdio.

Segundo a Cartilha de Segurança para Internet, a confidencialidade significa "proteger uma informação contra acesso não autorizado". A criptografia garante esse aspecto pela capacidade de cifrar um dado, tornando o mesmo secreto, e depois decifrá-lo.

A integridade significa "proteger a informação contra alteração não autorizada". A mesma é garantida pela criptografia utilizando funções de *hash*. Com essas funções é possível identificar a mudança em um texto codificado executando a mesma função sobre o texto claro e verificando os resultados. Além disso, é possível verificar a alteração em um dado levando

em consideração a característica dessa função de codificar um texto claro em tamanho fixo. Autenticação refere-se a “verificar se a entidade é realmente quem ela diz ser”. A entidade pode ser um processo, software, sistema, ou computador. Isso ocorre depois da identificação, para verificar se o computador é reconhecido de forma autentica. Esse aspecto pode ser alcançado pela utilização de uma assinatura digital (Cert.br – Cartilha de Segurança para Internet, 2015).

O conceito de não repúdio é “evitar que uma entidade possa negar que foi ela quem executou uma ação”. Com isso é possível identificar certamente o autor de ações não desejadas a informações ou serviços. Essas ações podem compreender envio de requisições a serviços ou até mesmo envio de dados comprometidos. Dentro da área da criptografia, utilizando o certificado digital, esse aspecto é garantido (Cert.br – Cartilha de Segurança para Internet, 2015).

Além dos serviços citados a criptografia também garante a comprovação temporal. Esse serviço tem como objetivo garantir que uma determinada informação existia em um determinado momento, mesmo que no instante da verificação da mesma, ela possa ter sido excluído.

1.2 Criptografia

Segundo Andress (2014, p.93), “criptografia é a ciência de manter informação segura” (tradução nossa). A mesma pode ser dividida essencialmente em 3 etapas: geração de chave(s), codificação e decodificação. A geração de chave(s) tem a função de produzir a(s) chave(s) que será(ão) utilizada(s) nos processos de codificação e decodificação. Codificação torna possível transformar um texto claro em texto cifrado. A decodificação é o processo de recuperar o texto claro por intermédio da cifra.

Um exemplo antigo da utilização da criptografia é a Cifra de Cesar, utilizada por Júlio Cesar. Essa cifra funciona da seguinte maneira: era feita a modificação das letras para a distante de um determinado número. Digamos que é necessário o envio de uma mensagem. Uma letra dessa mensagem é A, e o número escolhido foi 3, portanto a letra que será colocada na mensagem no lugar dessa será D. Da mesma maneira, mas em sentido oposto, é feita a conversão da cifra para o texto claro. Apenas quem souber a quantidade de letras que deve contar saberá decifrar a mensagem (BLAIR, 2013).

Criptografias não tão antigas foram de grande ajuda em guerras para enviar mensagens cifradas. Um exemplo, a máquina Enigma, criada por alemães, foi vastamente utilizada na

segunda guerra mundial. Essa máquina utilizava rodas com valores das letras que deveriam ser configuradas de forma idêntica para transmissão correta das mensagens. Essas mensagens serviram para manter escondidos posicionamento de tropas, e locais que seriam atacados (ANDRESS, 2014, p.97).

A base da criptografia é a utilização de problemas matemáticos complicados para cifrar e decifrar algum dado. Levando em consideração a evolução e a expansão da aplicação desses algoritmos no dia a dia, foi feita a divisão em dois grandes tipos: criptografia simétrica, e criptografia assimétrica. Cada um deles possui suas diferenças como quantidade de chaves utilizadas e organização dos blocos de bytes.

1.2.1 Criptografia Simétrica

A criptografia simétrica utiliza apenas uma chave para codificar e decodificar um dado. Dessa forma, ambas as partes que desejam se comunicar devem possuir a mesma chave. Caso o atacante consiga interceptar a chave ele terá a capacidade de decifrar todo o dado transmitido, criar modificações das mensagens reais, cifrá-las e enviar para qualquer uma das partes.

Esse tipo de criptografia pode ser dividido em dois tipos: bloco e *stream*. O tipo de bloco divide o dado cifrado em vários blocos de tamanhos iguais. Já *stream* cifra cada bit da mensagem separadamente. Segundo Andress, J. (2014, p. 99) os algoritmos simétricos em bloco são mais utilizados que os de *stream*. Pois, os algoritmos de *stream* necessitam de mais tempo para executar, e possuem maior chance de ocorrer erros pela maior quantidade de bits utilizados durante a codificação e decodificação. Alguns exemplos bastante utilizados de algoritmos de bloco são: DES (*Data Encryption Standard*) e AES (*Advanced Encryption Standard*).

A figura 2 exemplifica um processo simples de criptografia simétrica. Nela o indivíduo Alice deseja enviar uma mensagem ao indivíduo Bob. Primeiramente ocorre a troca de uma chave secreta por um meio seguro. Após isso, Alice cifra a mensagem com a chave secreta compartilhada e envia ao Bob. O mesmo, utilizando a chave secreta, decifra a mensagem e tem conhecimento do texto claro.

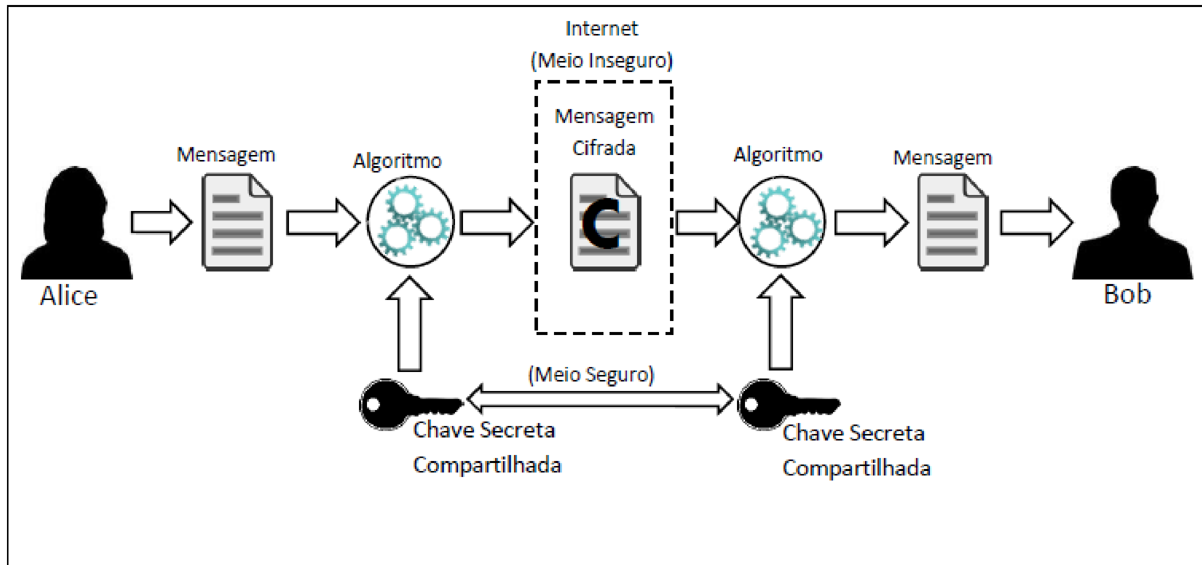

Figura 2: Processo de compartilhamento e funcionamento de chaves simétricas. Fonte: BILAR (2014).

1.2.2 Criptografia Assimétrica

A criptografia assimétrica, diferente da simétrica, utiliza duas chaves: uma chave privada e uma chave pública. A chave privada é utilizada para decifrar o dado, e deve ser de conhecimento apenas do sistema computacional que gerou o par de chaves.

A chave pública é utilizada para cifrar o dado. A mesma possui esse nome pois qualquer sistema que deseja se comunicar com outro via mensagens cifradas poderá requisitá-la. Ela é de conhecimento público, podendo ser disponibilizada sem qualquer segurança. Essa é uma vantagem da criptografia assimétrica sobre a simétrica, utilizando a chave pública não é necessário consumir meios seguros para envio da chave (ANDRESS, 2014, p.100).

Um algoritmo de chave assimétrica mundialmente conhecido é o RSA. Ao utilizar esse algoritmo é necessário seguir algumas especificações chamadas PKCS (*Public-key Cryptography Standards*). Essas especificações propõem normas para criação do par de chaves criptográficas, estabelecimento de protocolo de comunicação, certificação de documentos, entre outras (EMC, RSA Laboratories, 2015).

A figura 3 exemplifica o processo de troca de mensagens utilizando criptografia assimétrica. Nessa imagem Alice deseja enviar uma mensagem ao Bob. Para isso Bob gera um par de chaves e disponibiliza à Alice a chave pública. Então ela, utilizando a chave pública, cifra a mensagem e envia ao Bob. Ele, utilizando a chave privada, decifra a mensagem e tem conhecimento do texto claro.

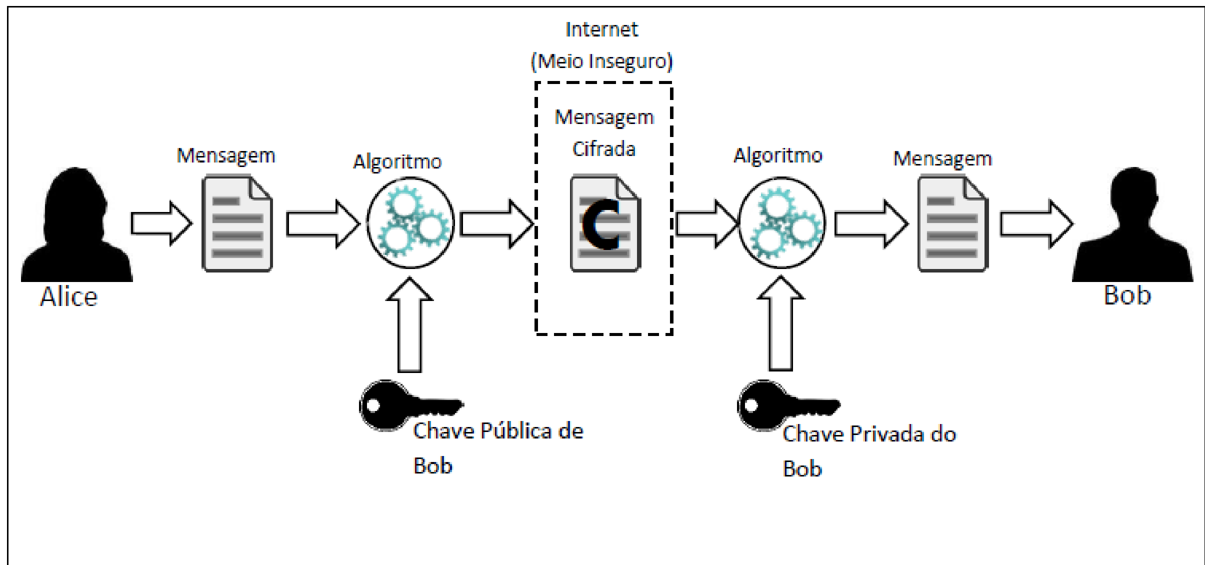

Figura 3: Processo de compartilhamento e funcionamento de chaves assimétricas. Fonte: BILAR (2014).

1.2.3 Funções de *Hash*

Conforme San-um e Srichavengsup (2013) explicam, uma função *hash* é "uma tecnologia chave na criptografia avançada, que codifica um dado de entrada de tamanho arbitrário em um valor *hash* de tamanho fixo" (tradução nossa). Essas funções podem ser divididas em: com chaves, e sem chaves. As funções com chaves utilizam uma chave criptográfica para criar o *hash* do texto claro. A outra, como o nome já diz, não necessita de chave para codificar.

Alguns algoritmos podem ser citados, como MD-5, SHA-1, SHA-2 e SHA-3. Esses são bastante difundidas na indústria e utilizadas para garantir a segurança na comunicação, validação de dados para verificar modificação, e em protocolos de segurança ou comunicação. Levando em consideração essa característica de grande utilização, é possível observar um grande interesse de *hackers* sobre essa tecnologia. Utilizando-se de vulnerabilidades os atacantes tem a capacidade de inserir dados dentro do *hash*, descobrir o texto claro cifrado (como por exemplo em uma função de autenticação), entre outros (SAN-UM e SRICHAVENGSUP, 2013).

1.2.4 Criptografia Homomórfica

Levando em consideração o avanço acelerado da tecnologia, sendo possível executar operações que alguns anos atrás demorariam horas em segundos, existe uma preocupação com

o estudo da computação quântica. De acordo com Hamdi (et. al, 2014), utilizando um computador quântico a criptografia atual não serviria ao seu propósito com o emprego do Algoritmo de Shor, que tem a capacidade de derivar a chave pública e obter a chave privada. É importante ressaltar que é possível fazer o uso desse algoritmo apenas em alguns algoritmos assimétricos como por exemplo o RSA, e em nenhum simétrico.

Todavia, não é possível utilizar o Algoritmo de Shor em esquemas de criptografia homomórfica para recuperar o valor da chave privada. Dessa maneira, a criptografia homomórfica recebe a classificação de pós-quântica. Com essa característica se torna notável a importância da utilização e no desenvolvimento dessa criptografia.

A principal característica da criptografia homomórfica é a capacidade de executar operações em um dado cifrado. Essa característica permite que esse dado seja enviado a outro computador, junto com a chave pública, e esse execute operações no dado sem ter conhecimento de qual é o valor real (BILAR, 2014).

1.2.4.1 Histórico da Criptografia Homomórfica

A primeira noção de criptografia homomórfica foi descrita no trabalho de Rivest, Adleman,e Dertouzos (1978). Nele é apresentada matematicamente a capacidade de execução de operações em dados cifrados, utilizando o exemplo de informações armazenadas em um banco de dados no meio empresarial.

Após isso, apareceram algumas implementações de SHE (do inglês: *Somewhat Homomorphic Encryption*, do português: Criptografia Parcialmente Homomórfica), com a capacidade limitada de operações em um dado cifrado. Essa capacidade limitada deve-se a acumulação de ruído no dado cifrado a cada operação. Podem ser citados os trabalhos de Paillier (1999), Goldwasser e Micali (1982).

Mas, apenas em 2009 foi publicada uma implementação de FHE (do inglês: *Fully Homomorphic Encryption*, do português: Criptografia Totalmente Homomórfica), criada por Gentry (2009). Nessa implementação Gentry partiu de um esquema SHE que utilizava reticulados, e depois o modificou para atingir um esquema FHE. Uma das características do mesmo é a de que ele tem a capacidade de executar uma avaliação do próprio circuito de decodificação. Além disso, uma parte da chave secreta é adicionada à chave pública, e por causa dessa adição é possível decifrar o dado cifrado por um polinômio de baixo grau.

Dessa forma, foi possível reduzir o ruído no texto cifrado, tornando arbitrária a quantidade de vezes que é possível executar operações em um texto codificado. A função de

reduzir o ruído em um texto cifrado foi chamada de *Bootstraping*. Essa função é ilustrada na figura 4.

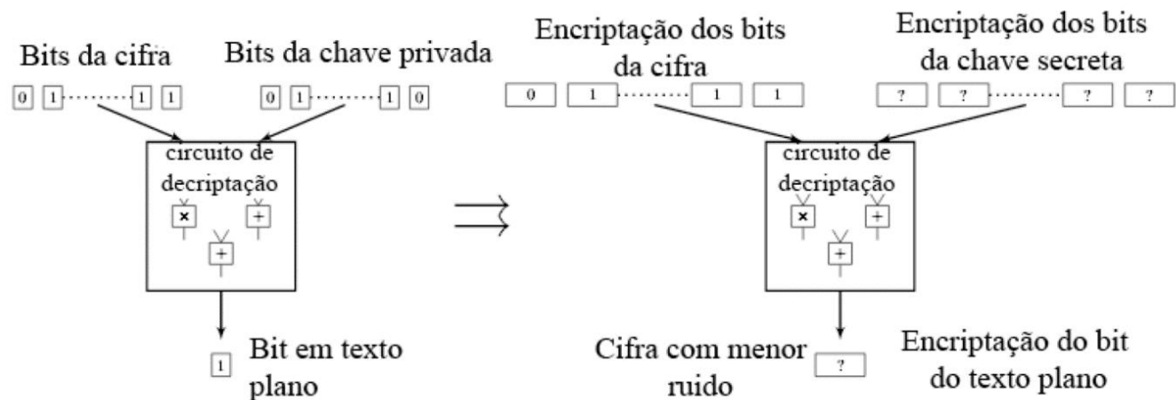

Figura 4: Esquema de Bootstraping. Fonte: SANTOS (2014).

Em 2011, na Eurocrypt, Gentry e Halevi (2011) apresentaram uma nova implementação de um esquema totalmente homomórfico. Nesse esquema foram introduzidas algumas melhorias como, por exemplo, foi o primeiro esquema que o determinante não necessitava ser primo para geração de chaves. Além dessa, foram apresentadas otimizações e simplificações em funções.

1.2.4.2 Classificação dos Algoritmos de Criptografia Homomórfica

Esse tipo de criptografia possui tanto algoritmos simétricos como assimétricos, e a escolha de qual utilizar depende do objetivo da aplicação. Além disso, os algoritmos de criptografia homomórfica podem ser divididos em dois grupos que são: Criptografia Totalmente Homomórfica (FHE) e Criptografia Parcialmente Homomórfica (SHE). O FHE é um algoritmo que tem a capacidade de executar as operações matemáticas de adição e de multiplicação em um dado cifrado. Uma diferença para o algoritmo SHE é a de que esse tem a capacidade de executar apenas uma das operações em um dado codificado (HAN, 2012).

A criptografia FHE pode ser subdividida, ressaltando suas principais vertentes, em LWE (do inglês: Learning With Errors, do português: Aprendendo com erros) que é composto de problemas matemáticos sobre aprendizagem de máquina. Outra subdivisão é de anéis de inteiros, que utiliza um conjunto de inteiros para formação das chaves criptográficas. E por último, reticulados, esse tipo é utilizado pelos primeiros esquemas de autoria de Gentry (2009). Na Figura 5 pode ser observada essa divisão claramente, exemplos de implementações de cada e ano de aparição.

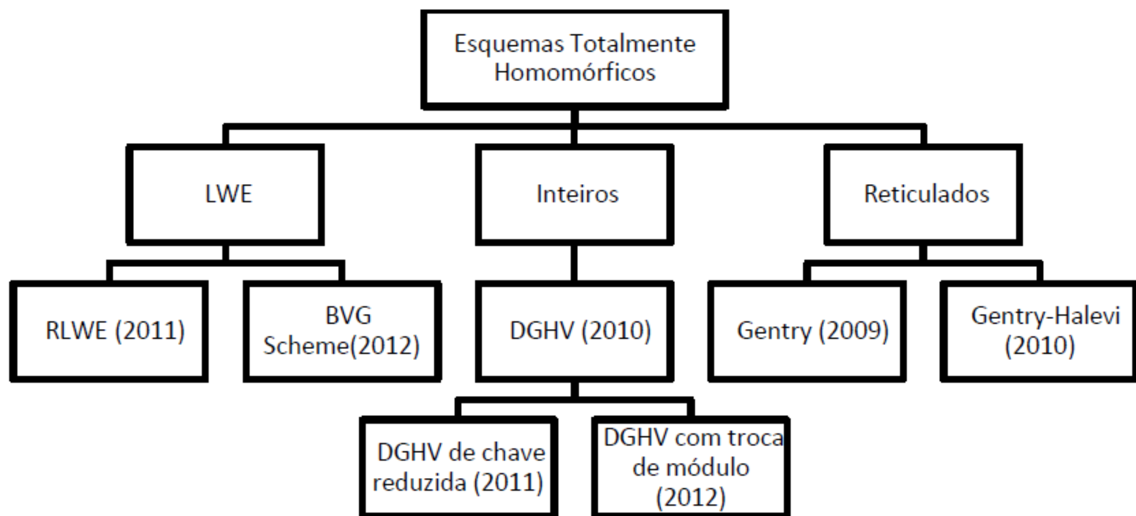

Figura 5: Principais esquemas totalmente homomórficos. Fonte: SANTOS, L. C., 2014.

1.2.4.3 Utilização da Criptografia Homomórfica

Os algoritmos de criptografia homomórfica existentes são geralmente utilizados em máquinas com grande capacidade de armazenamento, processamento e transmissão. Isso se deve ao tamanho das chaves geradas, que dependendo do algoritmo, como por exemplo o de Dijk (et. al, 2010), pode chegar a $\boldsymbol{\Theta}(\boldsymbol{\lambda}^{10})$.

Além disso, existem pesquisas sobre a utilização de hardware dedicado (FPGA) para execução desses algoritmos, dessa forma ocorre um aumento na velocidade de execução. Também podem ser encontradas implementações que utilizam processamento paralelo em GPUs, assim melhorando o desempenho do algoritmo (BERK, et al, 2012).

Existem várias aplicações para algoritmos de criptografia homomórfica, um exemplo é o de sistemas de votação. Nesse sistema o voto é cifrado no terminal de votação e enviado a um servidor, no mesmo todos os votos são somados e, ao final da votação, é decifrado para verificação do resultado final (PENG e BAO, 2010).

Além desse exemplo, levando em consideração que toda computação é feita a partir de um conjunto de somas, qualquer tipo de processamento pode ser executado. Um exemplo é o de execução de algoritmos que requerem grande capacidade de processamento em servidores em nuvem, dessa forma o servidor deve ser robusto para ser capaz de executar a operação requisitada. Outro exemplo é o de armazenamento de dados em nuvem criptografados de forma homomórfica, dessa forma apenas o usuário que cifrou os dados poderá obtê-los posteriormente na forma de texto claro.

O presente trabalho está posicionado em sentido oposto ao da utilização de computadores com grande poder computacional para execução de algoritmos homomórficos. Nele é necessária a execução da decodificação em um *Smart Card*. Essa tecnologia não possui capacidade de armazenar uma chave pública ou o dado cifrado inteiro. Além disso, a capacidade de processamento é bastante limitada, aumentando o tempo de execução do algoritmo.

1.3 Smart Card

CHEN, Z. (2000, p.3) descreve *Smart Card* como sendo:

> "Um *Smart Card* é um computador portátil e inviolável. Ao contrário de cartões magnéticos, *smart cards* transportam tanto poder de processamento e informação. Além disso, eles não precisam de acesso a bases de dados remotas no momento de uma transação."

Esse tipo de cartão possui processador, memória e sistema de entrada e saída. Tendo todos os componentes necessários para execução de funções, armazenamento de dados e comunicação com o mundo exterior. Isso o torna auto suficiente, sendo que apenas necessita de energia, introduzida pelos seus conectores, e as funções implementadas já podem ser chamadas para execução. Algumas vantagens dessa tecnologia que devem ser ressaltadas são: segurança, uma vez que não necessita de recursos externos para executar as suas funções, e tamanho reduzido, tornando fácil o armazenamento do mesmo (CHEN, 2000, p.4).

Essa tecnologia é amplamente empregada no dia a dia, como por exemplo em cartões de crédito, vale refeição, vale transporte, acesso a serviços / lugares, entre outros. De acordo com a *Smart Card Alliance (2015)*, o mercado de *smart card* espera um crescimento de 10% comparado ao ano anterior, sendo vendidos cerca de 8.790 bilhões de dispositivos.

1.4 Java Card

Um *java card* é um *smart card* que possui uma máquina virtual Java embutida. Essa máquina virtual é dividida em duas, uma máquina virtual dentro do cartão, e uma máquina virtual de fora do cartão. A máquina virtual de fora do cartão possui um conversor, esse tem o objetivo de compilar as aplicações. A que fica armazenada no cartão tem um interpretador com a capacidade de execução das aplicações contidas no cartão (CHEN, 2000, p.31).

Isso possibilita a criação de aplicações para esses cartões na linguagem Java. Essas aplicações são chamados de *Applets*, e possuem sua própria biblioteca (JCDK) de desenvolvimento com tamanho bastante reduzido em comparação com a utilizada em computadores (JDK). Em um mesmo cartão é possível armazenar vários *applets*, e utilizá-los sem que um interfira na execução de outro. Isso é possível através do *Applet Firewall* que faz o gerenciamento dos recursos utilizados por cada aplicação. Além disso, esse mecanismo fornece proteção contra tentativas de *hacking* (CHEN, Z., 2000, p.105).

O *Applet Firewall* é utilizado para proteger e separar aplicações e seus respectivos contextos. Ele adota uma divisão de contexto entre cada aplicação, e os coloca em um grupo de contexto caso possuírem o mesmo pacote. Dentro desse grupo os dados são protegidos, e qualquer tentativa de acesso a algum recurso de outro grupo é negada. Além disso, dentro de um *java card*, apenas um contexto se mantém ativo por vez, dessa forma apenas uma aplicação pode estar em execução por vez.

Apenas através da interface *Shareable* aplicações têm a capacidade de acessar dados ou serviços de outras. Dessa forma, tudo o que está dentro da classe que implementou essa interface está disponível para qualquer outra aplicação acessar. Esse mecanismo tem a capacidade de ignorar as restrições do *firewall* e não possui tratamento de contexto. Um exemplo da utilização dessa interface é dentro de um *applet* utilitário com vários métodos de tratamento de dados, conversão de tipos, atributos para padronização de exceções, entre outros (CHEN, 2000, p.112).

Os *applets* possuem a capacidade de acessar serviços do próprio JCRE (*Java Card Runtime Environment*) utilizando um espaço chamado de *Global Arrays*. Esses *arrays* armazenam dados públicos e apenas a JCRE tem a capacidade de criá-los.

1.5 Considerações finais do capítulo

Considerando a importância da criptografia homomórfica para o avanço do estudo da segurança da informação, é necessário ressaltar que quanto mais simples for a utilização, levando em consideração o tamanho das chaves criptográficas, dados cifrados e implementações de algoritmos, mais será utilizada essa criptografia. Consequentemente o problema ressaltado nesse capítulo sobre a computação quântica não surtirá efeito, e assim será possível garantir a segurança de aplicações e informações seguindo o avanço da pesquisa na área de computação quântica.

Esse tipo de criptografia é geralmente conhecido pela complexidade da utilização, como na implementação dos algoritmos da mesma, e necessidade de hardware robusto. Dessa forma, adotando um esquema FHE em um *java card* foi possível iniciar testes de execução em hardware com pouco poder computacional.

2 FUNDAMENTAÇÃO TEÓRICA

Após a apresentação dos conceitos utilizados no projeto, é necessário focar nas tecnologias que serão aplicadas no desenvolvimento. Com isso, o objetivo desse capítulo é explicar o esquema de criptografia escolhido, como as fórmulas matemáticas do mesmo, além das característica principais das tecnologias envolvidas no trabalho.

2.1 Criptografia Homomórfica Esquema DGHV

Na conferência Eurocrypt de 2010, Dijk (et. al, 2010) apresentaram uma nova implementação de um sistema FHE nomeada DGHV. Da mesma maneira que Gentry (2009) partiu de um esquema SHE para criar sua implementação FHE utilizando reticulados, o grupo partiu de uma implementação SHE e criou um esquema FHE baseado em anéis de inteiros. Para isso, foi necessário implementar o *Bootstraping* no novo esquema. Mas, mesmo tendo a simplicidade das operações serem feitas utilizando números inteiros, o tamanho da chave pública ainda era de $\boldsymbol{\Theta}(\boldsymbol{\lambda}^{10})$ (GUPTA e SHARMA, 2013).

Contudo, na Eurocrypt de 2011, Coron (et. al, 2011) publicaram uma nova implementação partindo do esquema SHE DGHV de Dijk (et. al, 2010). Como Coron explica "A ideia consiste em armazenar apenas um subconjunto da chave pública e então gerar *on the fly* a mesma combinando os elementos do subconjunto de modo multiplicativo" (tradução nossa). Dessa forma foi possível diminuir o tamanho da chave pública para $\boldsymbol{\Theta}(\boldsymbol{\lambda}^{7})$.

2.2 Esquema DGHV com Chave Pública Reduzida

No esquema criado por Coron (et. al, 2011), algumas definições são seguidas de forma quase idêntica do trabalho de Dijk (et. al, 2010). A seguinte notação é utilizada:

> "Para um número real x, foi denotado por $[x]$, $[x]$ e $[x]$ o arredondamento de $[x]$ para cima, para baixo, ou para o inteiro mais próximo para os inteiros z, p denota-se a redução de z módulo p por $[z]_p$ com -p/2 $< [z]_p \leq$ p/2. Também foi denotado $[z]_p$ por z mod p. Escreve-se f($\boldsymbol{\lambda}$) = Õ((g($\boldsymbol{\lambda}$)) se f($\boldsymbol{\lambda}$) = O(g($\boldsymbol{\lambda}$) log^k g($\boldsymbol{\lambda}$)) para algum k ∈ N" (tradução nossa).

Além disso, os parâmetros do esquema são definidos segundo Santos (2014) como:

- γ é o comprimento em bits de x_i's.
- η é o comprimento em bits da chave secreta p.
- $\boldsymbol{\rho}$ é o comprimento em bits do ruído r_i.
- τ é o número de x_i's na chave pública.
- $\boldsymbol{\rho'}$ é um parâmetro de ruído secundário utilizado para cifrar.

Esses parâmetros devem seguir as restrições citadas a seguir (tradução nossa):

- $\boldsymbol{\rho} = \omega(\log \boldsymbol{\lambda})$ para evitar ataques de força bruta no ruído
- $\boldsymbol{\eta} \geq (2\rho + \alpha) .\Theta(\boldsymbol{\lambda} \log^2 \boldsymbol{\lambda})$ para dar suporte a operações homomórficas para a avaliação do circuito de decodificação reduzido
- $\gamma = \omega(\eta^2.\log \boldsymbol{\lambda})$ para impedir ataques baseados em retículos com aproximação pelo problema de MDC
- $\alpha .\beta^2 \geq \gamma + \omega(\log \boldsymbol{\lambda})$ para a redução da aproximação por MDC
- $\rho' = 2\rho + \alpha + \omega(\log \boldsymbol{\lambda})$ para o parâmetro secundário de ruído

Mas, no esquema de Coron (et. al, 2011) é adicionado um novo parâmetro β. Nesse esquema são utilizados números inteiros x'_{ij} no formato de $x'_{ij} = x_{i,0}.x_{i,1} \bmod x_0$ para $1 \leq i, j \leq \beta$. Portanto, apenas 2β inteiros $x_{i,b}$ são necessários armazenar na chave pública para gerar $\tau = \beta^2$ inteiros x_{ij} utilizados na codificação.

Nesse esquema são utilizadas as seguintes primitivas: *KeyGen* (responsável pela geração de chaves), *Encrypt* (responsável pela codificação), *Decrypt* (faz a decodificação do código), *Evaluate* (operações de adição e multiplicação), *Expand* (expande o texto cifrado), *Recrypt* (operação que diminui o ruído no texto cifrado).

2.2.1 Primitiva *KeyGen* (1^λ)

Primeiramente é gerado um número primo aleatório $p \in \cap [2^{\eta-1}, 2^{\eta})$. Onde q_0 é um número aleatório ao quadrado de 2^{λ} inteiro no intervalo $[0, 2^{\lambda}/p)$, e $x_0 = q_0 \cdot p$. Então gere $x_{i,b}$ para $1 \leq i \leq \beta$ e $b \in \{0,1\}$, onde $q_{i,b}$ são números inteiros aleatórios no intervalo $[0, q_0)$ e $r_{i,b}$ são inteiros no intervalo $(-2^{p}, 2^{p})$. Sendo sk = p e pk = $(x_0, x_{1,0}, x_{1,1}, \dots x_{\beta,0}, x_{\beta,1})$ (CORON, et. al, 2011). Nessa primitiva devem ser seguidos:

$$x_{i,b} = p .q_{i,b} + r_{i,b},\ 1 \leq \beta,\ 0 \leq b \leq 1$$

2.2.2 Primitiva *Encrypt* (pk, m∈{0,1})

Inicialmente é gerado um vetor aleatório b = $(b_{i,j})$ de tamanho $\tau = \beta^2$. Então, gere um inteiro aleatório r no intervalo ($-2^{p'}, 2^{p'}$) (CORON, et. al, 2011). O texto cifrado é formado por:

$$c = m + 2r + 2 \sum_{1 \leq i,j \leq \beta} b_{ij} \cdot x_{i,0} \cdot x_{i,1} \bmod x_0$$

2.2.3 Primitiva Decrypt (sk, c)

Essa primitiva é a mesma descrita na implementação original DGHV. A mesma é utilizada para decodificar um texto cifrado. O texto plano é obtido calculando (c mod p) mod 2. A partir daí, levando em consideração que c mod p = c − p . [c/p] e p é um número ímpar, é possível executar m = $[c]_2 \oplus$ [[c/p]]2 (DIJK, et. al, 2010).

2.2.4 Primitiva *Evaluate* (pk, C, c1, ...,ct)

Como no *Decrypt*, o *Evaluate* é utilizado da mesma forma descrita na implementação DGHV original. É a primitiva que apresenta os métodos para executar operações em um dado cifrado (adição e multiplicação). Para isso é apenas necessário executar todas as somas ou multiplicações nos textos cifrados e depois executar mod x_0 (DIJK, et. al, 2010).

2.2.5 Primitiva *Expand* (pk, c)

A função *Expand* tem como objetivo criar uma representação expandida do texto cifrado. Essa representação é utilizada na função de decodificação do mesmo.

2.2.6 Primitiva *Recrypt* (pk, c, z)

Essa primitiva tem a capacidade de avaliar o circuito de decodificação de forma homomórfica em um texto cifrado. Nesse método é gerado um novo texto cifrado mas com ruído reduzido.

2.3 Implementação Esquema DGHV com Chave Reduzida

O esquema criado por Coron (CORON, et. al, 2011) apresenta uma melhoria significativa no esquema DGHV original, reduzindo o tamanho da chave pública de $\Theta(\lambda^{10})$ para $\Theta(\lambda^{7})$. Com a redução do tamanho da chave ocorre um crescimento na quantidade de aplicações reais que tem a capacidade de suportar tal grau de requerimento de hardware.

Após a pesquisa dos esquemas DGHV e a escolha da utilização do esquema de Coron (CORON, et. al, 2011) com chave reduzida, foi encontrada uma implementação desse esquema criado por Santos (2014). O esquema foi implementado na linguagem Python e apresenta código aberto para ser utilizado nesse trabalho.

Levando em consideração, que o intuito do projeto não é de implementar o esquema de Coron (CORON, et. al, 2011) completo, apenas será necessário implementar no *java card* a primitiva de *decrypt*. Além disso, é possível utilizar o trabalho de Santos (2014) para a maior parte das primitivas. A implementação criada pelo mesmo possui as seguintes primitivas implementadas:

- *Keygen*
- *Encrypt*
- *Decrypt*
- *Expand*
- *Evaluate (add, mul)*

Essas primitivas dão a capacidade de gerar as chaves criptográficas, cifrar um bit, decifrar um bit, e executar operações em um dado cifrado. Dessa forma, o que já está implementado compreende o escopo do projeto de cifrar um dado em um computador e decifrá-lo dentro de um *java card*. Isso torna necessário apenas implementar o código de decodificação na linguagem Java utilizando a API JCDK. O código referente a cada uma das primitivas é apresentado abaixo:

2.3.1 *Keygen*

```
def keygen(file, size='small'):
    P.setPar(size)
    tempo=-time.time()

    p = genZi(P._eta)
    q0, x0 = genX0(p, P._gamma, P._lambda)
    listaX = genX(P._beta, P._rho, p, q0)
    pkAsk = listaX
    pkAsk.insert(0, x0)

    while True:
        s0,s1=genSk(P._theta, P._thetam)
        if (s0.count(1)*s1.count(1)==15): break

    se=int(time.time()*1000)
    _kappa=P._gamma+6
    u11=genU11(se, s0, s1, P._theta, _kappa, p)
    sigma0 = encryptVector(s0, p, q0, x0, P._rho)
    sigma1 = encryptVector(s1, p, q0, x0, P._rho)
    tempo+=time.time()

    public=fheKey.pk(pkAsk, se, u11, sigma0, sigma1, P)
    secret=fheKey.sk(s0, s1, P)
    fheKey.write(public, 'pk_pickle_'+file)
    fheKey.write(secret, 'sk_pickle_'+file)
```

Figura 6: Código referente a primitiva Keygen parte 1. Fonte: SANTOS, 2014.

```
f = open(file, 'w')
f.write(("keygen executado em " +str(tempo) +" segundos"+'\n\n'))
f.write(('p==' + str(p)+'\n\n'))
f.write(('q0==' +str(q0)+'\n\n'))
f.write(('pk*=='+str(pkAsk)+'\n\n'))
f.write(('s0 =='+str(s0)+'\n\n'))
f.write(('s1 =='+str(s1)+'\n\n'))
f.write(('se =='+str(se)+'\n\n'))
f.write(('u11 =='+str(u11)+'\n\n'))
f.write(('sigma0 =='+str(sigma0)+'\n\n'))
f.write(('sigma1 =='+str(sigma1)+'\n\n'))
f.close
print('arquivo salvo', file)
```

Figura 7: Código referente a primitiva Keygen parte 2. Fonte: SANTOS, 2014.

2.3.2 *Encrypt*

```
def encrypt(pk, m):
    if m != 0 and m != 1:
        print('not a valid m')
        return 0

    alpha = pk.P._lambda
    matrix = [[0 for i in range(pk.P._beta)] for j in range(pk.P._beta)]
    for i in range(pk.P._beta):
        for j in range(pk.P._beta):
            matrix[i][j]=random.randint(0, 2**alpha -1)
    pprime=2**pk.P._rho
    r=random.randint(-pprime, pprime)
    pkAsk=(pk.pkAsk).copy()
    x0=pkAsk.pop(0)
    xi0=pkAsk[::2]
    xj1=pkAsk[1::2]

    somatorio=mpz(0)
    for i in range(pk.P._beta):
        for j in range(pk.P._beta):
            x=gmpy2.mul(xj1[j], xi0[i])
            somatorio += gmpy2.mul(matrix[i][j], x)
    c=(mpz(m)+2*r+2*somatorio)%x0
    return c
```

Figura 8: Código referente a primitiva Encrypt. Fonte: SANTOS, 2014.

2.3.3 *Decrypt*

```
def decrypt(sk, cAsk, z):
    soma=0
    l=int(math.sqrt(sk.P._theta))
    for i in range(l):
        for j in range(l):
            soma+=(sk.s0[i]*sk.s1[j]*z[i][j])
    soma=mpz(gmpy2.round_away(soma))
    m=(cAsk-soma)%2
    return m
```

Figura 9: Código referente a primitiva Decrypt. Fonte: SANTOS, 2014.

2.3.4 *Expand*

```
def expand(pk, cAsk):
    l=int(math.sqrt(pk.P._theta))
    y=[[0 for i in range(l)] for i in range(l)]
    kappa=pk.P._gamma+6

    n=2
    gmpy2.get_context().precision=n

    for i in range(l):
        for j in range(l):
            y[i][j]=float(gmpy2.mpfr(randomMatrix(i, j, kappa, pk.se, l)/(2**kappa)))
    y[0][0]=float(gmpy2.mpfr(pk.u11/2**kappa))

    gmpy2.get_context().precision=21000

    expand = [[1 for i in range(l)] for i in range(l)]
    for i in range(l):
        for j in range(l):
            expand[i][j]=float((cAsk * y[i][j])%2)
    return expand
```

Figura 10: Código referente a primitiva Expand. Fonte: SANTOS, 2014.

2.3.5 *Evaluate (add, mult)*

```
def add(pk, c1, c2):
    soma=gmpy2.add(c1,c2)
    return soma%pk.pkAsk[0]

def mul(pk, c1, c2):
    mul= gmpy2.mul(c1,c2)
    return mul%pk.pkAsk[0]
```

Figura 11: Código referente a primitiva Evaluate. Fonte: SANTOS, 2014.

2.3.6 Parâmetros Utilizados

Seguindo o que foi especificado no trabalho de Coron (CORON, et. al, 2011) algumas primitivas são utilizadas para diferenciar níveis de segurança, o que acarreta uma diferenciação no tamanho das chaves, cifra, entre outras. No caso desse projeto, a primitiva escolhida foi a *small*, mas todas são listadas na tabela 1.

Tabela 1. Parâmetros do esquema. Fonte: SANTOS, 2014.

Parâmetro	λ	ρ	η	γ	β	Θ	θ

Toy	42	16	1088	$1,6 \times 10^5$	12	144	15
Small	52	24	1632	$8,6 \times 10^5$	23	533	15
Medium	62	32	2176	$4,2 \times 10^6$	44	1972	15
Large	72	39	2652	$1,9 \times 10^7$	88	7897	15

2.3.7 Análise do Código

A implementação do esquema de Coron (CORON, et. al, 2011), pode ser utilizada para testes de execução possuindo a capacidade de codificar bits e executar as ações de adição, multiplicação, e decodificação nos mesmos. O código, ao executar a primitiva *keygen*, cria arquivos e armazena todos os atributos gerados, como as chaves criptográficas, s0, s1, entre outros. Esses atributos são necessários na execução das outras funções.

Para a execução das primitivas de *encrypt* ou *decrypt* os arquivos que armazenam as chaves pública e privada devem ser lidos e as chaves armazenadas em variáveis para execução das funcionalidades. Então, pode ser chamado o método referente a primitiva que deseja executar. O resultado da execução é apresentado no console.

2.4 Tecnologia Java Card

Diferentemente da máquina virtual Java, a tecnologia *java card* possui duas máquinas virtuais. Uma delas fica armazenada fora do cartão, ela possui um conversor e tem o objetivo de compilar um arquivo *.class* Java, gerando um arquivo chamado CAP. Esse arquivo é o *applet* que será inserido em um *java card*. A segunda máquina virtual fica armazenada dentro do cartão e é utilizada para executar as aplicações contidas nele. Para isso ela utiliza um interpretador que gerencia a execução, e controle de memória das aplicações. Na figura 12 é possível observar ambas as máquinas virtuais e a atividade de compilação e interpretação do arquivo compilado (CHEN, 2000, p.31).

.

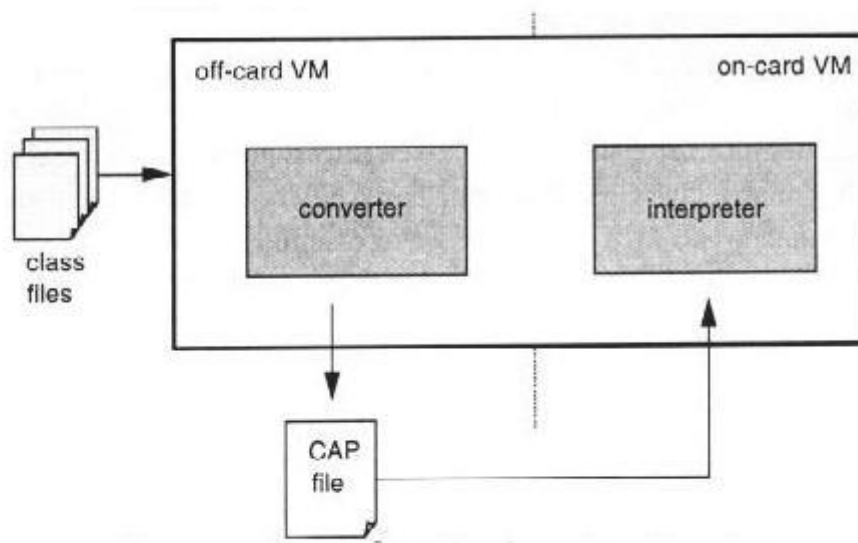

Figura 12 - *Java Card Virtual Machines*. Fonte: CHEN (2000, p.31).

Dentro de um cartão podem coexistir várias aplicações, para isso é necessário garantir a diferenciação de cada uma dentro do *java card*. Cada *applet* possui uma identificação chamada AID. Além disso, cada pacote de aplicações possui seu próprio AID também. Um AID é um *array* de *bytes* que é dividido em 2 partes. A primeira parte possui 5 *bytes* e é chamado de RID. Cada empresa que deseja desenvolver nessa tecnologia deve requisitar o seu RID a ISO. A segunda parte possui tamanho de 0 à 11 *bytes* e é chamado de PIX. Essa parte é utilizada para diferenciar uma aplicação de outra dentro de uma empresa, ou até mesmo a versão da aplicação (CHEN, Z. p.43, 2000).

2.4.1 API *Java Card*

A tecnologia *java card* possui sua própria API que segue as normas ISO 7816. Os pacotes principais contidos nessa API são *java.lang*, *javacard.framework*, *javacard.security*. De maneira resumida, esses pacotes, compreendem operações em *arrays* e variáveis, tratamento de exceções, implementação de algoritmos de segurança e interfaces de comunicação. Essas interfaces são utilizadas para facilitar a criação de aplicações, e gerenciamento das requisições de funcionalidades (CHEN, Z. 2000, p.40).

2.4.1.1 *Java Lang*

O pacote *java.lang* suporta a construção da base da API *java card*, como por exemplo possui as classes *Object* e *Throwable*. Todas as classes da API estendem de *Object* ou

possuem alguma classe “pai” na arquitetura que estende dela. A classe *Throwable* é a base utilizada para qualquer classe de exceção implementada. Além disso, esse pacote possui exceções para tratar erros que geralmente ocorrem apenas em tempo de execução. Como exemplo, podem ser citadas *RuntimeException*, *NullPointerException* e *ArrayStoreException* (CHEN, Z. 2000, p.40).

2.4.1.2 *Java Card Framework*

O pacote *javacard.framework* possui as classes que são utilizadas nas funcionalidades principais das aplicações. Como exemplos pode-se citar a classe *applet* e APDU. A classe *Applet* deve ser estendida por toda aplicação na tecnologia *java card*, ela possui métodos que ajudam na interação com a JCRE, tanto na instalação da aplicação como em tempo de execução da mesma. A classe APDU (do inglês: *Application Protocol Data Unit,* do português: Protocolo de Aplicação em Unidade de Dado) é utilizada para criar uma transparência, para o desenvolvedor, do protocolo de envio das mensagens, T=0 ou T=1. Além disso, é ela que possui todo o dado trafegado de fora para dentro da aplicação, e funções para tratamento desse dado (CHEN, Z. 2000, p.41).

2.4.1.3 *Javacard Security*

O pacote *javacard.secutiry* suporta principalmente a utilização de algoritmos de criptografia e de geração de números randômicos. Esse pacote possui interfaces tanto de algoritmos simétricos como assimétricos. Alguns desses algoritmos são RSA, AES, DES, entre outros. Além disso, utilizando a classe *MessageDigest* é possível criar *Hashs* de dados, algumas implementações são MD-5, SHA-256, SHA-512 (CHEN, Z. 2000, p.41).

2.4.1.4 *Javacardx Crypto*

Além dos pacotes citado anteriormente, a API *java card* possui um pacote de extensão chamado *javacardx.crypto*. Esse pacote possui apenas uma classe chamada *Cipher*, e uma interface chamada *KeyEncryption*. A classe *Cipher*, utilizando algumas classes do pacote *javacard.security*, possui métodos utilizados na codificação e decodificação de um dado. Além disso, ela trata a ocorrência de erros na divergência de modelos da utilização de algoritmos (CHEN, 2000, p.42).

2.4.2 Protocolo de Comunicação

O protocolo de comunicação utilizado na tecnologia *java card* é chamado APDU. De forma simplificada esse protocolo é um *array* de *bytes* onde cada posição do *array* deve ser armazenado um dado com finalidade específica. O tamanho máximo de uma APDU é de 256 *bytes* e caso seja necessário enviar dados com tamanho superior a isso deve ser enviada mais de uma APDU com o restante da informação. Esse protocolo é dividido em: CLA, INS, P1, P2, LC, *DATA FIELD*, LE.

O CLA tem o objetivo de armazenar a identificação da classe que será chamada para executar aquela APDU. O método INS é utilizado para identificar a funcionalidade da classe que deve ser executada. Os parâmetros P1 e P2 são utilizados para diferenciar execuções dentro das funcionalidades. Por exemplo, caso seja necessário, em um mesmo método, executar um tratamento diferente para um determinado parâmetro, o mesmo é enviado dentro de P1 e/ou P2. O campo LC informa qual o tamanho dos *bytes* enviados no *DATA FIELD*. O *DATA FIELD*, como o nome já diz, é utilizado para enviar *bytes* de informação para a aplicação. Seja para armazenamento ou para tratamento dos mesmos. E por fim, o campo LE tem o objetivo de informar qual o tamanho de *bytes* esperado no retorno.

Após a execução da funcionalidade requisitada o *applet* retorna uma APDU de resposta. Essa resposta possui tamanho máximo de 256 *bytes* da mesma forma que é a APDU de envio. A mesma é dividida em: *RESPONSE DATA*, SW1, SW2. O campo *RESPONSE DATA* é utilizado para retornar dados do *applet*. O SW1 e SW2 informam qual foi o resultado da execução da funcionalidade. Caso o valor de ambos somado seja 9000 a execução foi bem sucedida. Caso contrário, é necessário verificar o erro na documentação para saber qual tipo de exceção ocorreu.

O sistema utilizado para comunicação dentro de um *java card* é bastante parecido com o utilizado em uma comunicação cliente/servidor. Nesse caso o computador seria o cliente, e o *applet* o servidor. Dessa forma, é feita a requisição de uma funcionalidade a aplicação contida no cartão, essa funcionalidade é executada e é retornada uma informação como resposta. Essa resposta pode ser algum dado obtido de dentro do cartão, ou confirmação da correta execução da funcionalidade.

2.4.3 Estrutura de um Applet

Existem alguns métodos principais que devem ser de conhecimento de todo programador que deseja criar aplicações na tecnologia *java card*. Os mesmos são:

- *install*(*byte*[] bArray, *short* bOffset, *byte* bLength)
- *process*(APDU apdu)
- *register*(*byte*[] bArray, *short* bOffset, *byte* bLength)
- *select*()
- *deselect*()

2.4.3.1 *Install*(*byte*[] bArray, *short* bOffset, *byte* bLength)

O método *install* cria a instância do *applet* dentro do cartão. Esse método é chamado pela JCRE quando a aplicação está sendo instalada no cartão. Seguindo as boas práticas, é nesse método que são inicializadas as variáveis, dividindo uma parte da memória para a aplicação para evitar falta de memória durante a execução.

2.4.3.2 *Process*(APDU apdu)

Ele é chamado sempre que é feita uma chamada a aplicação depois que ela foi instalada. Dessa forma, sempre que existe um envio de um comando APDU (como pode ser observado na assinatura do método) ele é executado. O mesmo tem o objetivo de receber uma requisição, analisar qual funcionalidade está sendo requisitada e executar a mesma.

2.4.3.3 *Register*(*byte*[] bArray, *short* bOffset, *byte* bLength)

Esse método é utilizado para registrar a instância da aplicação na JCRE. Além disso, ele tem o objetivo de garantir que apenas um *applet* tenha determinado AID instalado em um mesmo cartão. São encontradas duas assinaturas para esse método, uma delas não possui parâmetros e utiliza o AID que foi informado na compilação da classe antes da instalação no cartão. A segunda, que é a informada do título, possui a capacidade de atribuir um AID diferente da utilizada na compilação.

2.4.3.4 *Select*()

Select é chamado sempre que o *applet* é selecionado. Ele pode ser utilizado para inicializar variáveis, alocar espaço na memória para *arrays* temporários, entre outras funcionalidades.

2.4.3.5 *Deselect*()

O método deselect é chamado sempre que ocorre a desseleção do *applet*. Com ele é possível limpar valores temporárias e *arrays* para que o espaço de memória seja utilizado por outra aplicação.

2.4.4 Segurança da Plataforma

A base dos métodos de segurança utilizados pela máquina virtual do *java card* são criados utilizando o que existe na linguagem Java. Eles tem o objetivo de manter o dado consistente e protegido contra ataques. Alguns deles são mencionadas no livro de Chen (CHEN, 2000, p.152) (tradução nossa):

> *"A linguagem Java é fortemente tipada. Não é possível fazer conversões de dados ilegais. Como por exemplo converter dados inteiros para ponteiros. A linguagem Java impõe verificações de limite de acesso à matriz. A linguagem Java não possui ponteiros aritméticos. Sobretudo, não existe maneira de forjar ponteiros para permitir programas maliciosos de procurarem dados dentro da memória. Variáveis precisam ser inicializadas antes de serem utilizadas. O nível de acesso a todas as classes, métodos, e variáveis é rigorosamente controlado. Por exemplo, um método privado não pode ser chamado de fora da classe que o criou."*

Na plataforma *java card* foram criados meios, além das já existentes no Java, para garantir a segurança. Esses tratam a persistência e gerenciamento de dados armazenados, envio e recebimento de mensagens, e acesso a métodos nativos. Os principais são: Objetos transientes e persistentes, atomicidade, *Applet Firewall*, compartilhamento de objetos e métodos nativos.

2.4.4.1 Objetos Transientes

Dentro de uma aplicação *java card* objetos podem ser transientes. Essa característica faz com que o objeto seja armazenado na memória ram, e caso a aplicação seja desselecionada ou resetada o objeto é excluído da memória. Isso é possível declarando o objeto como sendo "*CLEAR_ON_RESET*" ou "*CLEAR_ON_DESELECT*".

2.4.4.2 Atomicidade

Na plataforma *java card* os dados são armazenados em uma memória persistente. Caso ocorra queda de energia o *java card* define 3 aspectos. Primeiramente, ao ocorrer perda

de energia durante uma alteração em um objeto a tecnologia garante que o dado anterior armazenado é restaurado. Segundo, o método *arrayCopy* da classe Util garante a atomicidade na atualização de dados em lote. Por último, a plataforma garante atomicidade, dentro de uma transação, na alteração de vários objetos.

2.4.4.3 *Applet Firewall*

Cada aplicação possui seu espaço de memória, esse espaço é utilizado para armazenar objetos manipulados na execução e armazenamento de dados pelo *applet*. O *firewall* protege que esses objetos sejam acessados por outras aplicações armazenadas dentro do mesmo cartão.

A JCRE possui sua própria partição na memória, essa não pode ser acessada por aplicações contidas no cartão. Além disso, existem métodos nativos, alguns deles implementados na linguagem C, esses só podem ser utilizados pela empresa fabricante do cartão. O *firewall* protege a utilização desses métodos por aplicações que não forem criadas pelo fabricante.

2.4.4.4 Compartilhamento de Objetos

Uma maneira de uma aplicação acessar objetos de outro contexto é se eles implementarem a interface *shareable*. Essa interface torna acessível variáveis e métodos implementados, com isso é possível unificar métodos utilitários em uma classe e disponibilizá-los para vários *applets* instalados no cartão.

Outra maneira de tornar objetos acessíveis por outras aplicações é com a utilização de *arrays* globais. Todos os dados contidos nesses *arrays* são públicos e podem ser acessados por qualquer *applet* a qualquer hora.

2.4.4.5 Métodos Nativos

Métodos nativos são métodos que não são executados pela máquina virtual do *java card*. Eles existem apenas em aplicações criadas pelos fabricantes dos cartões, e essas aplicações tem a característica de serem armazenadas na memória ROM. Portanto, esses métodos não estão sujeitos as regras de segurança implementadas na plataforma. Eles podem ser implementados em C, fazer alteração em qualquer parte da memória, e possuem o controle total das aplicações armazenadas no cartão.

3 MATERIAS E METODOS

Levando em consideração as informações que podem ser armazenadas em um *java card*, como dados pessoais, dados financeiros, chave de acesso a algum serviço, entre outros, é necessário garantir a segurança durante a comunicação com essa tecnologia. Além disso, qualquer dado utilizado para autenticar o portador do cartão deve ser mantido em segredo para que não seja possível replicar o mesmo e acessar qualquer serviço em execução no cartão. Para isso a transação entre o computador e o cartão deve ser criptografada, tornando difícil a descoberta das mensagens trocadas entre ambas as partes.

O objetivo do projeto é, utilizando criptografia homomórfica, transportar um PIN codificado para o cartão, sendo que o cartão deve decodificar o PIN e autenticar o usuário. Durante o projeto é necessário executar a análise dos resultados obtidos por testes. Esses testes compreendem a análise do tempo de execução e a capacidade de utilização em um cenário real. Para a resolução do projeto foi necessário seguir os seguintes passos:

- Estudo da tecnologia *Java Card*: pesquisa da tecnologia para adquirir conhecimento na API, estudo das melhores práticas para economizar memória durante a execução, e trabalhos correlatos para entender como é criada uma aplicação a nível de produção.
- Estudo da Criptografia Homomórfica: pesquisa da criptografia para conhecer os diferentes esquemas dessa criptografia, escolha do melhor para o projeto, busca por implementação do esquema. Procura de trabalhos correlatos para entendimento das aplicações que utilizam criptografia homomórfica.
- Desenvolvimento de aplicação desktop na linguagem Java: essa aplicação tem o objetivo de fazer a comunicação com uma implementação da criptografia homomórfica, e enviar a chave privada, dados codificados, e cifra expandida para o cartão.
- Desenvolvimento de aplicação *Java Card* (*Applet*): essa aplicação deve possuir a funcionalidade de decodificação de um dado cifrado homomorficamente. É necessário utilizar as informações adquiridas no estudo da tecnologia para armazenar parte da chave, parte da cifra, e um fragmento da cifra estendida de forma eficiente.
- Desenvolvimento de aplicação Python: levando em consideração que o código implementado em python por Santos (2014) será utilizado, é necessário alterar o código para receber os dados enviados pela aplicação java e retornar o resultado para o mesmo.

- Testes na aplicação: é necessário executar testes para avaliar o desempenho das aplicações envolvidas. Esses testes compreendem tanto a geração das chaves, e codificação dos dados pelo gerenciador, como o envio do dado cifrado para o *applet* e decodificação do mesmo.
- Análise dos dados obtidos durante os testes para criação da conclusão do projeto: levantamento dos dados obtidos e análise dos mesmos. Com os dados é possível avaliar a utilização da decodificação em um ambiente real utilizando a tecnologia *java card*.

3.1 Materias Utilizados

Para a execução do projeto foram utilizados os seguintes materiais:

- Samsung Ativ Book 6 com 8 GB de Memória Ram de 1600 MHz, 1 TB de HD com 7200 RPM, e com processador i5 3230M. As características do processador são: frequência base de 2.6 GHz, frequência máxima de 3.2 GHz, 3 MB de memória cache, número de núcleos igual a 2, número de threads igual a 4.
- O sistema operacional utilizado foi o Windows 8.1.
- Eclipse IDE.
- JDK e JCDK.
- Simulador JCardsim.
- Biometric SDK
- Banco de dados Postgresql
- Implementação do esquema de Coron (et al 2011) de chave reduzida implementado por Santos (2014).

4 TRABALHOS CORRELATOS E IMPLEMENTAÇÕES

Para entender a área em que se encontra o presente trabalho é necessário analisar projetos contidos na mesma área. Levando em consideração, a utilização a tecnologia *Java Card* e Criptografia Homomórfica, é necessário analisar projetos de ambas as áreas. Essa análise tem o objetivo de conhecer o que deve ser estudado de forma aprofundada, o que é utilizado nos dias de hoje, e os possíveis obstáculos durante a execução do projeto.

Na área de criptografia homomórfica os exemplos mais comuns são de armazenamento de dados codificados homomorficamente, e a implementação de um esquema homomórfico para garantir que o voto, em alguma eleição, seja secreto. Uma das formas de garantir o voto secreto é alterar a comunicação entre o terminal do voto e o servidor de votos. Essa tarefa deve ser feita como explicado a seguir: o servidor de votação gera o par de chaves homomórficas, então a chave pública é enviada ao terminal. Ao efetuar uma votação o terminal cifra o voto e o envia ao servidor. O servidor tem a capacidade de executar operações sobre o dado cifrado, levando em conta que é um esquema homomórfico, então ele apenas decifra o dado, utilizando a chave privada, quando a votação for finalizada.

4.1 Método Utilizando Algoritmo de Criptografia Homomórfica One-way no Cenário de Comunicação em Redes

No trabalho de Peng e Bao (PENG e BAO, 2010) é feita a análise sobre um método de votação secreta, utilizando criptografia homomórfica, e propõe um novo método utilizando criptografia homomórfica *one-way* no cenário de comunicação em redes. Esse método utiliza um servidor seguro, servidor de votação, terminal de votação, um computador para o avaliador, e a rede em si. Cada eleitor recebe um inteiro secreto aleatório, esse inteiro é utilizado na operação de codificação do voto. Além do eleitor, o servidor de votação também recebe esse inteiro. O primeiro eleitor executa W1*f(V1) mod P, onde W1 é o inteiro aleatório secreto. f(V1) é a função de codificação executada no valor escolhido durante a votação (0 para não e 1 para sim), e P é um inteiro positivo grande.

Então o primeiro eleitor envia o resultado dessa operação para o segundo eleitor e ao servidor de votação, e assim segue até o final da votação. Após o térmico da votação, o ultimo eleitor envia o resultado das múltiplas operações ao servidor de votação. Esse servidor executa operações matemáticas para retirar do resultado os inteiros aleatórios utilizados nas operações de codificação obtendo f(V1+V2+…+Vn). Para verificar o resultado o servidor

gera uma tabela com todas as combinações de voto possíveis, sendo T1={C1 (=f(V1+V2+…+Vn)) , (V1+V2+…+Vn) }.

Dessa maneira, é possível validar qual a quantidade de votos, sim e não, levando em consideração a quantidade de eleitores. Esse esquema é possível dentro de uma comunicação de redes sendo que a quantidade de eleitores seria pequena, não necessitando de uma tabela muito grande. Esse trabalho mostra uma utilização da criptografia homomórfica dentro de um ambiente real, dessa forma é possível garantir a segurança, não sendo possível falsificar um voto, e o anonimato do eleitor.

4.2 Modelo de Transação Seguro e de Custo Eficaz para Serviços Financeiros

Ao estudar a tecnologia *java card*, é possível encontrar trabalhos, que utilizam a mesma, para criação de aplicações para autenticar usuários, acesso móvel a funcionalidade, e execução de transações financeiras. Nessas transações financeiras o cartão pode armazenar os dados necessário para autenticar o usuário do cartão, e/ou armazenar o valor monetário inserido pelo proprietário.

Um exemplo de aplicação nesse meio consta no trabalho de Munjal e Moona (2009). Onde foi proposto um modelo de aplicação para s*mart card* que possui o objetivo de executar transações financeiras. Esse modelo precisa garantir autenticação, integridade, confidencialidade, e que não seja possível replicar algum dado. Para isso é utilizado um certificado digital que é assinado pelo usuário e pelo banco, esse certificado fica armazenado no cartão e também na base de dados do banco.

Quando o usuário deseja executar alguma operação primeiramente ocorre a troca de certificados entre o cartão e o serviço bancário, e ambos os certificados são avaliados pelas partes para verificar a autenticidade. Após isso, ocorre a verificação da autenticidade de ambos os lados, utilizando as chaves públicas, estabelecendo chave de sessão. Nessa autenticação é feita a troca de um código de autenticação cifrado pelas chaves públicas. E então, é feito o envio ao serviço bancário dos dados bancários para que seja possível executar transações financeiras.

Esse projeto apresenta como exemplo de aplicação três cenários para execução de transações financeiras. Um deles possui a utilização de um celular como meio de conversação entre o serviço bancário e o *smart card*. A comunicação entre o celular e o cartão seria

possível utilizando NFC (*Near Field Communication*). E a comunicação entre o celular e o sistema bancário seria feita utilizando a internet do aparelho.

Nesse exemplo é possível observar a necessidade de garantir a segurança durante e após a transação financeira. As implementações encontradas que utilizam essa tecnologia estão fortemente ligadas à área de segurança, utilizando várias chaves e métodos para garantir a proteção dos dados armazenados e trafegados.

5 DESENVOLVIMENTO

A arquitetura do projeto pode ser dividida em três partes: aplicação *java card*, aplicação java SE, e aplicação python. O *applet* com a capacidade de armazenar a identificação do usuário, a biometria, o PIN e a chave privada homomórfica. A identificação do usuário é utilizada para verificar a identidade do usuário, para diferenciar qual é o PIN codificado e a cifra expandida que deve ser enviada ao cartão. A chave privada é utilizada na operação de decodificação do PIN e, após a decodificação do PIN, é feita a autenticação do usuário. A biometria é utilizada em conjunto com o PIN diminuindo as chances de réplica da autenticação.

Qualquer transmissão do cartão para o gerenciador e vice-versa ocorre dentro de uma sessão segura. A sessão é criada seguindo os passos: primeiramente é gerado um par de chaves RSA dentro do cartão. O módulo e o expoente da chave são enviados ao gerenciador que obtém a chave pública a partir dessas informações. Então o gerenciador gera uma chave AES de 16 bytes aleatórios, codifica com a chave pública, e envia ao cartão.

Dessa maneira ambos possuem a chave AES síncrona. Para validar a identificação de ambos, é utilizada uma autenticação *challenge-response*, onde o cartão gera 16 bytes aleatórios, codifica com a chave AES e os envia ao gerenciador. O gerenciador decodifica os bytes aleatórios, gera mais 16 bytes aleatórios, e cria um *hash* da concatenação de ambos. Então é enviado codificado para o cartão o *hash* e os bytes gerados pelo gerenciador. Onde o cartão gera o seu próprio *hash* da mesma maneira, e se ambos estiverem idênticos é validada a sessão segura.

O gerenciador deve ler os dados do usuário como PIN, biometria, identificação, requisitar os serviços do código python, enviar os mesmos para o cartão. Mas, como o cartão não tem memória suficiente para armazenar toda a cifra, e a cifra expandida, é necessário tratar essas informações e dividi-las em blocos. O código python tem as funcionalidades de gerar as chaves criptográficas, codificar o PIN, e expandir a cifra do mesmo.

Para um entendimento melhor sobre cada parte da arquitetura e suas principais funções foi criado um esquema. O esquema apresenta três círculos com os nomes de Módulo de Criptografia Homomórfica, Gerenciador, e *Applet*. Ligando os mesmo existem setas informando as atividades básicas executadas, onde as mesmas são divididas em 4 passos. Primeiramente é feita a requisição da geração de chaves, codificação. No segundo passo é retornada a chave privada e dado codificado. No terceiro é enviada a chave ao *applet*. Por fim, é enviada a cifra ao cartão. Esse esquema pode ser observado na figura 13.

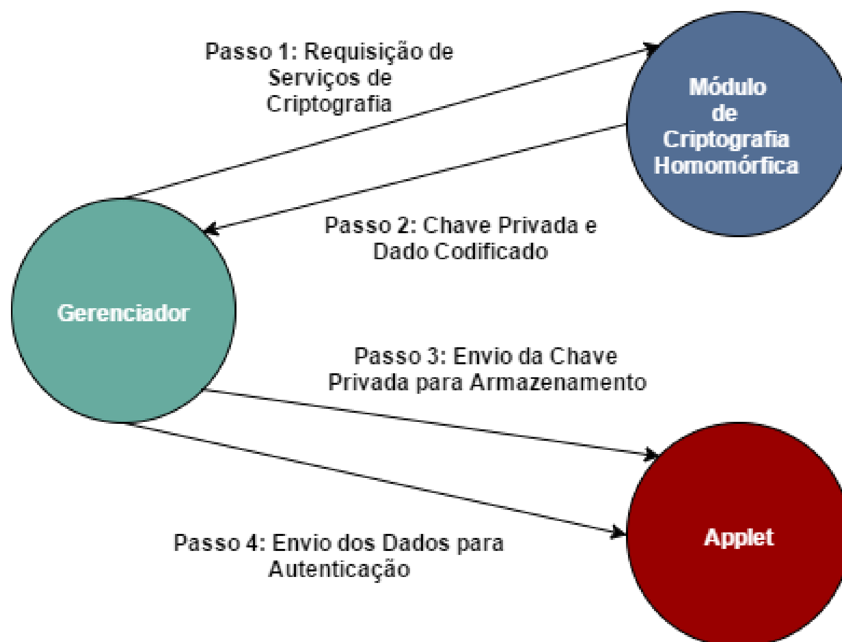

Figura 13 - Diagrama da Arquitetura

5.1 Implementação

Para facilitar a utilização do código python foi criado um *webservice* que pode ser acessado através de um host local ou externamente. O *webservice* tem a finalidade de receber requisições do gerenciador, executar as funções e retornar a resposta. Além disso, o código python de Santos (2014) salva em arquivos os resultados das funções implementadas para análise dos resultados. Para utilização no projeto o mesmo foi "refatorado", retornando os valores das funções e inseridos dentro do *webservice*.

Esse serviço possui algumas funções, onde as mesmas são: a) geração de chaves, codificação e expansão do PIN. b) retorno para o gerenciador da cifra do PIN e da cifra expandida. Durante a primeira função o *webservice* recebe os dados utilizados na identificação do usuário, e o pin para criação de um novo usuário. O par de chaves criptográficas é gerado, e o PIN é codificado com a chave pública. Então, cada cifra é expandida.

A identificação do usuário, a chave pública, o PIN codificado, e a cifra expandida são armazenados em um banco de dados postgresql para posterior uso. E então, a chave privada é retornada para o gerenciador. Para execução da segunda função o gerenciador envia a identificação do usuário, e o *webservice* retorna o PIN codificado e a cifra expandida.

Uma observação importante que deve ser citada é a de que o esquema totalmente homomórfico de Coron (et al, 2011) faz a codificação bit a bit. Sendo assim, pela utilização

de um PIN de 8 bytes é necessário executar 64 codificações, e cada cifra é expandida. A cada execução a cifra resultante é concatenada em uma *string* e separada por um conjunto de caracteres para facilitar a posterior divisão das mesmas.

Para o projeto foi escolhido utilizar o algoritmo no modo *Small*, o mesmo possui uma cifra expandida, que é uma matriz, de 23x23. Essa matriz é transformada em *string* e é concatenada, junto as outras expansões, em outra *string*. Sendo assim, o *webservice* irá armazenar todas as cifras, e todas as cifras expandidas em apenas duas *strings* diferentes para cada usuário.

O gerenciador, após enviar os dados de identificação e o PIN ao *websevice* aguarda o retorno da chave privada homomórfica para inserção no cartão. O mesmo deve inicializar o *applet* com a identificação, o PIN, e a biometria do usuário. Após a execução do ciclo completo de inicialização é possível iniciar a autenticação do usuário. Para isso, o gerenciador requisita o PIN codificado e a cifra expandida ao *webservice*, os mesmos são divididos em listas para enviar a cifra e a cifra expandida separadamente ao cartão.

Primeiramente a cifra expandida é dividida de acordo com as linhas da mesma, criando vários vetores. Levando em consideração, que é necessário enviar 64 cifras expandidas ao cartão, e que cada cifra possui 23 linhas, essa operação é executada 1472 vezes para completa decodificação.

Mas, a tecnologia *java card* não possui uma primitiva para tratamento de ponto flutuante o mesmo deve ser tratado dentro da aplicação. São levadas em consideração até quatro casas decimais após a vírgula, separando cada dado em três colunas na matriz. O primeiro armazena os valores inteiros, o segundo é a primeira e segunda casa depois da vírgula, e o terceiro é a terceira e a quarta casa. Cada valor é convertido para bytes, codificado pela chave AES utilizada na criação da sessão segura e enviado ao cartão linha por linha.

Durante a decodificação o cartão recebe cada linha da matriz, decodifica a mesma, e executa a soma dos dados fazendo o tratamento de ponto flutuante. Durante a soma é feita a multiplicação dos vetores da chave pelo valor armazenado na linha da matriz enviada, sendo s0 e s1 os vetores que compõe a chave privada, e z a linha da matriz:

$$s0[x] * s1[y] * z[x][y]$$

Visto que os vetores s0 e s1 apenas são constituídos por bytes 0 e 1, apenas quando o valor no vetor s0 no index “x” e o valor no vetor s1 no index “y” forem ambos o número 1 que será feita a soma. Pois, qualquer valor multiplicado por zero é zero. Dessa maneira, é possível ignorar a execução da soma para qualquer linha onde o valor de s0[x] for 0, e

qualquer coluna onde o valor de s1[y] for 0. Um exemplo desses vetores é possível observar a seguir:

$$S0 = [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$$

$$S1 = [1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$$

Analisando o exemplo anterior, apenas nas linhas com index de "x" igual a 0, 2, 13 e 20 o valor poderá ser alterado. Sendo assim, em uma matriz de 23x23 apenas 4 linhas serão utilizadas durante a decodificação. Além disso, apenas nas colunas com index de "y" igual a 0, 6, 9, e 19, onde os valores de s1 para o index "x" são 1, será executada a soma.

Caso o tempo de execução seja medido, é possível determinar os valores de s0. Sendo que, se o valor for 1 a velocidade em que o dado será retornado ao gerenciador será menor do que quando o valor for 0. Mas, ainda sem os valores reais do vetor s1 não é possível determinar qual é a chave privada, garantindo a segurança do sistema.

Após a soma de todas as linhas da matriz, o valor resultante é arredondado para o inteiro mais próximo. No algoritmo de Santos (2014), a operação executada depois do arredondamento é a subtração da cifra pela soma, e depois é executado o *mod* no resultado. Essa operação pode ser observada a seguir em python, onde cask é a cifra:

```
(cask - soma) % 2
```

O resultado dessa operação é o valor real da cifra, sendo 0 ou 1. Para não ser necessário enviar toda a cifra para o cartão, sendo que é um número inteiro que o cartão não tem capacidade de armazenar, são contadas quantas casas decimais existem na soma. Então é enviada a casa decimal acima da cifra que for diferente de 0. Sendo a cifra 2341295 e a soma 36, apenas o valor 295 será enviado ao cartão.

Esse processo é repetido para os 64 bits, e a cada execução é feito o deslocamento de bits para a esquerda em um byte. A cada 8 bits o byte utilizado é inserido em um vetor, e os próximos deslocamentos ocorrem em um novo byte. Dessa forma, no final da execução, todos os valores compreenderam 8 bytes. Esse resultado é utilizado na autenticação para liberar acesso as funcionalidades implementadas no *applet*.

Após a autenticação por PIN é requisitada a biometria do usuário ao cartão, o mesmo retorna ao gerenciador que autentica o mesmo caso a porcentagem de correspondência seja maior que 80%. Esse valor foi escolhido de acordo com os testes executados. Para o cálculo da porcentagem foi utilizada uma biblioteca chamada Biometric SDK, essa biblioteca é *Open Source*, desenvolvida em java e encontra-se disponível no site github.

5.2 Testes

Para analisar a eficiência da aplicação foi necessário executar testes unitários e obter o tempo de execução de cada função executada. Nos testes foi utilizado um notebook Samsung Ativ Book 6, com 1 TB de HD e 7200 RPM, com 8 GB de Memória Ram de 1600 GHZ, e processador i5 3230M. Para facilitar os testes unitários um simulador de *java card* foi utilizado, chamado JCardsim. O simulador permite criar *breakpoints* onde é possível parar a execução e verificar o valor de cada variável existente no *applet*. Alguns tipos diferentes de teste foram escolhidos, e os mesmos executados 100 vezes cada.

A aplicação desenvolvida no projeto primeiramente deve ser inicializada, onde um par de chaves RSA é gerado, e as informações do usuário são inseridas no cartão. Além disso, é executada a criação de sessão segura para cada função requisitada do cartão. Então, é possível autenticar o usuário, utilizando o PIN codificado homomórficamente, e pela biometria do usuário. Além da média dos resultados obtidos durante os testes, foi calculado o quartil 3 que leva em consideração 75% dos valores, e ignora dados com valores extremos.

5.2.1 Sem Homomorfismo

Como é necessário comparar o tempo de execução utilizando a criptografia homomórfica e sem essa criptografia, primeiramente foram executados testes unitários em uma aplicação capaz de autenticar por PIN sem o uso da mesma. Portanto, primeiramente o *applet* é inicializado, o par de chaves RSA é gerado, os dados do usuário como a identificação, pin e biometria são inseridos no cartão. Após esse passo, é feita a criação da sessão segura, onde a chave AES é enviada ao cartão, e ocorre a autenticação de ambos os lados por *challenge-response*. Então o PIN é enviado ao cartão, autenticado, e caso o resultado retorne positivo é possível requisitar a biometria do usuário para que seja autenticada no gerenciador da aplicação. O resultado desse teste, em milissegundos, pode ser observado na figura 14.

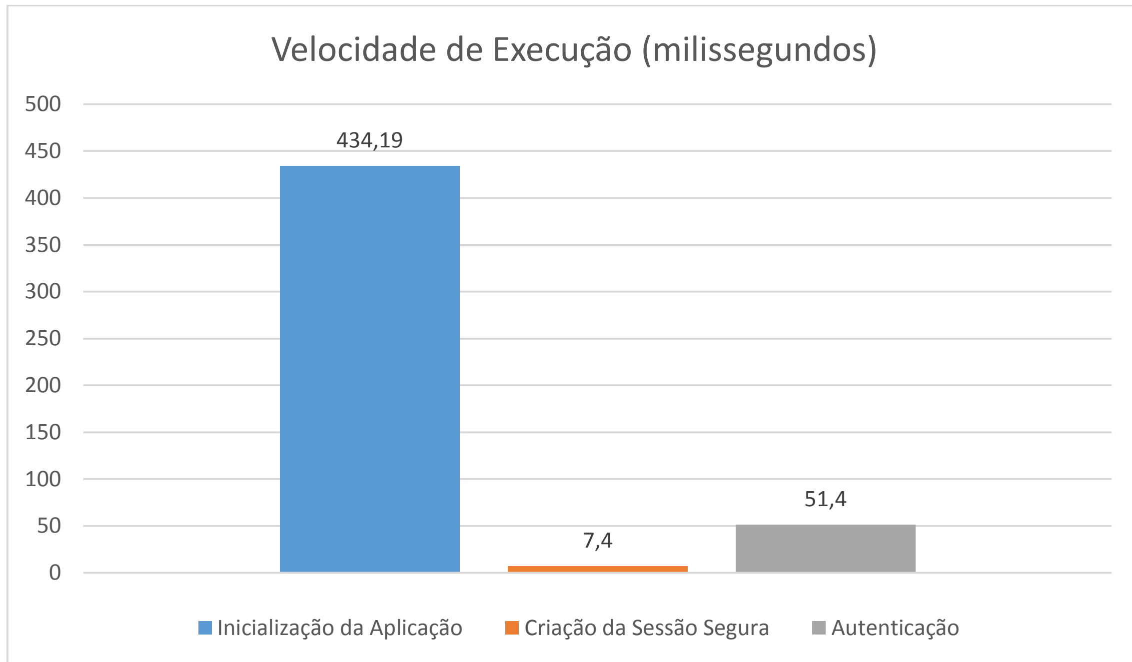

Figura 14 - Gráfico sem homomorfismo

O tempo elevado, comparado com as outras funções, encontrado na inicialização da aplicação é referente a inserção de dados na aplicação, sendo PIN, biometria e identificação do usuário. Para inserir a biometria no cartão é necessário a leitura de um *template* biométrico, dividi-lo em partes, e codificar todas essas partes para envio. O quartil encontrado na execução dos testes, para essa funcionalidade, foi de 545,5. Já na autenticação pelo PIN e biometria o valor encontrado foi de 49. E, por fim, durante a criação da sessão segura o valor encontrado para o quartil foi de apenas 7. Analisando a diferença entre a média e os dados de quartil encontrados, é possível dizer que a mesma ocorre pelos programas executando de forma concorrente no computador, sendo que, quanto menor o tempo de execução menos a função será afetada pela concorrência.

5.2.2 Com Homomorfismo

No teste utilizando criptografia homomórfica é necessário levar em consideração o tempo de geração das chaves, codificação do PIN e expansão da cifra. Portanto, primeiramente é requisitado ao *webservice* a geração das chaves, codificação do PIN, e expansão da cifra. Depois dessa operação a chave privada é retornada ao gerenciador que armazena a mesma no *applet*. Assim, o *applet* é inicializado seguindo as mesmas operações executadas sem homomorfismo, diferente apenas pelo armazenamento dos vetores que correspondem a chave privada gerada.

Depois, a sessão segura é criada, e o PIN codificado e a cifra expandida são requisitados pelo gerenciador. Esses dados são tratados pelo gerenciador antes de ser enviado ao cartão. O *applet* decodifica o PIN, autentica o usuário, e depois de autenticar pela biometria é liberada a execução das outras funcionalidades. É possível observar na figura 15 o tempo de execução dessas operações, sendo que as operações de geração de chaves, codificação e expansão das cifras não aparecem na figura, pois o tempo de execução da mesma foi muito superior do das outras. Para a mesma, o tempo foi de 355360,3 milissegundos, aumentando o tempo de execução em cerca de 6 minutos.

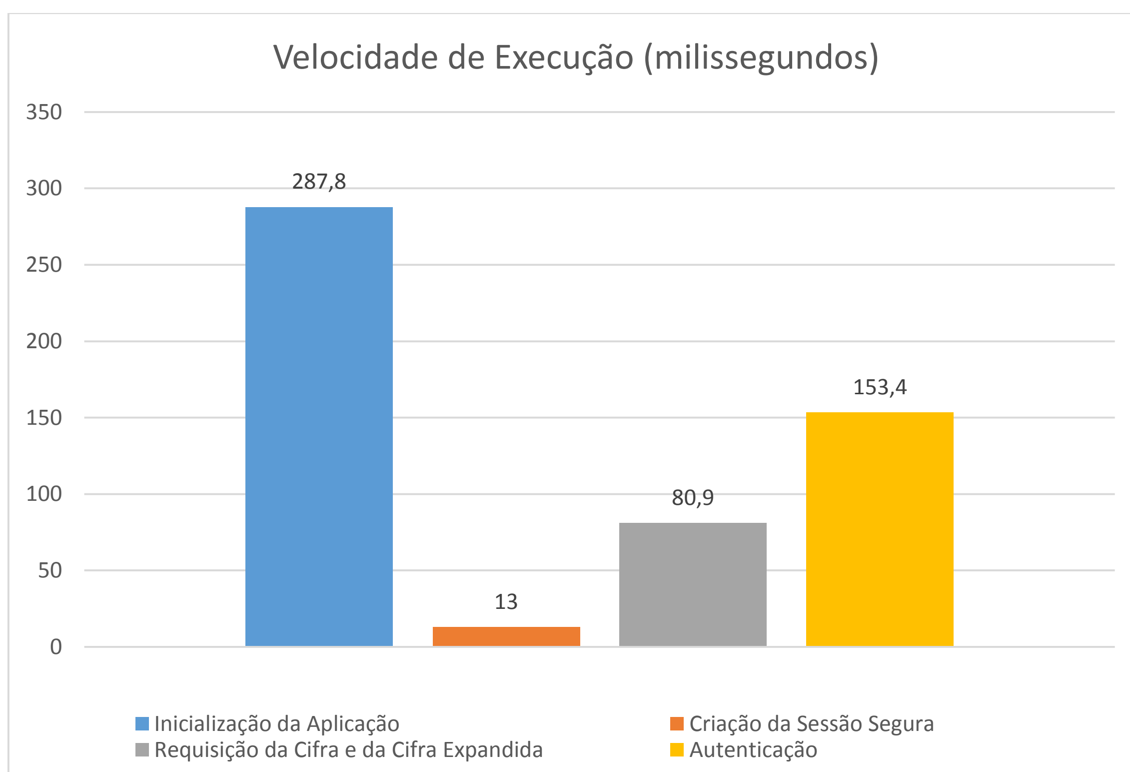

Figura 15 - Gráfico com homomorfismo

Observando o tempo médio de execução da inicialização da aplicação sem homomorfismo e com homomorfismo, é possível verificar uma diferença grande entre os resultados. Para isso, foi necessário ignorar valores extremos através do cálculo do quartil, tendo obtido valor de 401,75 para a operação com homomorfismo.

Comparando a autenticação sem a utilização de criptografia homomórfica com a que faz o uso da mesma, é possível verificar que o tempo médio de execução triplicou. Sendo que, o quartil encontrado para essa função foi de 140,75. Essa diferença é referente a necessidade de envios contínuos de informação para ao *applet* e tratamento da mesma.

Pelo resultado de tempo apresentado nas funções de geração de chaves, codificação e expansão das cifras, foram executados testes com esses dados pré gerados. As chaves foram

geradas, o PIN foi codificado, e as cifras expandidas, e todas essas informações foram armazenadas em um banco de dados. Sendo que, ao receber a requisição para geração das chaves o *webservice* verifica, pela identificação, se o usuário existe no banco de dados, se sim ele apenas retorna a chave privada. Isso reduziu o tempo de execução médio de 355360,3 para 68,41 milissegundos.

5.2.3 Tempo de decodificação no *Smart Card*

Levando em consideração, que o objetivo do projeto é executar a decodificação homomórfica para autenticação em *java card*, é possível restringir os resultados de teste apenas para as interações entre o gerenciador e o cartão. Esse cenário apenas compreende o envio dos dados formatados para o cartão, e a autenticação por PIN executada no cartão. A autenticação por biometria não foi levada em consideração nesse testes por nela não utilizar criptografia homomórfica, e sim apenas a chave AES da sessão segura. Para os testes unitários foi feito o uso de um simulador é possível converter o *clock* do computador, sem retirar os programas concorrentes, sem adicionar o tempo de envio da informação do notebook ao cartão, e calcular de acordo com o *clock* de um cartão. Além disso, não foi levado em consideração o número de ciclos de cada processador.

O tempo médio encontrado para a execução das operações do cartão, durante a autenticação, foi de 8,72 milissegundos. Com valor para quartil encontrado de 9. Como o *clock* do notebook varia entre 2,6 Ghz à 3,2 Ghz, foi feito o uso nesse cálculo do maior *clock*. O *clock* encontrado em *java cards* varia de 3,5712 Mhz à 4,9152 Mhz, da mesma forma que será utilizada a maior frequência do processador do notebook, no cartão o valor será de 4,9152 (CHEN, 2000).

Primeiramente é necessário converter de Ghz para Mhz a frequência do processador, para isso é feita a multiplicação do *clock* do computador por 1000. Então, é possível multiplicar o valor resultante pelo tempo e então dividi-lo pelo *clock* do cartão. No cálculo apresentado a seguir o “cclock” é o *clock* máximo do cartão, “nclock” é o *clock* máximo do notebook, o tempo de execução foi denotado por “t”, e o tempo resultante por “tres”.

$$tres = \frac{nclock * 1000 * t}{cclock}$$

$$tres = \frac{3,2 * 1000 * 8,72}{4,9152}$$

$$tres = 5677,0833$$

Analisando o resultado é possível observar que a execução completa da decodificação e autenticação por PIN demora em média 5,677 segundos para ser executada. Esse resultado apresenta um tempo relativamente baixo se for comparado com as outras operações executadas, e o custo da utilização de criptografia homomórfica.

6 CONCLUSÃO

De acordo com os resultados obtidos durante o desenvolvimento e os testes na aplicação, é possível observar a capacidade da utilização de um algoritmo de criptografia homomórfica em um *java card*. Essa afirmação é fundada pelo tempo de execução médio encontrado de 5,677 segundos durante a decodificação de um dado no *applet*. Mesmo tendo que executar 64 vezes a decodificação o tempo não ultrapassou um minuto, tornando possível a execução da mesma sem necessidade de um tempo elevado.

Além disso, a autenticação utilizando um dado codificado homomórficamente torna segura a transmissão do PIN codificado ao *applet*, escondendo o PIN do usuário. Sendo assim, com a divisão das operações e o envio fragmentado dos dados necessários durante a decodificação, é possível executar a decodificação homomórfica dentro de um *smart card*. O mais importante é pela utilização de criptografia homomórfica não é possível obter a chave pública, derivá-la utilizando o algoritmo de Shor (SHOR, 1994) e decodificar o PIN para descobrir o valor real.

É necessário ressaltar que não são levados em consideração os programas executados de forma concorrente no computador, e também o custo de envio das informações ao cartão (custo de transmissão). Sendo que, os valores encontrados nos testes serão alterados em um cenário real.

Em relação a trabalhos futuros, espera-se executar os testes em um *smart card*, para comparar o tempo de execução com os resultados obtidos nos testes no simulador. Além disso, é possível ampliar a aplicação do *webservice*, com a criação de novas funções que utilizam a criptografia homomórfica.

REFERÊNCIAS

AGUILAR-MELCHOR, Carlos et al, **Recent Advances in Homomorphic Encryption: A Possible Future for Signal Processing in the Encrypted Domain**, IEEE Signal Processing Magazine, fev. 2013.

ANDRESS, J., "**The Basics of Information Security**". Syngress, 2014.

BERK, S. et al, **Accelerating Fully Homomorphic Encryption Using GPU**, IEEE Conference on High Performance Extreme Computing (HPEC), 2012.

BILAR, G. R., **Implementação do Esquema Totalmente Homomórfico sobre Números Inteiros Utilizando Python com Compressão de Chave Pública**, 2014.

BHAKTA, Ram, HARRIS, Ian G., **Semantic Analysis of Dialogs to Detect Social Engineering Attacks**, IEEE International Conference on Semantic Computing (ICSC), feb. 2015.

BLAIR, A., **Learning the Caesar and Vigenere Cipher by hierarchical evolutionary re-combination**, IEEE Congress on Evolutionary Computation (CEC), jun. 2013.

Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil, Cartilha de Segurança para Internet. Disponível em: http://cartilha.cert.br/. Acessado em: 17/03/2015.

CHEN, Z., **Java Card Technology for Smartcards: Architecture and Programmer's Guide**, Addison-Wesley Professional, set. 2000.

CORON, J. S., MANDAL, A., NACCACHE, D. e TIBOUCHI, M., **Fully Homomorphic Encryption over the Integers with Shorter Public Keys**. In P. Rogaway (Ed.), CRYPTO 2011, LNCS, vol. 6841, Springer, pp. 487-504. Full version available at IACR eprint, 2011.

DIJK, M. V., GENTRY, C., HALEVI, S., VAIKUNTANATHAN, V., **Fully Homomorphic Encryption over the Integers** Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. 2010.

EMC, RSA Laboratories, **Public-key Cryptography Standards (PKCS)**. Disponível em: http://brazil.emc.com/emc-plus/rsa-labs/standards-initiatives/public-key-cryptography-standards.htm. Acessado em: 12/03/2015.

FAROOQ, A., ULLAH, Kakakhel S.R., **Information Security Awareness: Comparing Perceptions and Training Preferences**, 2nd National Conference on Information Assurance (NCIA), dez. 2013.

GENTRY, C., **A Fully Homomorphic Encryption Scheme**. Ph.D. thesis, Stanford University, 2009. Disponível em: http://crypto.stanford.edu/craig. Acessado em: 18/03/2015.

GENTRY, C., e HALEVI, S., **Implementing Gentry's Fully-homomorphic Encryption Scheme**, Advances in Cryptology-EUROCRYPT 2011, pp. 129-148, 2011.

GOLDWASSER, S., MICALI, S., **Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information**, STOC '82 Proceedings of the fourteenth annual ACM symposium on Theory of computing, 1982.

GUPTA, C. P., SHARMA, I., **A Fully Homomorphic Encryption scheme with Symmetric Keys with Application to Private Data Processing in Clouds**, Fourth International Conference on theNetwork of the Future (NOF), 2013.

HAMDI, S.M., ZUHORI, S.T., MAHMUD, F., PAL, B., **A Compare Between Shor's Quantum Factoring Algorithm and General Number Field Sieve**, International Conference on Electrical Engineering and Information & Communication Technology (ICEEICT), abr. 2014.

HAN, J. et al, **The Implemention and Application of Fully Homomorphic Encryption Scheme**, Second International Conference on Instrumentation & Measurement, Computer, Communication and Control, 2012.

MIR, M.S., WANI, S., IBRAHIM, J., **Critical Information Security Challenges: An Appraisal**, 5º International Conference on Information and Communication Technology for the Muslim World (ICT4M), mar.2013.

MUNJAL, N., MOONA, R., **Secure and Cost Effective Transaction Model for Financial Services**, Department of Computer Science and Engineering Indian Institute of Technology, out 2009.

PAILLIER, P. **Public-key Cryptosystems Based on Composite Degree Residuosity Classes**, Advances in cryptology - EUROCRYPT’99. Springer Berlin Heidelberg, 1999.

PAULI, Josh, **Introdução ao Hacking e aos Testes de Invasão**, Facilitando o Hacking Ético e os Testes de Invasão, Novatec, 2013.

PENG, K., BAO F., **Efficient Proof of Validity of Votes in Homomorphic E-Voting**, 4th International Conference on Network and System Security (NSS), set. 2010.

RIVEST, R., ADLEMAN, L., DERTOUZOS, M., **On Data Banks and Privacy Homomorphisms**, 1978.

RUIU, D., **Learning from Information Security History**, IEEE Security & Privacy, xfev.2006.

SAN-UM, W., SRICHAVENGSUP, W., **A Topologically Simple Keyed Hash Function Using a Single Robust Absolute-value Chaotic Map**, IEEE International Conference on Communication, Networks and Satellite (COMNETSAT), 2013.

SANTOS, L. C., **Implementação do Esquema Totalmente Homomórfico Sobre Inteitos com Chave Reduzida**, 2014.

SHOR, P. W. **Algorithms for quantum computation: discrete logarithms and factoring**. In Shor, P. W., editor, 35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994), páginas 124–134. IEEE Comput. Soc. Press, 1994.

SMART CARD ALLIANCE, **Smart Card Market Information**, Disponível em: http://www.smartcardalliance.org/smart-cards-intro-market-information/. Acessado em: 13/03/2015

US Government, Legal Information Institute, Título 44, Capítulo 35, Subseção 3502, Cornell University Law School.Disponível em: www.law.cornell.edu/uscode/44/3542.html. Acessado em: 09/03/2015.

APÊNDICE A – CÓDIGOS DA DECODIFICAÇÃO – APPLET

1 VerifyPinSk

Método que faz o tratamento, com uma máquina de estados, das funções executadas na decodificação. Onde essa operação é dividida em "executePlusSkExpand", "sendDecimalPlaces", "executeMenusSum", e Autenticação. No mesmo é feita a verificação da correta execução dos métodos, e caso ocorra alguma exceção o processo inteiro é desfeito. Sendo assim, apenas é executada a autenticação se todos os 64 bits forem decodificados e concatenados no array de bytes do PIN.

```
private boolean verifyPinSk(APDU apdu, byte[] buffer, short recLen) {

        try {
            cipherAES.init(aesKey, Cipher.MODE_DECRYPT, new byte[16], (short) 0, (short) 16);
            cipherAES.doFinal(buffer, ISO7816.OFFSET_CDATA, recLen, buffer, ISO7816.OFFSET_CDATA);
        } catch (Exception e) {
            ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }

        switch (stateMachine) {
        case 1:
            if (executePlusSkExpand(buffer))
                stateMachine = 2;
            break;
        case 2:
            if (sendDecimalPlaces(apdu, buffer))
                stateMachine = 3;
            break;
        case 3:
            if (executeMenusSum(buffer))
                if (pinByteCount < 64) {
                    somaSk = 0;
                    somaSkFra = 0;
                    indexSk0Verify = 0;
                    indexMatrixSk0 = 0;
                    stateMachine = 1;
                } else
                    stateMachine = 4;
            break;
        case 4:
            if (ownerPIN.check(new byte[] { 49, 50, 51, 52, 53, 54, 55, 56 }, (short) 0, (byte) 8) == false)
                ISOException.throwIt(SW_VERIFICATION_FAILED);
            stateMachine = 5;
            break;
        }

        if (stateMachine == 5)
            return true;
    }
```

Figura 16 - VerifyPinSk. Fonte: própria.

3 SendDecimalPlaces

Verifica quantas casas decimais são necessárias para executar a subtração da cifra. Essa função é executada contando quantas casas decimais existem na variável soma, dividindo por múltiplos de 10 até que o valor seja maior que a soma. Então, é enviada a quantidade para o gerenciador retornar um fragmento da cifra. Levando em consideração, que a última operação é uma subtração não é necessário enviar o valor completo da cifra.

```java
public boolean sendDecimalPlaces(APDU apdu, byte[] buffer) {

    try {
        short decPlaces = countDecimalPlaces(somaSk);

        cipherAES.init(aesKey, Cipher.MODE_ENCRYPT, new byte[16], (short) 0, (short) 16);

        byte[] balanceBytes = new byte[16];
        balanceBytes[14] = shortToBytes(decPlaces)[0];
        balanceBytes[15] = shortToBytes(decPlaces)[1];

        cipherAES.doFinal(balanceBytes, (short) 0, (short) 16, buffer, (short) 0);
        sendData(apdu, buffer, (short) 16);

        return true;

    } catch (Exception e) {
        ISOException.throwIt(ISO7816.SW_WRONG_DATA);
    }

    return false;
}
```

Figura 17 - SendDecimalPlaces. Fonte: própria.

```java
public short countDecimalPlaces(short numb) {

    short test = 10, decPlace = 1;
    boolean finish = false;

    while (!finish) {
        if (numb < test)
            finish = true;
        else {
            decPlace++;
            test *= 10;
        }
    }

    return decPlace;
}
```

Figura 18 - CountDecimalPlaces. Fonte: própria.

2 ExecutePlusSkExpand

Faz a leitura da cifra expandida, multiplica os valores por s0 e s1, e então o resultado é somado na variável soma. Além disso, faz o tratamento das casas decimais. O tratamento é executado contando 5 casas decimais de uma variável *short*, sendo o valor máximo dela de 32000, qualquer valor acima de 9999 é subtraído e somado 1 ao valor inteiro.

```java
public boolean executePlusSkExpand(byte[] buffer) {

    try {
        byte b0 = 0;

        if (listIgnoreSk0[indexSk0Verify] == indexMatrixSk0) {

            byte[] expand = new byte[69];
            Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA), expand, (short) 0, (short) 69);

            for (short i = 0; i < listIgnoreSk1.length; i++) {
                if (i != 0 && listIgnoreSk1[i] == 0)
                    break;

                somaSk += Util.makeShort((byte) 0, expand[listIgnoreSk1[i] * 3]);
                short tempShortSk1 = Util.makeShort(b0, expand[(listIgnoreSk1[i] * 3) + 1]);
                short tempShortks2 = Util.makeShort(b0, expand[(listIgnoreSk1[i] * 3) + 2]);

                short tempShortSkFra = (short) ((tempShortSk1 * 100) + tempShortks2);

                somaSkFra += tempShortSkFra;

                if (somaSkFra >= 10000) {
                    somaSkFra -= 10000;
                    somaSk++;
                }

            }

            indexSk0Verify++;
        }

        indexMatrixSk0++;

        if (indexMatrixSk0 == 23) {
            roundSomaSk();
            return true;
        }
    } catch (Exception e) {
        ISOException.throwIt(ISO7816.SW_WRONG_DATA);
    }
    return false;
}
```

Figura 19 - ExecutePlusSkExpand. Fonte: própria.

4 ExecuteMenusSum

Esse método faz a subtração da cifra pela soma, e então executa a operação mod 2, além disso executa o deslocamento dos bits nos bytes do pin de acordo com o resultado obtido.

```
public boolean executeMenusSum(byte[] buffer) {
        try {
            byte[] sum = new byte[16];
            Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA), sum, (short) 0, (short) 16);
            short countDec = Util.makeShort(sum[14], sum[15]);
            short dec = (short) (countDec / 2);
            short mult = 1;
            for (short i = 0; i < dec; i++) {
                mult *= 10;
            }
            short sumShort = 0;
            int j = 0;
            while (j < countDec) {
                short sumAux = Util.makeShort(sum[j], sum[j + 1]);
                sumShort += sumAux * mult;
                mult = (short) (mult / 100);
                j += 2;
            }
            short result = (short) ((sumShort - somaSk) % 2);
            if (result == 0) {
                pinByteAux = (byte) (pinByteAux << 1);
            } else {
                pinByteAux = (byte) (~pinByteAux);
                pinByteAux = (byte) (pinByteAux << 1);
                pinByteAux = (byte) (~pinByteAux);
            }
            pinByteCount++;
            if (pinByteCount % 8 == 0) {
                pinDecrypt[(pinByteCount / 8) - 1] = pinByteAux;
                pinByteAux = 0;
            }
            return true;
        } catch (Exception e) {
            ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }
        return false;
    }
```

Figura 20 – ExecuteMenusSum. Fonte: própria.