Implementação e Desempenho em *Smart Card* do Algoritmo Criptográfico SHA-3

Fábio Dacêncio Pereira
Universidade Federal de Sergipe
Aracajú, Sergipe, Brasil.

Cesar Giacomini Penteado
Universidade Federal do ABC
Santo André, São Paulo, Brasil.

Abstract - The Information Integrity is one of the goals related to information security that stands out in the current scenario with the advent of the Internet. One of the techniques and methods to ensure the integrity of information is the use of hash function, which generates a byte string (hash) that should be unique. But much of the current functions can no longer prevent malicious attacks and ensure that the information has only one hash to identifying digital content. Through a public call the National Institute of Standards and Technology (NIST) called technological centers, companies and the scientific community to submit proposals for the new standard hash function, called SHA-3. Once the candidates presented, they are exposed and evaluated in several respects. In this process, the Keccak algorithm was winner of the call in 2012. This paper presents the SHA-3 implemented in smart cards, in order to obtain performance data for future comparison with related work.

Resumo - A Integridade da Informação é uma das metas relacionadas à segurança de informações que se destaca no cenário atual com o advento da Internet. Uma das técnicas e métodos para garantir a integridade de uma informação é a utilização de funções de hash, que gera uma cadeia de bytes (hash) que deve ser única para identificar um conteúdo digital. Porém grande parte das funções atuais já não consegue evitar ataques maliciosos e garantir que a informação tenha apenas um hash. Por meio de uma chamada pública o National Institute of Standards and Technology (NIST) convocou centros tecnológicos, empresas e a comunidade científica para enviar propostas para o novo padrão de função de hash, chamado de SHA-3. Uma vez apresentados os candidatos, estes são expostos e avaliados em diversos aspectos. Neste processo, o algoritmo Keccak foi vencedor da chamada em 2012. Neste trabalho apresenta-se o algoritmo SHA-3 implementado em smart cards, com o intuito de obter de dados de desempenho para futura comparação com trabalhos correlatos.

I. INTRODUÇÃO

Com o advento da Internet serviços como comunicação, entretenimento, negócios, entre outros, fizeram da rede mundial de computadores um arranjo computacional complexo e organizado de recursos e pessoas. Os serviços, as facilidades e a eficiência da Internet são atrativos que levam esta a uma evolução contínua. Entretanto, problemas que antes não eram considerados, atualmente tornaram-se uma preocupação de usuários e empresas, como por exemplo, a segurança de informações neste ambiente.

Neste trabalho foi destacada uma das soluções para mitigar problemas relacionados a um serviço importante na área de segurança de informações, a integridade da informação. A integridade da informação é uma das metas relacionadas à segurança de informações que se destaca no cenário atual. Uma das técnicas e métodos para garantir a integridade de uma informação é a utilização de funções de hash, que gera uma cadeia de bytes (hash) de tamanho fixo a partir de uma determinada informação. Este hash tem a incumbência de identificar a informação na qual originou este hash de maneira única, onde qualquer mudança mínima desta informação altera completamente o valor do hash.

Porém grande parte das funções atuais já não consegue evitar ataques malíciosos e garantir que a informação tenha apenas um hash. A fim de resolver este problema, por meio de uma chamada pública o National Institute of Standards and Technology (NIST) convocou centros tecnológicos, empresas e a comunidade científica para enviar propostas para o novo padrão de função de hash, chamado de SHA-3. Uma vez apresentados os candidatos, os mesmos serão expostos e avaliados em diversos aspectos.

Neste contexto, este trabalho se propôs a selecionar um dos algoritmos finalistas da chamada para o SHA-3 e posteriormente implementá-lo em um dispositivo smart card, com o intuito de obter de dados de desempenho para comparação com trabalhos correlatos.

II. FUNÇÕES DE HASH

Um dos princípios da criptografia é a integridade, ou seja, garantir que uma informação não sofra qualquer tipo de alteração indesejada no seu armazenamento, transmissão ou apresentação. Uma categoria específica de algoritmos tem a incumbência de tratar e implementar esse tipo de serviço de segurança, são conhecidos como funções de hash.

Funções de hash são funções matemáticas que a partir de uma informação cifrada ou não e de tamanho variável, gera uma cadeia de valores de tamanho fixo, denominado de hash (ou *message digest*). Esta sequência gerada (hash) tem a função de identificar uma informação de maneira única, onde qualquer alteração efetuada na informação, por mínimo que seja, altera o resultado do valor do hash [1].

Funções de hash são operações de único sentido, ou seja, não é possível obter uma informação a partir de seu hash gerado. Além disso, uma informação cifrada tem tamanho similar à informação original (texto claro), já as funções de hash geram apenas um resumo, que não tem relação com o tamanho da informação original.

Uma função de hash H segue algumas características ou propriedades distintas que a diferem de outras funções similares [1], que são:

- Dada uma mensagem M, é fácil gerar seu hash h.
- Dado h, é computacionalmente inviável encontrar M tal que H(M) = h. Esta propriedade é conhecida como resistência de pré-imagem.
- Dado M, é computacionalmente inviável encontrar M' tal que H(M') = H(M). Esta propriedade é conhecida como resistência de segunda pré-imagem.

Uma função de hash resistente a colisões é uma função H que além de satisfazer as características descritas acima, também satisfaz a propriedade [2] em que:

- É computacionalmente inviável encontrar um par M, M' tal que H(M) = H(M'). Esta propriedade é conhecida como resistência a colisões.

Entre os algoritmos de hash utilizados atualmente destacam-se Message-Digest Algorithm 5 (MD5) e o Secure Hash Algorithm (SHA-1 e SHA-2). O MD5 é um algoritmo de hash de 128 bits desenvolvido em 1991 por Ronald Rivest, Adi Shamir e Leonard Adleman (fundadores da RSA Data Security), atualmente é muito utilizado para verificação de integridade de arquivos e logins em softwares que utilizam o protocolo Peer-to-Peer (P2P).

O SHA foi desenvolvido pela National Security Agency (NSA) e publicado pela National Institute of Standards and Technology (NIST) que padronizou a função nos EUA. Os três algoritmos SHA são diferentes estruturalmente e são conhecidos como SHA-0, SHA-1 e SHA-2. Na família SHA-2 existem ainda quatro variantes que possuem estruturas similares, porém geram hashes de tamanhos diferentes que são o SHA-224, SHA-256, SHA-384 e SHA-512, e são utilizados atualmente em aplicações que exigem alta segurança da integridade.

No entanto foram reportados sucessos de ataques tanto para o algoritmo MD5 [3], assim como, para os algoritmos SHA-0 e SHA-1[4], que geram colisões (informações diferentes que produzem hashes iguais), o que fere o princípio das funções de hash, que é a de garantir a integridade de uma informação.

III. SHA-3

A função SHA-2 atualmente continua segura e inquebrável, porém como o mesmo compartilha de uma herança estrutural similar ao seu antecessor, o SHA-1, o torna suspeito e levanta dúvidas quanto a sua segurança.

Como resposta, em 2007 foi aberta uma competição com o princípio de escolher um novo padrão de função de hash. Organizado pelo National Institute of Standards and Technology (NIST), atualmente a competição se encontra em sua terceira e última etapa. A terceira etapa da competição começou no final de 2010, onde cinco algoritmos finalistas foram selecionados. É previsto que o algoritmo finalista seja anunciado e publicado pelo NIST em 2012 [5].

As funções escolhidas para esta terceira etapa são: BLAKE, Grøstl, JH, Keccak e Skein. Por fim, em 2012, fi divulgado o algoritmo vencedor, o Keccak, desenvolvida por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche.

IV. KECCAK

O algoritmo Keccaks apresenta uma estrutura relativamente simples e versátil, um dos pontos destacados em relação aos seus competidores.

O algoritmo Keccak pertence à família de funções de esponja e utilizam esta construção para produzir transformações ou permutações de tamanho fixo com o intuito de criar um algoritmo que a partir de uma entrada de qualquer tamanho gere uma saída de tamanho arbitrário [6]. Um dos grandes atrativos
desta nova função de hash é a de que a mesma pode gerar a partir de uma entrada de tamanho variável uma saída de tamanho infinito. Além disso, funções de esponja possuem relativa segurança contra todos os ataques genéricos existentes [6].

O algoritmo Keccak consiste de duas partes: a função *Keccak-f[b](A,RC),* que realiza as permutações e operações lógicas sobre os dados e a função de esponja *Keccak[r,c](M)*, que organiza e prepara os dados de entrada para realizar a manipulação desses dados pela função de permutação e organiza os valores de saída para gerar o *hash*.

A. Função de Permutação

O Keccak utiliza de técnicas de permutação para gerar o hash. Na função de permutação pode ser escolhida uma das sete permutações disponíveis, denotada como *Keccak-f[b]*, onde $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, que representa a largura de permutações. A largura da permutação é também a largura do estado S na função de esponja.

Os valores do estado são organizados em uma matriz *A* de formato 5x5, que contém 25 posições de tamanho *w* bits, onde $w \in \{1, 2, 4, 8, 16, 32, 64\}$, tal que

$w = b \div 25$,

por exemplo, caso b seja igual a 1600, o tamanho de cada posição da matriz irá ter 64 bits.

Esta função realiza um número de rodadas *nr* onde em cada rodada são realizadas cinco etapas que realizam operações lógicas e permutações de bits nos blocos de dados contidos na matriz *A*. O número de rodadas *nr* depende da largura de permutação, que é dada por

$nr = 12 + 2*l$,

onde $2^l = w$.

Seguindo o exemplo acima, se *w* for igual a 64, o número de rodadas seria igual a 24, portanto em cada uma das 24 rodadas seriam realizadas as cinco etapas de operações presentes na função de permutação.

As cinco etapas citadas anteriormente que irão manipular os dados são referenciadas com letras gregas, que são θ (theta), ρ (rho), π (pi), χ (chi) e ι (iota). Cada uma delas tem objetivos diferentes e maneiras específicas para a manipulação dos dados da matriz.

A Fig. 1 apresenta o pseudocódigo da função de permutação, onde a matriz A contendo blocos de informação e o *RC* (*round constants*) são parâmetros de entradas, e são realizadas as operações lógicas XOR, NOT e AND, permutações e rotações de bits sobre as informações contidas na matriz *A*, que geram ao final da rodada uma nova matriz de saída *A* de tamanho 5x5 com informações operadas pelas cinco etapas.

```
Keccak-f[b](A) {
  forall i in 0…nr-1
    A = Round[b](A, RC[i])
  return A
}
Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   forall x in
0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                    forall x in 0…4
  A[x,y] = A[x,y] xor D[x],                  forall (x,y) in (0…4,0…4)

  ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                 forall (x,y) in
(0…4,0…4)
```

Figura 1. Pseudocódigo de Keccak-f[b].

```
Keccak[r,c](M) {
  Initialization and padding
  S[x,y] = 0,                                  forall (x,y) in (0…4,0…4)
  P = M || 0x01 || 0x00 || … || 0x00
  P = P xor (0x00 || … || 0x00 || 0x80)

  Absorbing phase
  forall block Pi in P
    S[x,y] = S[x,y] xor Pi[x+5*y],   forall (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                           forall (x,y) such that x+5*y < r/w
  S = Keccak-f[r+c](S)
  return Z
}
```

Figura 2. Função de Esponja Keccak[r,c]

de bits sobre as informações contidas na matriz *A*, que geram ao final da rodada uma nova matriz de saída *A* de tamanho 5x5 com informações operadas pelas cinco etapas.

O parâmetro *RC* utilizada pela função de permutação é um vetor contendo 24 valores no formato hexadecimal que são utilizados na etapa *ι*.

Em cada rodada *nr* um desses valores é utilizado (o valor é referente ao número da rodada) para realizar a operação lógica XOR no primeiro bloco da matriz A. Esta operação tem como objetivo "quebrar" a simetria, para evitar brechas que podem ser exploradas em ataques contra o algoritmo [7].

B. Função de Esponja

A função *Keccak[r,c]* utiliza a construção em esponja, que recebe um valor de entrada de tamanho variável e gera uma saída de tamanho arbitrário [7]. A função recebe dois parâmetros, onde: *r* é o parâmetro de *bitrate* e *c* é o parâmetro de capacidade.

O parâmetro *r* define o tamanho de que cada bloco irá ter depois da informação ser quebrada em *P* pedaços de tamanho *r*. É necessário quebrar a informação pois o algoritmo não pode ser aplicado (ou não é desejável por questões de segurança) em uma informação inteira caso ela seja muito grande, mas sim em pequenos pedaços dela.

O parâmetro *c* afeta o desempenho do algoritmo e na segurança do hash gerado, onde quanto maior o valor de *c*, mais seguro o hash gerado, porém exige mais desempenho da máquina.

A soma dos parâmetros *r* e *c* define o número da largura da permutação escolhida, por exemplo, na permutação 1600, um dos valores adotados adotado é $r = 1024$ e $c = 576$, onde $r + c = 1600$. Na competição do NIST, os desenvolvedores do Keccak submeteram quatro propostas do tamanho em bits n do hash gerado e dos parâmetros *r* e *c* [8], que são: (i) $n = 224$: Keccak[$r = 1152$, $c = 448$]; (ii) $n = 256$: Keccak[$r = 1088$, $c = 512$]; (iii) $n = 384$: Keccak[$r = 832$, $c = 768$]; (iv) $n = 512$: Keccak[$r = 576$, $c = 1024$].

A função *Keccak[r,c]* possui as fases de inicialização, *padding*, absorção e compressão, que podem ser vistas no pseudocódigo representado na Fig. 2.

Na fase de inicialização é criada uma matriz de estado *S* de tamanho 5x5 que irá ser preenchida pela informação de entrada. Na fase de *padding* ocorre o preenchimento da informação de entrada *M* com um padrão de *0x01* (valor hexadecimal), seguido de *n 0x00* necessários seguido de *0x80* final, para tornar a informação *M* múltipla do parâmetro *r*.

O preenchimento da informação é necessário, pois o algoritmo disponibiliza a opção da mensagem de entrada ter um tamanho variável que nem sempre dispõe de um tamanho aceitável (múltipla de *r*) para realizar as operações de permutação.

Na fase de absorção (*absorbing*), a informação preenchida que foi armazenada numa variável *P* é quebrada em pedaços *Pi* de tamanho *r* e depois inseridos na matriz de estado *S*. Após a inserção dos valores na matriz são realizados as permutações, ou seja, a função de permutação *Keccak-f[b](S)* é aplicada dos valores de *S*. Esta fase é realizada enquanto existir blocos *Pi* que ainda não foram aplicados pela função de permutação.

Por fim, na fase de compressão (*squeezing*) é definido *hLength*, variável que define o tamanho do hash que será gerado, e então é realizada a concatenação dos blocos de dados já permutados da matriz *S*, que gera uma "*string*" *Z* que é o valor hash (com formato hexadecimal) da informação de entrada que tem tamanho em bits determinado por n. A fase de compressão é realizada enquanto $hLength - r > 0$, que executa novamente a função de permutação sobre a matriz *S* e concatena os valores em *Z* novamente caso a condição seja verdadeira. Isto faz com que a geração do hash seja ainda mais aleatória, prevenindo ataques que geram colisões [7].

V. SMART CARDS

Um dos critérios de seleção do algoritmo SHA-3 vencedor é sua capacidade de execução em plataformas com recursos limitados de processamento e memória e entrada e saída de dados. Neste sentido neste trabalho foi explorado o algoritmo Keccak em plataformas de tipo smart cards.

Smart cards são, basicamente, circuitos integrados incorporados em cartões, geralmente com seu corpo de plástico

e com dimensões de um cartão de crédito [9]. Existem cartões com outras dimensões, como por exemplo, o cartão SIM, ou então cartão GSM-SIM, muito utilizado atualmente em dispositivos de telefonia móvel, para a identificação, controle e armazenamento de dados.

Além dos cartões SIM, outra área que vem expandindo o uso de *smart cards* são as corporações bancárias, que estão distribuindo cartões de crédito com smart cards incorporados, juntamente com a já conhecida tarja magnética.

Comumente é possível encontrar três tipos de *smart cards*: (i).Cartões de memória(têm apenas a capacidade de armazenar, alterar e excluir dados, onde a comunicação com o cartão é feita através de contatos metálicos existentes sobre o corpo do cartão). (ii).Cartões microprocessados (contém um processador incorporado capaz de executar operações e aplicativos. A comunicação é similar aos cartões de memória). (iii). Cartões sem contato (podem ser do primeiro ou segundo tipo, porém possui uma antena incorporada em seu corpo de plástico e sua comunicação é feita através de radio frequência).

É possível encontrar cartões que suportam ambos os tipos de interface, ou seja, podem trabalhar tanto com os contatos metálicos, como através de radio frequência. Estes cartões são denominados como smart cards de interface dupla (dual interface). Independentemente do tipo de smart card escolhido, todos eles compartilham o mesmo destino, que é a de possuir um hardware limitado.

Smart cards com poder de processamento geralmente possuem processadores com barramento de 8 ou 16 bits, e sua memória

EEPROM é capaz de armazenar algumas centenas de Kbytes. Existem também smart cards que possuem memórias ROM e RAM, além da EEPROM.

A comunicação entre o cartão com contato e o host é feita através da troca de blocos de dados (bytes) entre si, onde estes blocos são definidos como Protocolo de Unidade de Dados (APDU), que é o protocolo no nível de aplicação definido pela ISO 7816.

Muitos cartões atualmente vêm com a tecnologia Java Card incorporada, o que facilita na criação e controle de aplicações (bancárias, de segurança, telecomunicações, entre outras) pelos desenvolvedores, pelo fato de que tais aplicações são criadas com a utilização da linguagem Java, sem a necessidade de se utilizar linguagens especificas para a implementação de aplicativos em smart cards.

VI. JAVA CARDS

Java card é a tecnologia que permite que os *smart cards* sejam capazes de executar pequenos aplicativos (applets) criados na linguagem Java (com uma versão mais restrita de recursos de programação). A tecnologia Java Card define um ambiente de execução Java Card (JCRE) e fornece classes e métodos para auxiliar no desenvolvimento de aplicações [10]. A tecnologia consequentemente também disponibiliza a possibilidade de execução destes aplicativos inseridos no *smart card*, através do *Java Card Virtual Machine* (JCVM) contido no mesmo.

Alguns dos benefícios do uso da tecnologia é que a mesma fornece facilidade para o desenvolvimento de aplicações, pois usa uma linguagem de alto nível, traz vários mecanismos de segurança, como por exemplo, o uso de firewalls para separar cada aplicação (impedindo uma acessar outra indevidamente), possui uma independência ao hardware, podendo ser executado um aplicativo em qualquer smart card com suporte a plataforma Java Card, e é possível armazenar e controlar múltiplas aplicações dentro de um mesmo cartão [10].

O JCRE define o ambiente de execução Java Card e tem a responsabilidade de gerenciar e disponibilizar recursos do smart card e também gerencia a execução dos applets, ou seja, ele basicamente é o Sistema Operacional do cartão. O JCRE consiste na máquina virtual Java Card (JCVM), nas Java Card APIs, aplicações específicas do fabricante e das classes do sistema do JCRE.

A JCVM tem a função de executar os applets, controlar a alocação da memória e gerenciar os objetos instanciados, além de prover recursos para executar os applets independentemente do hardware em questão [10].

A linguagem utilizada para o desenvolvimento dos aplicativos é a linguagem Java, porém com restrições de recursos não suportados pelo *smart card*, devido ao fato de que o mesmo dispõe recursos limitados de processamento e memória. Na Tabela I são citados alguns dos recursos suportados e não suportados em cartões que utilizam a plataforma Java Card 2.2 [10].

TABELA I. RECURSOS JAVA SUPORTADOS E NÃO SUPORTADOS

Recursos suportados	**Recursos não suportados**
Tipos boolean, byte e short	Theads
Arrays unidimensionais	Tipos double, float e long
Pacotes Java	Arrays Multidimensionais
Classes,interface e exceptions	Char e Strings

VII. DESENVOLVIMENTO DO KECCAK

Para o desenvolvimento da função na linguagem Java utilizou-se o ambiente de desenvolvimento Eclipse SDK v3.6.1 juntamente com o *plugin* JCOP *Tools*. O JCOP *Tools* é um *plugin* de desenvolvimento de aplicativos para *smart cards*, que foi desenvolvido pela IBM e atualmente é distribuída pela NXP. Este *plugin* disponibiliza um ambiente funcional e amigável para o desenvolvimento e gerenciamento de aplicativos em dispositivos *smart cards*, como por exemplo, a facilidade de importação, remoção e execução de aplicativos nos *smart cards*.

Além disso, ele disponibiliza um simulador de *smart cards*, possibilitando desenvolver e testar aplicativos sem ter fisicamente um cartão conectado a uma leitora. O JRE utilizado e suportado pelo JCOP *Tools* é a versão 1.5.0.16. O *plugin* JCOP Tools fornece o depurador de erros para os programas Java, o compilador de *bytecodes*, o conversor para arquivos CAP e o JCOP *Shell*, utilizado para enviar comandos APDU ao cartão.

A implementação da função de permutação utilizada neste projeto foi desenvolvido por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche, criadores da função Keccak na linguagem C, onde esta função foi convertida para a linguagem Java e adaptada para suportar a tecnologia Java Card bem como suas restrições. A função de esponja foi desenvolvida de acordo com o pseudocódigo da função e baseando-se na função implementada em python.

A função de esponja *Keccak*[r = 144, c = 256] recebe uma informação armazenada num vetor *M* de tamanho variável como entrada e gera um hash de saída de tamanho fixo. O tamanho do hash gerado na implementação é de 224 bits. A função de esponja realiza o "tratamento" da informação de entrada, para que ela seja aplicada pela função de permutação e assim seja gerado o hash. Este tratamento é realizado em três etapas: preenchimento (*padding*), absorção (*absorbing*) e por fim a compressão (*squeezing*).

A função de permutação *Keccak-f* [400] recebe o vetor de estados *S* já preenchida e absorvida pela função de esponja. Esta função tem o objetivo de manipular os blocos de informações através de operações lógicas, rotações de bits e troca de posições para geração do hash de uma informação.

Pelas limitações da tecnologia Java Card, podemos apenas utilizar variáveis do tipo byte e short, que equivalem a 8 e 16 bytes respectivamente. Por este motivo, foi definido utilizarmos a permutação $b = 25 * w$, onde w é o comprimento máximo da palavra, que no caso é 16 e que reflete na permutação 400 e consequentemente define o número de rodadas $nr = 20$. A função possui cinco etapas com propósitos específicos, que são: θ, ρ, π, χ e ι, onde estas etapas serão executas 20 vezes.

VIII. RESULTADOS

Nesta seção apresenta-se a análise da função Keccak e seu comportamento dentro do cartão. Em termos de operações lógicas, a função *Keccak-f* [*b*] utiliza aproximadamente [7]:

- $76n_r$ XORs, $25n_r$ ANDs, $25n_r$ NOTs e $29n_r$ rotações de *b* bits.

Utilizando o *Keccak-f* [400], temos ($n_r = 20$):

- 1520 XORs, 500 ANDs, 500 NOTs, e 580 rotações de 16 bits.

Em relação à utilização de memória, tem-se os dados representados na Tabela II. Estes valores foram retirados com o uso de comandos dentro do cartão, disponibilizados pelo Java Card.

TABELA II. MEMÓRIAS UTILIZADAS PELA FUNÇÃO KECCAK

Tamanho da Função (bytes)	EEPROM Utilizado (bytes)	RAM Utilizada (bytes)
1275	2579	703

A função Keccak ocupa 1275 bytes na memória EEPROM do cartão, onde este valor representa apenas o tamanho da função carregada na memória. Para utilização nas operações da função, são utilizados 2579 bytes da memória EEPROM e 703 bytes da memória RAM.

Em relação ao tempo de execução, a Tabela III descreve uma comparação com as funções de *hash* MD5 e SHA, ambas disponibilizadas pelo cartão através da tecnologia Java Card. Estas funções foram executadas dentro do cartão, passando um valor de entrada padrão.

TABELA III. COMPARAÇÃO DE TEMPO DE EXECUÇÃO ENTRE TRÊS FUNÇÕES DE HASH

Função	Tempo aproximado de execução (s)	Tamanho *hash* (bits)
MD5	0,194	256
SHA	0,196	300
Keccak	37,392	224

Como visto na Tabela III, o tempo aproximado que a função Keccak implementada neste projeto leva para executar é de aproximadamente 37 segundos, um tempo muito elevado em relação às funções MD5 e SHA, que executam em menos de 1/5 de segundo. Este tempo elevado é indesejável, tornando seu uso (no momento atual) impraticável para a maioria das aplicações reais.

É importante salientar que a diferença apresentada entre MD5 e SHA em relação ao Keccak é justificada pelo fato que o Keccak foi implementado no cartão como uma aplicação (applet) em memória EEPROM e os algoritmos MD5 e SHA são dedicados e já presentes neste modelo de smart card podendo fazer uso dos coprocessadores embarcados.

IX. CONCLUSÃO

Durante a implementação da função não se obteve problemas para adaptá-la à tecnologia Java Card, em grande parte por suportar a linguagem Java, o que facilitou em muito a implementação. Importante ressaltar que utilizou-se a implementação da função Keccak compatível à arquitetura do cartão (Keccak[400]), que é diferente da função proposta para a competição (Keccak[1600]), que possui padrões de segurança maiores porém necessita de um hardware potente, o que seria inviável utilizar esta função no cartão.

Foi observado que a função Keccak implementada neste projeto requer muito tempo de processamento para gerar a função de hash, o que torna seu uso ineficaz. A otimização da função deve ser realizada para diminuir este tempo e umas das formas é a utilização da memória RAM efetivamente, que fornece uma velocidade de acesso no mínimo 10x maior comparado a memória EEPROM.

Com o uso parcial da memória RAM foi possível atingir o tempo de processamento descrito na seção IX, que é de aproximados 37 segundos, contra os 60 segundos da implementação anterior, onde a função apenas utilizava a memória EEPROM. Com o uso efetivo da memória RAM e uma melhor otimização do código acredita-se que o tempo de execução diminua consideravelmente e se torne aceitável.

Uma das etapas deste trabalho foi a comparação dos resultados obtidos com trabalhos correlatos. No trabalho de Gouciem [11] foi realizado o estudo informações referentes aos ciclos/bytes (número de ciclos de clock para processar um byte) e *troughtput* (quantidade de dados processados em determinado tempo) para os microcontroladores comumente utilizados em

Pesquisa Financiada pela FAPESP

smart card.

No entanto, a implementação descrita neste projeto executa de fato em smart cards, diferente do trabalho correlato encontrado [11], que não especifica se realmente foi implementado o algoritmo keccak em uma arquitetura de *smart card* ou apenas foi aferido o número de ciclos para o microcontrolador comumente adotado como unidade de processamento e controle em *smart card* específicos.

Isto tem impacto direto nos resultados, uma vez que, não considerados atrasos da arquitetura de dispositivo *smart card* completo, como mecanismos de entrada e saída (comunicação) e hierarquia de memórias. Elementos estes que são considerados gargalos em qualquer arquitetura, o que pode levar a diferentes resultados nos quesitos memoria e tempo de execução.

REFERÊNCIAS

[1] B. Schneier, "Applied Cryptography: Protocols, Algorithms, and Source Code in C". 2nd Edition. New York: John Wiley & Sons, 1996.

[2] A. J. Menezes, P. C. Oorschot, S. A. Vanstone, "Handbook of Applied Cryptography". CRC Press, 1996.

[3] X. Wang, D. Feng, X. Lai, H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD". Crypto'04, 2004.

[4] X. Wang, Y. L. Yin, H. Yu, "Finding Collisions in the Full SHA-1". Crypto'05, 2005.

[5] National Institute of Standards And Technology, "Cryptographic Hash Algorithm Competition". 2008, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[6] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, "The Keccak Reference". Version 3, Janeiro 2011, http://keccak.noekeon.org/keccak-reference-3.0.pdf.

[7] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, "The Keccak SHA-3 Submission" Version 3, Janeiro 2011, http://keccak.noekeon.org/Keccak-submission-3.pdf.

[8] W. Rankl, "Smart Cards Applications: Design Models for using and programming smart cards". Chichester: John Wiley & Sons, 2007.

[9] Z. Chen, "Java Card Technology for Smart Cards: Architecture and Programmer's Guide". Addison-Wesley, 2000.

[10] M. Gouciem, "Comparison of Seven SHA-3 Candidates Software Implementations on Smart Cards". Outubro 2010, http://eprint.iacr.org/2010/531.pdf.