Criptografia Homomórfica Utilizada para Autênticação em Smart Card

Lucas Zanco Ladeira
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil.

Edward David Moreno Ordonez
Universidade Federal de Sergipe
Aracajú, Sergipe, Brasil.

***Abstract*— Smart cards have the capacity to store one or many applications and execute them as they are requested. These applications may have a purpose to store personal information, and even execute financial transactions. In case of financial applications, it is necessary to use authentication so that if the card is stolen or lost it can't be used. Thus, the authentication information is of vital importance to block the usage of unauthorized people. That makes necessary to keep the transmission of the authentication information to the smart card secure, this is possible using cryptographic algorithm. The project objective is to use an algorithm of homomorphic cryptography to mantain secret the PIN during transmission to the card.**

***Index Terms*— Smart Card, Java Card, Authentication, Homomorphic Cryptography**

***Resumo*— *Smart cards* tem a capacidade de armazenar uma ou múltiplas aplicações e as executar quando requisitadas. Essas aplicações podem compreender armazenamento de dados pessoais, até mesmo execução de transações financeiras. No caso de aplicações financeiras, é necessário fazer o uso de autenticação para que se o cartão for roubado ou perdido o mesmo não possa ser utilizado. Sendo assim, a informação autenticadora é de vital importância para barrar utilização não autorizada. Isso torna necessário manter a transmissão da mesma ao *smart card* segura, fazendo o uso de algoritmos criptográficos. O objetivo do projeto é de utilizar um algoritmo de criptografia homomórfica para tornar sigiloso um PIN durante a transmissão ao cartão.**

***Index Terms*— Smart Card, Java Card, Autenticação, Criptografia Homomórfica.**

I. INTRODUÇÃO

C. *Smart Card*

Sistemas embarcados são sistemas que foram criados para executar tarefas específicas, encapsulados, podendo ter tamanho bastante reduzido ao comparar com computadores pessoais. *Smart card* é um exemplo de sistema computacional embarcado, compacto, capaz de armazenar aplicações e as executar ao serem requisitadas. Esse tipo de cartão possui processador, memória volátil, memória persistente, sistema de entrada e saída [1].

Um problema encontrado no desenvolvimento para essa tecnologia é a capacidade de processamento e armazenamento de informações limitado. O clock dos processadores desses cartões podem variar de 3,5712 mhz até 4,9152 mhz, em cartões mais simples, para que se mantenha estável o processamento [1]. Além disso, é possível encontrar cartões com não muito mais de 80 Kb de memória persistente. Isso impossibilita prover, nos mesmos, aplicações como bancos de dados pequenos, ou até mesmo cálculos matemáticos com grandes inteiros.

Para facilitar o desenvolvimento à essa tecnologia foi inserida uma máquina virtual java em um tipo de *smart card*, sendo queos cartões que possuem essa característica são chamados de *java cards*. O uso dessa máquina virtual possibilita o desenvolvimento de aplicações na linguagem java para cartões inteligentes. Onde, dentro de um mesmo cartão podem ter inúmeras aplicações [1].

Essas aplicações são chamadas de *applets*, e geralmente compreendem a execução de transações financeiras pelo crédito armazenado no cartão, ou até mesmo fornecimento de dados pessoais para autenticação [2][3][4]. Uma característica facilmente encontrada nesses aplicativos é a presença de autenticação para liberar acesso aos serviços fornecidos. Isso torna necessária a utilização de algoritmos criptográficos para tornar a informação autenticadora sigilosa, além disso, deve ser autenticada para validar a alteração da mesma.

Para ilustrar a arquitetura existente em um *java card* em relação ao hardware do *smart card* e as aplicações observe a figura 1. Na figura da camada *Java Card Firewall* para baixo o programador não tem acesso, sendo que apenas pode desenvolver os *applets* e instalá-los no cartão.

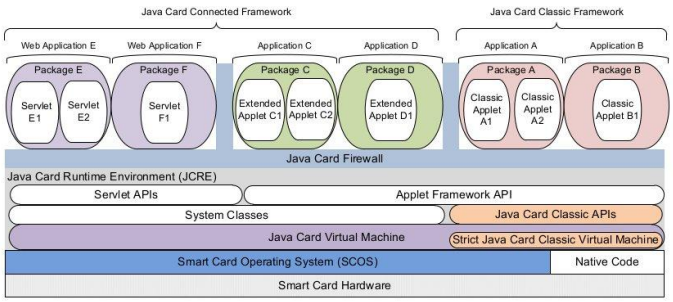

Figura 1- Representação Genérica da Arquitetura *Java* Card 3. Fonte: [5]

D. *Java Card*

Para que dispositivos possam se comunicar com as aplicações contidas em *java cards* é utilizado o protocolo APDU (*Application Protocol Data Unit*). A grosso modo, ele é um conjunto de *bytes* hexadecimais, sendo que, cada *byte* expressa informação sobre qual aplicação deve ser executada, qual método da mesma, o dado que está sendo recebido/enviado, ou faz parte do dado [1]. O tamanho máximo de *bytes* existentes em uma chamada é de 256 *bytes*, sendo necessário enviar mais vezes caso exceda esse valor.

O cabeçalho do APDU de envio é dividido em 6 campos: *CLA*, *INS*, *P1*, *P2*, *LC*, *LE*. O primeiro diz qual aplicação (classe) está sendo chamada para execução. O *INS* identifica qual método da classe está sendo chamado. De forma parecida com *INS*, os parâmetros P1 e P2 são usados como *flags* para

validar diferentes caminhos de execução dentro de uma mesma função. Por último, LC e LE informam o tamanho do dado recebido e o tamanho do dado enviado esperado como retorno, respectivamente. O resto dos *bytes* compreendem o campo do próprio dado a ser recebido, chamado *DATA FIELD* [1].

Após a execução é retornado um APDU diferente como resposta, o mesmo possui apenas três campos: *RESPONSE DATA*, *SW1*, *SW2*. O *SW1* e *SW2* informam se ocorreu algum erro, caso tenha executado a função requisitada com sucesso o valor "9000" estará contido nesses campos. Como o próprio nome já diz o campo *RESPONSE DATA* armazena o resultado, caso exista, da funcionalidade [1].

O processo de compilação, instalação e execução dos *applets* no *java card* utilizam duas máquinas virtuais. Uma delas fora do cartão com o objetivo de compilar o código fonte e a outra dentro do cartão para executar o mesmo. A primeira converte o arquivo binário java *.class* para *.cap* que o cartão tem a capacidade de interpretá-lo [1].

A segunda máquina virtual utiliza um interpretador para executar as aplicações, a mesma deve controlar as sessões dos *applets*, e gerenciar a memória para que um *applet* não acesse a partição de outro. Esse gerenciamento é possível através do AID (*Application Identifier*), que é a identificação da aplicação. O mesmo é dividido em dois vetores, o primeiro com cinco *bytes* para identificar a empresa desenvolvedora, e o segundo, com 0 à 11 *bytes*, para diferenciar uma aplicação de outra de uma mesma empresa [1]. De forma simplificada essas operações são apresentadas na figura 2.

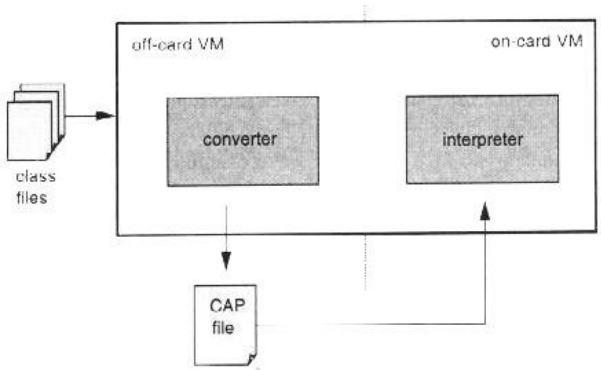

Fig 2. Java Card Virtual Machines. Fonte: [1]

E. Criptografia

A necessidade de manter determinada informação secreta iniciou pela expansão do uso de computadores pessoais. Onde, foram encontradas vulnerabilidades nos sistemas, possibilitando a extração remota de informações pela internet [6]. Sendo assim, dados pessoais, empresariais, e até mesmo informações bancárias estariam comprometidas sem a utilização da criptografia.

A criptografia faz o uso de esquemas matemáticos para transformar um texto claro em um texto codificado. Da forma mais simples, isso é possível através do uso de uma chave para codificar o dado, e apenas quem possuir a chave pode decodificar o dado. A quantidade de chaves de um esquema de criptografia divide o mesmo em dois grandes grupos: criptografia simétrica e criptografia assimétrica [7].

Esquemas de criptografia simétrica utilizam apenas uma chave para codificar e decodificar o dado. Caso um atacante obtenha a chave qualquer dado transmitido se torna público ao atacante [7]. Digamos que Rodolfo e Leonardo queiram conversar com o uso desse tipo de algoritmo, primeiramente é necessário transferir a chave de forma segura de um para o outro. Após o envio da chave, Rodolfo codifica o dado com a chave, e envia ao Leonardo por um meio inseguro. Dessa maneira, apenas Leonardo tem condições de decodificar a cifra e descobrir qual o texto claro. Alguns exemplos de algoritmos de criptografia simétrica são: DES, 3DES, AES.

Esquemas de criptografia assimétrica utilizam uma chave para codificar (chave pública) o dado e outra para decodificar (chave privada). Onde, esse tipo de criptografia também é chamada de criptografia de chave pública. Algoritmos assimétricos se tornaram populares por não necessitarem que a chave que codifica seja enviada por um meio seguro [7]. Sendo assim, caso Rodolfo queira enviar uma mensagem à Leonardo, primeiramente ele deve requisitar a chave pública de Leonardo. Então Rodolfo codifica a informação com a chave pública e envia a Leonardo. Apenas com a chave privada de Leonardo a cifra pode ser decodificada.

F. Criptografia Homomórfica

A principal característica de um algoritmo criptográfico homomórfico é a capacidade de executar operações matemáticas no dado codificado. Isso possibilita o envio do dado para processamento na nuvem, caso seja necessária uma máquina com maior poder de processamento. Essas operações não necessitam da chave privada, sendo assim o computador utilizado na nuvem não tem a capacidade de decodificar o dado [8].

Esquemas criptográficos homomórficos podem ser divididos em criptografia parcialmente homomórfica (SWE, Somewhat Homomorphic Encryption), e criptografia totalmente homomórfica (FHE, Fully Homomorphic Encryption). Na criptografia SWE é possível executar apenas uma das operações matemáticas, adição ou multiplicação. Diferente disso, na criptografia FHE é possível executar tanto a operação de adição como a de multiplicação [8].

Os algoritmos FHE podem ser subdivididos em algoritmos baseados em anéis de inteiros, que utiliza inteiros para a geração das chaves e codificação dos dados. Baseado em reticulados, os primeiros esquemas de Gentry [9][10] utilizam esse tipo de algoritmo. E, LWE (*Learning with errors*), que utiliza problemas matemáticas sobre aprendizagem de máquina.

Ao executar operações em um cifra, ocorre um crescimento no número de ruído nela. Dessa maneira, ao executar algumas operações pode ser não possível obter o resultado correto após a decodificação do dado. Já em 2009 foi publicada uma implementação de FHE (Fully Homomorphic Encryption) criada por Gentry [9]. Nessa implementação era possível executar um número arbitrário de vezes operações em uma cifra, por causa de uma operação de redução de ruído chamada de Bootstraping. Essa operação é possível pela capacidade desse esquema avaliar o seu próprio circuito de decodificação.

Além disso, uma parte da chave secreta é adicionada à chave pública, sendo possível decifrar com um polinômio de baixo grau.

Uma desvantagem dessa implementação é que o tamanho da chave pública é de $\Theta(\lambda^{10})$. Limitando a quantidade de dispositivos com memória suficiente para armazenar a chave.

Contudo, na Eurocrypt de 2011, Coron et al [11] publicaram uma implementação FHE com o nome de "*Fully Homomorphic Encryption over the Integers with Shorter Public Keys*". Como o nome já diz essa implementação possui tamanho de chave pública reduzida, sendo $\Theta(\lambda^{7})$. A ideia da execução é armazenar um subconjunto da chave pública e então gerar, durante a execução, a mesma combinando os elementos do subconjunto de modo multiplicativo. Aumentando o número de dispositivos com a capacidade de armazenar a chave pública.

Algoritmos homomórficos apresentam vantagens de acordo com dificuldade de quebra da cifra, impossibilidade da utilização de algoritmos pré-quânticos para quebra de chaves, entre outros. Um exemplo de algoritmo pré-quântico é o algoritmo de Shor [12] que deriva a chave privada a partir da chave pública em esquemas de fatoração de inteiros. Portanto, é possível garantir a confidencialidade de um dado durante a transmissão do mesmo. Um exemplo da utilização desses algoritmos é na autenticação de um usuário pois, se tratando de dados pessoais é necessário que ele se mantenha em segredo.

II. APLICAÇÃO DE ESQUEMAS HOMOMÓRFICOS PARA AUTENTICAÇÃO

G. Customisation of Paillier Homomorphic Encryption for Efficient Binary Biometric Feature Vector Matching

Um exemplo de projeto utilizando essa criptografia para autenticação é no trabalho de [13], onde é utilizada a íris para autenticar um indivíduo. Esse processo é dividido em 2 partes, análise e identificação. Na análise é armazenado um ou mais *templates* de uma determinada íris. Antes do armazenamento são executados shifts cíclicos em cada *template*, seguindo o número de shifts executados, para aumentar a taxa de correspondência. Após isso, cada *template* é codificado e armazenado em um banco de dados.

A fase de identificação é iniciada pelo cliente, lendo a íris, codificando a mesma, e requisitando a autenticação. Essa íris é enviada a um servidor de autenticação, onde o mesmo faz a busca no banco de dados as íris armazenadas. O servidor de autenticação executa uma operação "*xor*" entre a íris enviada pelo cliente e cada *template* armazenado referente aquele indivíduo. Após isso, um terceiro servidor faz o cálculo do peso por Hamming utilizando a chave privada. Então é retornado ao cliente se o acesso foi permitido ou não, dependendo do peso calculado.

Nesse trabalho é alterado o esquema de [14] na operação de criptografia para execução de um "*xor*" em uma *string* binária. Para executar essa operação é necessário fazer a codificação bit a bit, o que torna o algoritmo ineficiente. Então, foi proposta uma nova forma de executar o "*xor*" de dois bits para melhorar a eficiência do algoritmo. Como o propósito da citação desse artigo é para exemplificar os usos da criptografia homomórfica para autenticação não serão explicadas mais profundamente essas alterações, para isso é necessário consultar o trabalho de [13].

H. Homomorphic Encryption Based Cancelable Biometrics Secure against Replay and Its Related Attack

Outro exemplo da utilização de criptografia homomórfica para autenticação é no trabalho de [15]. Nele é apresentado um protocolo para autenticação combinando o protocolo de [16], com o conceito de autenticação *challenge-response*. Esse protocolo é dito seguro, contra ataques de replicação, por utilizar dados randômicos gerados pelo servidor de autenticação a cada nova autenticação.

Além disso, alguns requisitos devem ser mantidos seguindo o protocolo de [16]: Os usuários utilizam a chave pública para codificar a biometria, não tendo acesso à chave privada. O servidor de autenticação não deve ter acesso a informações além de se a biometria foi aceita ou não. O decodificador não deve ter acesso a informações além da distância dos vetores biométricos.

Levando em consideração os algoritmos citados anteriormente é possível observar a capacidade da utilização da criptografia homomórfica para autenticação, a mesma pode ser com biometria por digital, biometria por íris, entre outras formas de autenticação.

No nosso projeto é utilizada a autenticação por PIN (senha de geralmente 8 caracteres), essa senha não é uma característica física do usuário, tornando fácil a replicação da mesma. Dessa forma, é necessário a garantia de sigilo durante a transmissão.

III.OBJETIVO DO PROJETO

Existe a necessidade de manter o dado utilizado em autenticações confidencial, dessa maneira restringe o uso de algum serviço ao usuário proprietário. Sendo assim, o objetivo do projeto é de garantir essa confidencialidade utilizando um algoritmo de criptografia homomórfica. Onde o dado utilizado na autenticação é o PIN. Observando os exemplos de aplicações que fazem uso dessa criptografia é necessário analisar a capacidade de decodificação dentro de um *smart card* e a eficiência da mesma.

Levando em consideração, a utilização da criptografia homomórfica pós-quântica em computadores com poder computacional elevado, e paralelização entre CPUs e GPUs, a execução de uma primitiva em um *java card* com poder computacional e memória disponível restritos se torna um desafio. Sendo assim, esse projeto quebra o paradigma de que não é possível utilizar essa criptografia em sistemas embarcados compactos como cartões inteligentes.

IV. ARQUITETURA DESENVOLVIDA

Para a execução da decodificação dentro do *smart card* primeiramente é necessário gerar as chaves criptográficas e codificar o dado utilizado na autenticação. Para isso foi criada uma aplicação com as primitivas da criptografia homomórfica implementadas. Para comunicação com o cartão foi criado um gerenciador do *applet*. E por último, o próprio *applet*. A separação entre o gerenciador e o módulo de criptografia foi necessário para a reutilização do código do mesmo. Além disso, esse módulo foi desenvolvido como um webservice, tor-

nado possível chamadas remotas ao mesmo.

Dessa maneira, o webservice tem os serviços de geração das chaves criptográficas, codificação do PIN, e expansão da cifra. O gerenciador deve requisitar os dados necessários para a decodificação ao webservice, e tratar os mesmos para envio ao *applet*, sendo que a cada comunicação com o cartão é possível enviar apenas 255 *bytes*. O *applet* deve decodificar o PIN utilizando a chave privada homomórfica, autenticar o usuário e liberar o acesso a qualquer outra funcionalidade implementada no cartão.

Para facilitar o entendimento da divisão presente no projeto observe a figura 3. As funcionalidades são divididas em dois grupos: o primeiro gera todas as informações necessárias para a decodificação, e inicializa o cartão. Sendo que, durante a inicialização do *applet* a chave privada é armazenada no mesmo. O segundo requisita a cifra e a cifra expandida e faz o tratamento desses dados, envia os mesmos para o cartão, decodifica o PIN e autentica o usuário.

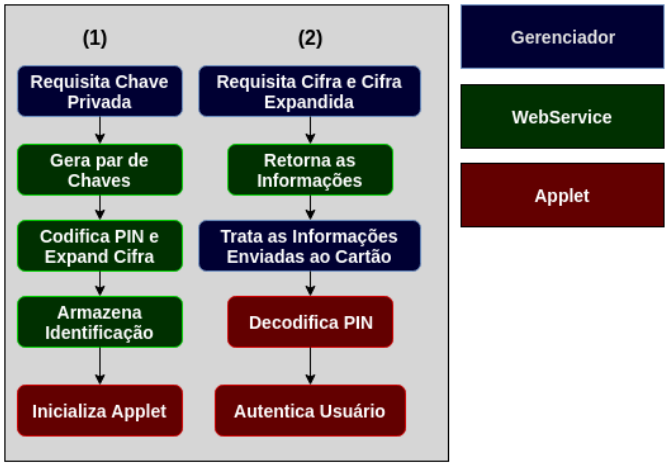

Figura 3 - Arquitetura Implementada.

V. IMPLEMENTAÇÃO DA AUTENTICAÇÃO

É necessário aprofundar na explicação de como é executada a codificação, expansão e decodificação. Isso torna possível o entendimento de como foram fragmentados os dados enviados ao cartão para correta execução da decodificação. Uma observação necessária é a de que o nível de segurança utilizado é o *small*, sendo que a implementação segue os parâmetros do mesmo.

A.Codificação e Expansão

O esquema de [11] faz a codificação bit a bit, levando em consideração que o PIN utilizado no projeto tem oito *bytes*, é necessário executar 64 codificações. Além disso, como cada cifra deve ser expandida, são executadas 64 expansões. Para persistir essas informações no webservice todas as cifras são concatenadas em uma *string* com um caractere para dividi-las, e a mesma operação é feita para as cifras expandidas em outra *string*. Dessa maneira, antes de enviar ao cartão é executado o tratamento desses dados para fragmentar cada cifra e cifra expandida.

B. Decodificação

Antes que o gerenciador envie os dados ao cartão para que seja executada a decodificação o mesmo deve passar por um tratamento. As cifras e cifras expandidas devem ser separadas umas das outras, e subdivididas para envio ao cartão.

Um exemplo é que a cifra expandida é composta por matrizes de 23x23 com valores fracionários. Sendo que, apenas é possível enviar 255 *bytes* por vez, os valores são tratados e divididos em três: parte inteira, duas primeiras casas decimais, e duas últimas casas decimais. Para facilitar o desenvolvimento cada linha é enviada por vez como um vetor de 69 valores (*23*3*). Um outro ponto que é necessário citar é de que para enviar todas as cifras expandidas ao cartão são executados 1472 envios, sendo 23 linhas da matriz por 64 cifras expandidas.

Para melhorar a eficiência da aplicação algumas linhas da matriz podem ser ignoradas durante a execução da operação de multiplicação da decodificação. A chave privada é composta por dois vetores de números binários (0 ou 1), digamos que os dois vetores que compreendem a chave pública são:

$$s0 = [1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1]$$
$$s1 = [0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1]$$

Nesse caso, ao executar a operação que compreende a multiplicação de um index *i* do vetor *s0*, um index *j* do vetor *s1*, e um index *ij* da matriz (cifra expandida, expand):

$$soma += s0[i] * s1[j] * expand[i][j]$$

Apenas será somado algum valor diferente de zero quando o valor do index de *s0* e de *s1* for um. Sendo assim, qualquer linha onde o valor contido no index do *s0* for zero pode ser ignorada, pois nenhum valor de *s1* será levado em consideração. No caso do exemplo apenas é necessário executar a soma para quatro linhas, e no total 16 valores serão somados.

Por último, para não ser necessário o envio de toda a cifra para o *applet* foram analisadas as operações onde é utilizada a mesma. As operações são as seguintes:

$$(cifra - soma)\ mod\ 2$$

Levando em consideração que a operação executada primeiramente é uma soma, é apenas fundamental o envio de um valor com uma casa decimal acima do valor da soma, caso essa casa decimal seja diferente de zero. Portando, é feita a contagem das casas decimais da soma e requisitada a quantidade necessária da cifra para efetuar a operação. Dessa maneira não é obrigatório o envio da cifra completa, pois ao executar *mod 2* apenas a casa decimal menos significativa é levada em consideração.

VI. TEMPO DE EXECUÇÃO

Para verificar a eficiência da execução da primitiva *decrypt* no *java card*, foram executados testes unitários e obtidos os tempos de execução. Os testes compreendem a análise do tem-

po de execução das funções dentro do cartão, sendo elas decodificação e autenticação, sendo executados 100 vezes. As mesmas foram testadas de duas maneiras: fazendo o uso do esquema homomórfico e apenas com o algoritmo AES.

É importante citar que mesmo sem a utilização do esquema homomórfico todos os dados transmitidos ao cartão são enviados dentro de uma sessão segura, portanto estão codificados pelo algoritmo AES, sendo o dado autenticador ou não.

A configuração do computador utilizado durante os testes é a seguinte: Samsung Ativ Book 6 com processador i5 3230M, Memória Ram 8 GB 1600 Mhz, HD com 1 TB 5400 RPM. Os mesmos foram executados dentro de um simulador de *java cards* chamado *jcardsim*, ele possui implementada a API JCDK, algoritmos criptográficos, e simula a execução de uma aplicação real. Além disso, compreende tanto a execução do *applet*, como a instalação e tratamento de exceções.

A figura 4 apresenta os dados obtidos pelos testes. O tempo foi medido em nanosegundos e a média dos mesmos foi calculada com o quartis 3. Esse método faz o uso de 75% dos dados, ignorando os picos e baixas dispersantes. O teste com criptografia homomórfica obteve tempo médio de execução de 7771408 nanosegundos, e sem homomorfismo de 21414,8. Analisando os resultados é possível observar uma grande diferença entre os dois algoritmos, sendo que com homomorfismo é aproximadamente 363 vezes maior que o outro.

Mesmo com uma diferença tão gritante dos resultados, ainda mostra a capacidade de execução da primitiva em *smart cards*, sendo que no simulador demorou apenas 7,771408 milissegundos. Para conversão desse tempo em um cenário real foi calculada a conversão apenas de acordo com o *clock* máximo do processador.

Portanto, foram utilizados os valores 3,2 Ghz que é o *clock* máximo do processador do computador, e 4,9152 Mhz para um cartão simples [1]. Dessa maneira, a conversão do tempo de acordo com o *clock*, para a execução dentro de um cartão, foi calculada assim:

$$timeres = \frac{pcclock * 1000 * time}{cardclock}$$

$$timeres = \frac{3{,}2 * 1000 * 7{,}771408}{4{,}9152}$$

$$timeres = 5059{,}510416667$$

Sendo, *timeres* o tempo resultante, *pcclock* o *clock* do computador, *cardclock* o *clock* do cartão e *time* o tempo encontrado nos testes. O resultado obtido foi de 5059,51 milissegundos aproximadamente, convertendo para segundos temos 5,05951 para decodificar e autenticar um usuário com criptografia homomórfica. Esse resultado se mostra promissor, sendo que esquemas homomórficos são utilizados geralmente em computadores com grande poder computacional, e não em sistemas restritos como *smart cards*.

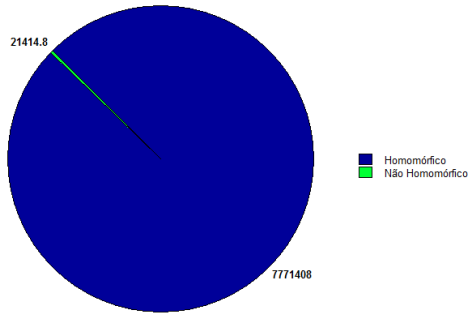

Figura 4 - Tempo de Execução.

VII. MEMÓRIA UTILIZADA

Além do tempo de execução foi calculada a quantidade de memória necessária para armazenar as chaves utilizadas na decodificação, cifra, e cifra expandida. Primeiramente foi analisada a memória utilizada para a autenticação sem homomorfismo, para isso é necessário um vetor de 256 *bytes* compreendendo a chave AES, um *buffer* com 22 *bytes* sendo *CLA* (1 *byte*), *INS* (1 *byte*), *P1* (1 *byte*), *P2* (1 *byte*), *LE* (1 *byte*), *LC* (1 *byte*) e *DATA FIELD* (16 *bytes*).

Como o algoritmo utilizado é o AES para codificar é necessário um vetor de entrada de 16 *bytes*, sendo assim os 8 *bytes* do PIN são concatenados com si próprio resultando PIN + PIN de 16 *bytes*. Por último o vetor com o PIN real do usuário com 8 *bytes*. A soma total é de 286 *bytes*, apenas considerando o tamanho da chave AES, cifra e *buffer*.

O homomorfismo requer, além da cifra, a cifra expandida que é uma matriz de 23x23 no nível de segurança *small*. Para que não seja necessário armazenar a matriz inteira é feito o envio linha por linha, possibilitando a reutilização do espaço na memória. Como já foi explicado no capítulo “V" cada linha ocupa 69 *bytes*.

Para o cálculo da soma são utilizadas quatro variáveis *short*, onde armazenam os três tipos de fragmento da cifra expandida, e a soma total. Além disso, dois *shorts* para armazenar os *index* da matriz e vetores da chave. A chave privada homomórfica compreende dois vetores de *bytes* com tamanho 22, e são armazenados por inteiro no cartão.

Após a computação da soma, é executada a subtração da cifra pela soma e *mod 2*, isso necessita de dois *shorts*, um para armazenar a cifra, e outro para armazenar o resultado. Por fim, dois vetores de *bytes* são necessários, um para armazenar o PIN real do usuário, e o outro o PIN resultante da decodificação, ambos de 8 *bytes*.

Uma observação válida é a de que mesmo usando o esquema homomórfico todos os dados são enviados dentro de uma sessão segura, sendo necessário considerar a chave AES além da chave homomórfica. Também, a cada envio ao cartão existe o tamanho do *buffer* que muda de acordo com a informação enviada, sendo considerado de tamanho 6 *bytes* (*header*), e reutilizado para todas as requisições. O resultado encontrado no cenário homomórfico é de: 397 *bytes* e 8 *shorts*.

Considerando apenas os *bytes*, pois a quantidade de *bytes* que ocupa uma variável *short* depende do valor da mesma, é possível comparar ambas as implementações. Observando a

figura 5, é possível verificar que a diferença de memória utilizada entre as diferentes implementações não é grande como a de análise de tempo. Isso é possível através da divisão das operações em menores, e a reutilização de espaço de memória.

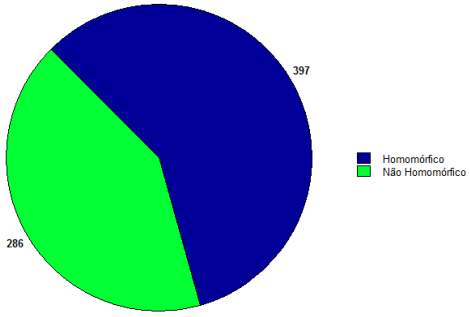

Figura 5 - Memória Utilizada.

VIII. CONCLUSÃO

Os testes mostram a capacidade da execução da decodificação homomórfica em um *java card*. Isso foi possível através da fragmentação das operações e dos dados enviados ao cartão. Além disso, o tempo encontrado se mostrou promissor sendo possível a decodificação e autenticação em poucos segundos. A quantidade de memória utilizada compreende uma configuração simples de um *smart card*, necessitando de menos de 1 Kb de memória.

Durante os cálculos de tempo não foi levada em consideração a quantidade de ciclos necessária para a execução de cada operação, podendo variar entre a execução no simulador e em um *java card* real. Além disso, os programas executados de forma concorrente no computador também não foram levados em consideração, sendo IDEs, serviço de banco de dados, entre outros.

O objetivo da implementação é de executar a decodificação da maneira mais rápida, e avaliar o tempo obtido. Sendo assim, não foram levados em consideração todos os ataques existentes. Um exemplo é o de canais laterais, com a capacidade de analisar o tempo de execução de cada operação na tentativa de obter algum dado correspondente a chave privada, ou até mesmo o texto claro do PIN.

REFERÊNCIAS

[1] Z. Chen, “Java Card Technology for Smartcards: Architecture and Programmer's Guide”, Addison-Wesley Professional, setembro 2000.

[2] N. Munjal, R. Moona, “Secure and Cost Effective Transaction Model for Financial Services”, Department of Computer Science and Engineering Indian Institute of Technology, outubro 2009.

[3] H. Park, K. Chun, S. Ahn, “The Security Requirement for off-line E-cash system basedon IC Card”, junho 2005.

[4] H. Kim, S. Lee, I. Shin, “Design and Implementation of In-House Electronic Money Using Java Cards”, 2013.

[5] R. N. Ankram, K. Markantonakis, “Smart Cards: State-of-the-Art to Future Directions”, IEEE International Symposium on Signal Processing and Information Technology, dezembro 2013.

[6] D. RUIU, “Learning from Information Security History, IEEE Security & Privacy”, fevereiro 2006.

[7] J. Andress, “The Basics of Information Security”. Syngress, 2014.

[8] L. C. Santos, “Implementação do Esquema Totalmente Homomórfico Sobre Inteitos com Chave Reduzida”, 2014.

[9] C. Gentry, “A Fully Homomorphic Encryption Scheme”. Ph.D. thesis, Stanford University, 2009. Disponível em: http://crypto.stanford.edu/craig. Acessado em: 18/03/2015.

[10] C. Gentry, e S. Halevi, “Implementing Gentry's Fully-homomorphic Encryption Scheme”, Advances in Cryptology-EUROCRYPT 2011, pp. 129-148, 2011

[11] J. S. Coron, A. Mandal, D. Naccache, e M. Tibouchi, “Fully Homomorphic Encryption over the Integers with Shorter Public Keys”. Em P. Rogaway (Ed.), CRYPTO 2011, LNCS, vol. 6841, Springer, pp. 487-504. Full version available at IACR eprint, 2011.

[12] S. M. HAMDI, S. T. Zuhori, F. Mahmud, B. Pal, “A Compare between Shor's quantum factoring algorithm and General Number Field Sieve”, International Conference on Electrical Engineering and Information & Communication Technology (ICEEICT), abril 2014.

[13] G. M. Penn, G. Potzelsberger, M. Rohde, A. Uhl, “Customisation of Paillier Homomorphic Encryption for Efficient Binary Biometric Feature Vector Matching”, setembro 2014.

[14] P. Paillier, “Public-key Cryptosystems Based on Composite Degree Residuosity Classes”, Advances in cryptology - EUROCRYPT’99. Springer Berlin Heidelberg, 1999.

[15] T. Hirano, M. Hattori, T. Ito, N. Matsuda, T. Mori, “Homomorphic Encryption Based Cancelable Biometrics Secure against Replay and Its Related Attack”, outubro 2012.

[16] M. Hattori, Y. Shibata, T. Ito, N. Matsuda, K. Takashima, T. Yoneda, “Provably-secure Cancelable Biometrics Using 2-DNF Evaluation”. Journal on Information Processing, 20(2):496–507, 2012.