FUNDAÇÃO DE ENSINO "EURÍPEDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPEDES DE MARÍLIA" – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FERNANDO YOKOTA MARQUES

PROJETO, DESENVOLVIMENTO E ANÁLISE DE UM DOS ALGORITMOS SHA-3 FINALISTAS EM SMART CARDS

MARÍLIA
2011

FERNANDO YOKOTA MARQUES

PROJETO, DESENVOLVIMENTO E ANÁLISE DE UM DOS ALGORITMOS SHA-3 FINALISTAS EM SMART CARDS

Trabalho de Conclusão 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:
Prof. Dr. FÁBIO DACÊNCIO PEREIRA

MARÍLIA
2011

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Fernando Yokota Marques

PROJETO, DESENVOLVIMENTO E ANÁLISE DE UM DOS ALGORITMOS SHA-3 FINALISTAS EM SMART CARDS

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: 10 (Dez)

Orientador: Fábio Dacêncio Pereira

1º. Examinador: Rodolfo Barros Chiaramonte

2º. Examinador: Fabio Lucio Meira

Marília, 28 de novembro de 2011.

Dedico este trabalho a minha família, em especial a minha mãe, pelo incentivo e oportunidade ao estudo. Sem ela não teria realizado este trabalho.

AGRADECIMENTOS

A todos os professores do curso de Ciência da Computação que eu tive a oportunidade de conhecer, e aos professores que coordenam o curso que, com empenho e dedicação, sempre tentam torná-lo melhor.

Em especial ao prof. Fábio Dacêncio pela orientação, constante apoio e incentivo não só deste trabalho, mas também de outras oportunidades.

Ao Edson Emilio e à LSITEC que concedeu todos os dispositivos utilizados neste trabalho e que cedeu parte do seu tempo para ensinar a utilizar os mesmos.

YOKOTA, Fernando. **Projeto, Desenvolvimento e Análise de um dos Algoritmos SHA-3 Finalistas em Smart Cards**. 2011. 62 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2011.

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. Porém grande parte das funções atuais já não consegue evitar ataques maliciosos e garantir que a informação tenha apenas um *hash.* Atualmente 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 contexto, neste trabalho foi selecionado um dos algoritmos finalistas da chamada para o SHA-3 e posteriormente este foi implementado em *smart cards*, com o intuito de obter dados de desempenho para futura comparação com trabalhos correlatos e divulgação na comunidade científica.

Palavras-chave: Funções de *Hash,* SHA-3, Java Card, *Smart Cards*.

YOKOTA, Fernando. **Projeto, Desenvolvimento e Análise de um dos Algoritmos SHA-3 Finalistas em Smart Cards**. 2011. 62 f. Trabalho de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino “Eurípides Soares da Rocha”, Marília, 2011.

ABSTRACT

The Integrity of Information 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 functions, which generates a stream of bytes (hash) that must be unique. But most of the current functions can no longer prevent malicious attacks and ensure that the information has only one hash. Currently through a public call the National Institute of Standards and Technology (NIST) called technology centers, business companies and the scientific community to submit proposals for the new hash function standard, called SHA-3. Once the candidates presented, they are exposed and evaluated in several aspects. In this context, in this work was selected a finalist of the algorithms call for SHA-3 and posteriorly implemented in smart cards, in order to obtain performance data for future comparison with related work and divulgation in the scientific community.

Keywords: Hash Functions, SHA-3, Java Card, Smart Cards.

LISTA DE ILUSTRAÇÕES

Figura 1 – Processo de Cifragem de uma mensagem 18
Figura 2 – Criptografia com chave simétrica 21
Figura 3 – Criptografia com chave assimétrica 23
Figura 4 – Função de hash 24
Figura 5 – As operações de cada rodada e suas etapas de *Keccak-f[b]* 30
Figura 6 – Função de Esponja *Keccak[r,c]* 32
Figura 7 – Módulo com contatos do smart card 35
Figura 8 – Estrutura do APDU de comando 36
Figura 9 – Estrutura do APDU de resposta 37
Figura 10 – Arquitetura da tecnologia Java Card 40
Figura 11 – Estrutura padrão do AID 41
Figura 12 – Estrutura de um aplicativo em Java Card 42
Figura 13 – Ciclo de Desenvolvimento de um aplicativo em Java Card 42
Figura 14 – Leitor de cartões com interface USB 44
Figura 15 – Smart Card utilizado no projeto 45
Figura 16 – Arquitetura do hardware smart card NXP P541G072 46
Figura 17 – Etapa de padding 48
Figura 18 – Etapa de absorção 49
Figura 19 – Etapa de compressão 50
Figura 20 – Etapa θ 51
Figura 21 – Etapas ρ e π 51
Figura 22 – Vetor de rotações utilizado na etapa ρ 52
Figura 23 – Etapa χ 52
Figura 24 – Etapa ι 53
Figura 25 – Vetor de constantes da rodada utilizado na etapa ι 53

LISTA DE TABELAS

Tabela 1 – Cifra de Cesar 19
Tabela 2 – *Timeline* provável da Competição do NIST 26
Tabela 3 – *Round Constants* de RC[i] 31
Tabela 4 – Recursos Java suportados e não suportados 40
Tabela 5 – Algoritmos criptográficos geralmente encontrados em *smart cards* 43
Tabela 6 – Memórias utilizadas pela função Keccak 54
Tabela 7 – Comparação de tempo de execução entre três funções de *hash* 55

LISTA DE SIGLAS

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
AID	Application Identifier
APDU	Application Protocol Data Unit
C-APDU	Command Application Protocol Data Unit
DES	Data Encryption Standard
EEPROM	Electrical Erasable Programmable ROM
GSM	Global System for Mobile Communications
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JCDK	Java Card Development Kit
JCOP	Java Card Open Platform
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
MD5	Message-Digest Algorithm 5
NIST	National Institute of Standards and Technology
P2P	Peer-To-Peer
PIX	Proprietary Application Identifier Extension
RAM	Random Access Memory
R-APDU	Response Application Protocol Data Unit
RC4	Rivest Ciphers 4
RID	Registered Application Provider Identifier
ROM	Read Only Memory
SHA	Secure Hash Algorithm
SIM	Subscriber Identity Module

SUMÁRIO

INTRODUÇÃO ... 13
Objetivos ... 14
Problemática e Justificativa ... 14
Metodologia ... 15
Organização do Trabalho ... 16
CAPÍTULO 1 – SEGURANÇA DA INFORMAÇÃO ... 17
1.1 Criptografia ... 18
1.2 História e Conceitos da Criptografia ... 19
1.3 Algoritmos de Criptografia ... 20
1.3.1 Criptografia de Chave Simétrica ... 21
1.3.2 Criptografia de Chave Assimétrica ... 22
1.3.3 Funções de Hash ... 24
CAPÍTULO 2 – COMPETIÇÃO DA NIST ... 26
2.1 Timeline da Competição ... 26
2.2 Situação Atual da Competição ... 28
CAPÍTULO 3 – ALGORITMO KECCAK ... 29
3.1 Função de Permutação ... 29
3.2 Função de Esponja ... 31
CAPÍTULO 4 – SMART CARDS ... 34
4.1 Justificativa da Arquitetura Escolhida ... 34
4.2 Categorias de Smart Cards e Seu Funcionamento ... 34
4.3 Protocolo APDU ... 36
4.3.1 APDU de Comando ... 36
4.3.2 APDU de Resposta ... 37
CAPÍTULO 5 – JAVA CARD ... 39
5.1 Aplicativo em Java Card ... 41
5.2 Algoritmos Criptográficos em Smart Cards ... 43
CAPÍTULO 6 – TECNOLOGIAS EMPREGADAS NO PROJETO ... 44
6.1 Java Card NXP P541G072 ... 44
6.2 IDE Eclipse e JCOP Tools ... 46
CAPÍTULO 7 – DESENVOLVIMENTO DO KECCAK EM JAVA ... 48
7.1 Desenvolvimento da Função de Esponja ... 48
7.1.1 Etapa de Padding ... 48
7.1.2 Etapa de Absorção ... 49
7.1.3 Etapa de Compressão ... 49
7.2 Desenvolvimento da Função de Permutação ... 50
7.2.1 Etapa θ ... 51
7.2.2 Etapas ρ e π ... 51
7.2.3 Etapa χ ... 52
7.2.4 Etapa ι ... 53

CAPÍTULO 8 – ANÁLISE E RESULTADOS DA FUNÇÃO KECCAK 54
CONCLUSÕES 56
REFERÊNCIAS 58
APÊNDICE A – Função Keccak implementada em Java 60

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 da integridade e da confidencialidade das 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. Esta cadeia gerada tem a incumbência de "identificar" a informação 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 dados de desempenho para comparação com trabalhos correlatos e divulgação na comunidade científica.

Objetivos

O projeto teve como objetivo geral o estudo, seleção, implementação e análise do desempenho em *smart card* de um dos algoritmos de *hash* finalistas da terceira etapa da competição do NIST. Para isso foi definido alguns objetivos específicos apresentados na sequência:

a. Aprendizado da plataforma Java Card.
b. Refinamento dos critérios de escolha do algoritmo de *hash* que será implementado.
c. Definição das métricas de medição de desempenho para comparação com trabalhos correlatos.
d. Aprendizado refinado sobre funções de *hash.*
e. Acompanhamento do cenário e comunidade nacional e internacional de pesquisa (acompanhamento de congressos e revistas específicos).

Problemática e Justificativa

A cada fase da chamada pública para escolha da nova função de *hash* são realizados comentários dos avaliadores e muitos artigos são escritos avaliando e comparando os candidatos (NIST, 2008b).

A terceira e última etapa aconteceu em Dezembro de 2010, onde foram escolhidos os cinco finalistas da competição: BLAKE, Grøstl, JH, Keccak, e Skein. Apontar falhas principalmente de segurança e desempenho é o papel da comunidade científica até a escolha e definição do algoritmo vencedor.

Neste contexto, o problema explorado neste trabalho foi à seleção consciente e justificada de um dos algoritmos finalistas. Em seguida foi definido uma plataforma para a implementação e testes do algoritmo selecionado, que no caso foi o *smart card*.

Contribuir com o esforço da comunidade científica para a criação de parâmetros para a análise, comparação e seleção da função de *hash* vencedora é uma das justificativas do projeto proposto. Este trabalho tem também como objetivo mostrar como a função de *hash* escolhida se comporta dentro de uma arquitetura com grandes restrições de recursos e processamento.

Metodologia

A metodologia de pesquisa e desenvolvimento foi concentrada em três etapas principais:

1. Escolha da função de *hash* e estudo das tecnologias envolvidas:
    a. Escolher uma das funções de *hash* da segunda etapa, analisando os comentários realizados pela comunidade científica, de cada uma das funções.
    b. Estudar especificamente o funcionamento das funções de *hash.*
    c. Aprender a utilizar as ferramentas disponibilizadas pelo *Java Card Development Kit.*
    d. Definir as métricas de desempenho para avaliação da função e para a comparação com trabalhos correlatos.

2. Implementação e desenvolvimento do projeto:
    a. Implementar a função de *hash* utilizando a plataforma Java Card, a partir do algoritmo de *hash* escolhido.
    b. Compilar e simular a função com as ferramentas inclusas no *Java Card Development Kit.*
    c. Testar e validar o funcionamento da função.

3. Realizar testes e comparar com trabalhos correlatos:
    a. Realizar testes na função de *hash* e analisar os resultados de desempenho gerados.
    b. Comparar os dados obtidos dos testes de desempenho realizados na função de *hash* com trabalhos correlatos.
    c. Publicar os resultados e as comparações realizadas no formato de artigos científicos.

Organização do Trabalho

No capítulo 2 é descrito sobre a Segurança da Informação e qual a sua importância dentro da área computacional. É abordado de maneira sucinta a criptografia e sua história, propriedades e seus métodos mais utilizados. Em especifico, é descrito uma de suas propriedades: a integridade da informação, e uma das formas de realizá-la, que é através de funções de *hash*, tema principal neste projeto.

No capítulo 3 é abordado sobre a competição do NIST para a escolha da função de *hash* SHA-3, o motivo da sua realização e como se encontra a competição atualmente.

O capítulo 4 descreve o algoritmo Keccak, um dos finalistas da competição da NIST que foi escolhido neste projeto para ser implementado e analisado. É visto suas características e funcionamento, e o motivo em que levou a escolha deste algoritmo.

No capítulo 5 é descrito o dispositivo empregado no projeto, que é o *smart card,* e é apresentado suas características, formatos e modos de comunicação.

O capítulo 6 descreve sobre a tecnologia Java Card, sua utilidade, modos de funcionamento, peculiaridades da tecnologia e alguns benefícios de se utilizá-la.

O capítulo 7 aborda as tecnologias empregadas no projeto, como o tipo de cartão empregado e suas características, o leitor utilizado para realizar a comunicação entre cartão e terminal e a interface de desenvolvimento utilizada.

No capítulo 8 são descritos as etapas de desenvolvimento do algoritmo Keccak na linguagem Java. É detalhada especificamente cada etapa que compõe o Keccak.

O capitulo 9 apresenta a análise feita sobre a função Keccak implantada dentro do cartão, e descreve informações referentes ao desempenho da função.

Por fim, no capítulo 10 é feita a conclusão do projeto, abordando questões importantes observadas no decorrer da implementação e na execução da função dentro do *smart card*.

CAPÍTULO 1 – SEGURANÇA DA INFORMAÇÃO

Segurança da Informação é compreendida como uma série de medidas que têm como objetivo garantir que as informações de qualquer origem (digital, papéis, documentos, etc.) sejam protegidas de acessos por pessoas não autorizadas, que estejam sempre disponíveis e que sejam confiáveis.

No passado a segurança da informação era, de maneira geral, simples de assegurar, bastando apenas armazenar uma informação de importância em um meio físico que disponibilizasse um dispositivo de segurança, como por exemplo, um cofre (BURNETT; PAINE, 2002).

Atualmente já não é tão simples garantir a segurança de uma informação, devido a sua transformação gerada pelo surgimento dos computadores, o que elevou a informação ao nível digital e o tornou mais complexo, dificultando os mecanismos para garantir a segurança.

Segundo Filho (2004, p.1):

> "É oportuno salientar que, nos dias atuais, a informação constitui uma mercadoria, ou até mesmo uma *commodity*, de suma importância para as organizações dos diversos segmentos. Por esta razão, segurança da informação tem sido uma questão de elevada prioridade nas organizações".

Com o surgimento da Internet e serviços que a utilizam, como serviços de entretenimento, comunicação, negócios, entre outros, a informação tornou-se mais acessível para todos, o que aumenta ainda mais a dificuldade de garantir sua segurança, como por exemplo, o acesso à informação restrito apenas a pessoas autorizadas.

Especificamente, a Segurança da Informação tem como objetivo assegurar alguns componentes básicos de uma informação, que são:

- Confidencialidade – componente em que é tratado sobre a ocultação da informação para indivíduos ou qualquer outra entidade que não sejam autorizadas a vê-la. "A necessidade de manter o segredo da informação decorre da utilização de computadores em áreas sensíveis, como o governo e a indústria" (BISHOP, 2003, p. 4) (tradução nossa).
- Integridade – componente que, como o próprio nome sugere, garante que a informação disponibilizada é confiável e original, ou seja, que não foi manipulada de maneira imprópria ou não autorizada (BISHOP, 2003).

- Disponibilidade – componente que garante que o acesso à informação seja confiável e que sempre garanta a disponibilidade da informação quando seu uso se faz necessário.

Uma das áreas que se preocupa basicamente com a segurança da informação é a criptografia, que realiza um estudo profundo e contínuo sobre o assunto e fornece uma série de métodos e técnicas para assegurar a informação e seus componentes.

1.1 Criptografia

A criptografia (do grego *kryptós*, "oculto", e *gráphein*, "escrita") é a ciência que estuda um conjunto de métodos e técnicas que transformam uma informação legível em algo sem sentido aparente. Esta informação se torna secreta e apenas disponível ao seu remetente e ao destinatário, sendo possível recuperar a informação a partir dos dados sem sentido (BURNETT; PAINE, 2002).

O processo de converter o dado ou mensagem original (também chamado de texto claro) em um texto ilegível (ou texto cifrado) é denominado de cifragem, já o processo inverso é denominado de decifragem. A Figura 1 descreve o processo de cifragem e decifragem de uma mensagem, que é possível através da utilização de uma função criptográfica.

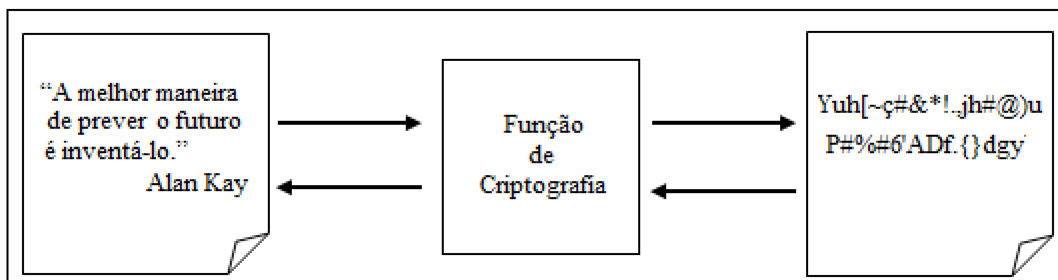

Figura 1 – Processo de Cifragem de uma mensagem

O processo de cifragem de uma informação pode ser realizado através do uso de métodos de substituição ou transposição. No método de cifragem por substituição as letras ou um grupo de letras são trocadas por outras letras, símbolos ou uma combinação destes de acordo com um sistema predefinido. No método de cifragem por transposição as letras que

compõem a informação são postas em ordem diferente, ou seja, permutadas, fazendo com que as mesmas percam o sentido (MORENO; PEREIRA; CHIARAMONTE, 2005).

A criptografia é derivada da Criptologia, ciência que engloba o estudo das técnicas e métodos utilizados na criptografia para cifragem e decifragem de informações. Dentro da criptologia também é visto o estudo da criptoanálise, ciência que estuda técnicas para quebra de uma cifragem, ou seja, conseguir decifrar uma informação cifrada e obter sua informação legível.

1.2 História e Conceitos da Criptografia

A criptologia só se tornou uma ciência nas ultimas décadas, onde anteriormente era considerada como uma arte muito antiga e que caminhou junto com o homem e sua necessidade de se esconder uma informação. De acordo com Moreno, Pereira e Chiaramonte (2005, p. 22):

> “A criptografia é tão antiga quanto a própria escrita, visto que já estava presente no sistema de escrita hieroglífica dos egípcios [1900 a.C.]. Os romanos utilizavam códigos secretos para comunicar planos de batalha. Com as guerras mundiais e a invenção do computador, a criptografia cresceu incorporando complexos algoritmos matemáticos”.

Uma das famosas técnicas de criptografia usadas para cifragem por substituição é a Cifra de Cesar, utilizada pelo imperador romano Júlio Cesar em 50 a.C. para aumentar a segurança de mensagens governamentais (MORENO; PEREIRA; CHIARAMONTE, 2005). A técnica consistia em substituir cada letra da mensagem, desviando-a em três posições - A se tornava D, B se tornava E, e assim sucessivamente, como demonstrado na Tabela 1.

Tabela 1 – Cifra de Cesar

Normal	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Cifrado	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Apesar de antiga, a cifra de césar é um dos poucos, senão o único, algoritmo da antiguidade que ainda é utilizado em aplicações modernas, frequentemente incorporado como parte de esquemas mais complexos.

Ao decorrer dos anos vários estudos foram realizados, métodos criptográficos foram criados e cifras quebradas. A criptografia foi amplamente utilizada nas guerras para esconder informações dos inimigos, onde eram criadas máquinas para o uso específico de cifragem de

informações e comunicações. Obviamente também foi muito utilizado a criptoanálise para descobrir informações comprometedoras e decisivas, ou seja, a criptologia em si foi e ainda é de extrema importância para o governo e nações.

Com o desenvolvimento tecnológico, em específico o surgimento dos computadores, as informações tomaram outro formato, o digital, porém ainda partilhavam da mesma necessidade de uma informação convencional escrita num papel, que era de mantê-la privada. Influenciada com esta nova era digital, a criptografia também mudou, distanciando-se de seus conceitos tradicionais, ampliando seus horizontes e adaptando-se a esta nova era.

Com isso a criptologia além de estudar novas técnicas criptográficas para manter a confidencialidade da informação, ampliou sua gama de aplicações e passou a atender outros componentes da informação, que são:

- Integridade – Garante se uma informação foi manipulada ou não por alguém não autorizado, garantindo assim sua confiabilidade.
- Autenticidade – Garante que a informação transmitida é realmente originária do emissor, garantindo a segurança e identificação da origem da informação.
- Irretratabilidade – Garante que o autor da informação não consiga negar sua autoria, garantindo provas inegáveis da autoria da informação.

Para garantir os componentes da informação descritos acima, utiliza-se uma série de métodos e técnicas, geralmente específicos para cada componente da informação.

1.3 Algoritmos de Criptografia

Como citado anteriormente, geralmente são utilizados métodos ou funções criptográficas especificas para garantir cada um dos componentes da informação. A cifra de César, por exemplo, garante somente a confidencialidade da mensagem, porém na criptografia pode-se adicionar meios que além de ocultarem uma informação também garantem a autenticidade do emissor (BUNETT; PAINE, 2002).

Existem também funções que garantem somente a integridade da informação, ou junto com ela também garantem a autenticidade e irretratabilidade do emissor.

1.3.1 Criptografia de Chave Simétrica

Na criptografia de chave simétrica (também conhecida como criptografia de chave única), um algoritmo de criptografia utiliza uma chave para cifrar a informação. Para torná-la legível novamente utiliza-se a mesma chave para recuperar as informações. Ou seja, é utilizada uma única chave tanto para a operação de cifragem quando para a decifragem da informação. A Fig. 2 ilustra um algoritmo de criptografia que utiliza a chave para converter um texto simples em um texto cifrado e o inverso.

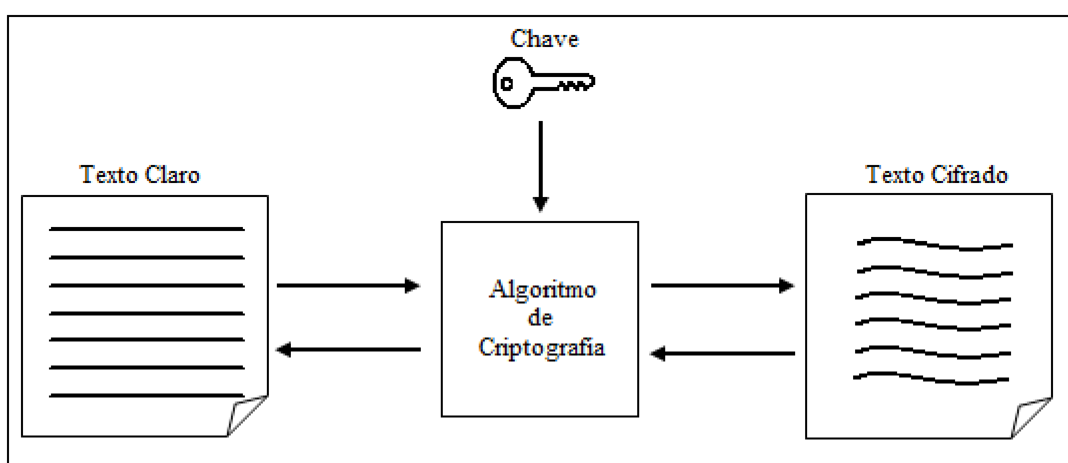

Figura 2 – Criptografia com chave simétrica

Existem dois tipos básicos de algoritmos de chave simétrica: a cifra de blocos e a de fluxo (SCHNEIER, 1996). Ambos relacionam o modo de como a informação será tratada quando forem processadas.

A cifra de bloco opera sobre blocos de dados. A informação é dividida em blocos de dados de tamanho fixo, geralmente possuindo 64 ou 128 bits (quando não se completa o tamanho estipulado, o bloco é preenchido com dados, geralmente de valor 0), e assim se aplica a função de criptografia em cada um dos blocos.

A cifra de fluxo opera com o fluxo da informação, criptografando-a bit a bit, ou algumas vezes em bytes, em um fluxo continuo até o seu término.

Dentre os algoritmos de criptográfica simétrica que foram e ainda são utilizados, podemos citar alguns conhecidos, como os algoritmos RC4 (*Rivest Ciphers* 4), DES (*Data Encryption Standard*) e o AES (*Advanced Encryption Standard*).

O DES é um algoritmo de criptografia em blocos que utiliza uma chave de 56 bits para criar uma tabela de chaves, onde esta tabela é utilizada tanto para cifrar uma informação

quanto para decifrá-la (BURNETT; PAINE, 2002). O DES foi desenvolvido na década de 70 por pesquisadores da IBM (em especial Horst Feistel) em conjunto com *National Security Agency* (NSA), agência responsável pela segurança de informações dos Estados Unidos.

O AES advém da competição realizada pelo NIST para procura de um novo padrão de algoritmo criptográfico, para suceder o DES. Em 2000, o NIST anunciou o algoritmo Rijandael (desenvolvido por Vincent Rijmen e Joan Daemen) como vencedor da competição (BURNETT; PAINE, 2002). Diferente do DES, que foi motivo de críticas pelo tamanho de sua chave que possuía apenas 56 bits, o AES utilizava chaves de 128, 192 ou 256 bits.

O RC4, diferentemente dos algoritmos DES e AES, utiliza a cifragem de fluxo. Foi desenvolvido em 1987 pela RSA *Data Security* e nunca foi publicado seu funcionamento oficialmente, por interesses financeiros da empresa (BURNETT; PAINE, 2002). Porém em 1994 *hackers* conseguiram descobrir seu algoritmo, que foi disponibilizado na Internet. Atualmente o RC4 é utilizado em protocolos como o *Secure Socket Layer* (SSL) e no *Wired Equivalent Privacy* (WEP).

O uso destes tipos de funções criptográficas de chave simétrica pode gerar um problema chamado de “problema de distribuição de chaves” (MORENO; PEREIRA; CHIARAMONTE, 2005), pelo fato de que todas as partes relacionadas àquela informação, ou seja, seu emissor e seu(s) receptor(es) devem conhecer a chave a para acessar a informação.

Com isso, a ação de transmissão da chave para o receptor deve ser feita de maneira segura, pois caso alguém intercepte a transmissão da mensagem cifrada e consiga obtê-la, o mesmo também consegue interceptar a transmissão da chave, o que torna a cifragem da mensagem em vão.

1.3.2 Criptografia de Chave Assimétrica

A criptografia de chave assimétrica (também conhecido como criptografia de chave pública) é um dos meios que contornam o problema de distribuição de chaves que advém do uso de algoritmos de criptografia de chave simétrica. Para isso, neste esquema são utilizadas duas chaves diferentes.

Na criptografia simétrica uma chave serve tanto para cifrar quanto para decifrar, já na criptografia de chave pública uma chave é utilizado para cifrar uma informação, e a outra para decifrá-lo. A chave para criptografar informações é a chave pública, que todos os envolvidos

têm conhecimento, já a chave privada é mantida em segredo e é utilizada para decifrar a informação.

Para melhor compreensão, na Fig. 3 é ilustrado o processo realizado pelo algoritmo. Para transmitir uma mensagem, o emissor cifra a mensagem com a chave pública do receptor em questão, e o receptor ao receber a mensagem só pode decifrá-la utilizando sua chave privada.

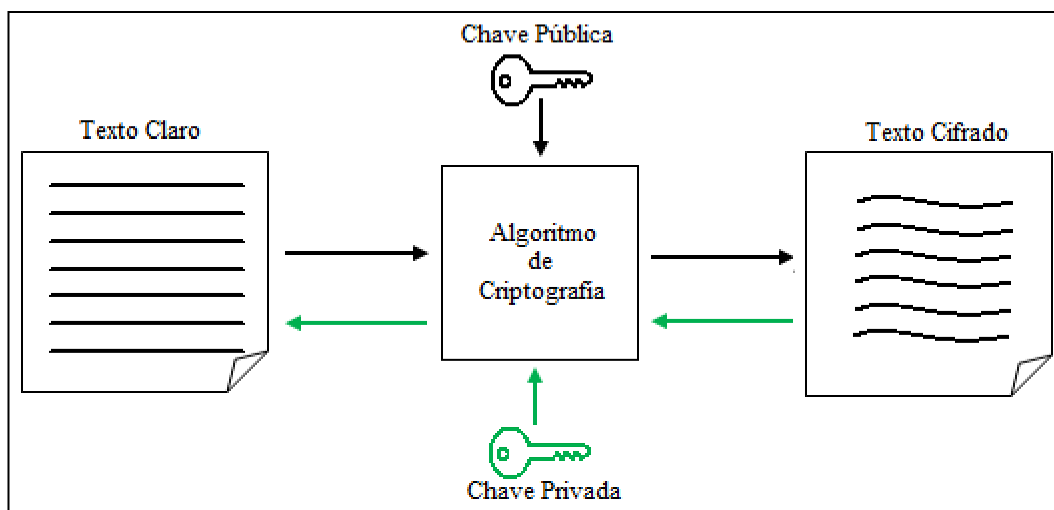

Figura 3 – Criptografia com chave assimétrica

Podemos citar algoritmos conhecidos que utilizam este esquema, que são o Diffie-Hellman e o RSA. O conceito de criptografia de chave publica foi introduzido na década de 1970, por Whitfield Diffie e seu professor Martin Hellman. Eles resolveram o problema da distribuição de chaves em particular, usando o esquema de duas chaves: uma pública, que era distribuída para todos, e uma privada, na qual se mantinha secreta (SCHNEIER, 1996). Este esquema ou método é chamado de Diffie-Hellman, e ainda é utilizado hoje em dia.

Ronald Rivest em conjunto de seus colegas Adir Shamir e Len Adleman inventaram o algoritmo RSA, que foi publicado em 1978 (BURNETT; PAINE, 2002). O RSA foi criado seguindo o mesmo principio do método de Diffie-Helllman, utilizando duas chaves diferentes para a operação de cifragem e decifragem.

Porém tanto as funções de chave simétrica quanto as funções de chave assimétrica têm a função principal de garantir a confidencialidade da informação. Para garantir a integridade, são utilizadas funções de *hash*.

1.3.3 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 especifica 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 (diferente das funções de criptografia, ou seja, eles não cifram informações) que a partir de uma informação (chamado de pré-imagem) 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* (SCHNEIER, 1996).

Funções de *hash* são operações de mão única, que significa que 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 pequeno digesto da mensagem, que não é relativo ao tamanho da informação original. Na Fig. 4 é possível ver um *hash* gerado a partir da de uma *string* vazia como entrada para a função.

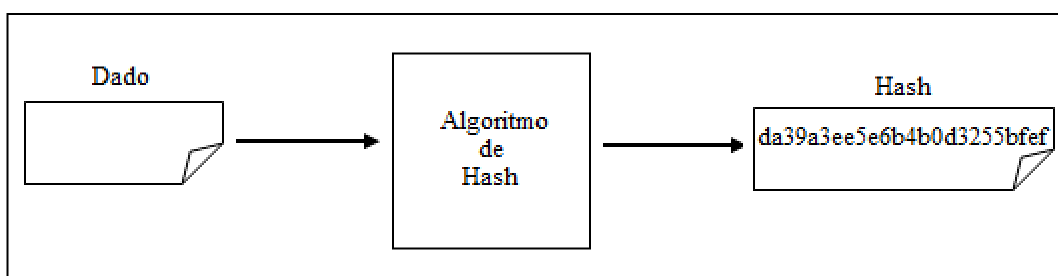

Figura 4 – Função de hash

Uma função de *hash* H segue algumas características ou propriedades distintas que a diferem de outras funções similares (SCHNEIER, 1996), 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 (MENEZES; OORSCHOT; VANSTONE, 1996) 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 *hashs* de tamanhos diferentes que são o SHA-224, SHA-256, SHA-384 e SHA-512, e são utilizados atualmente em aplicações que exijam alta segurança da integridade.

No entanto foram reportados sucessos de ataques tanto para o algoritmo MD5 (WANG et al., 2004), assim como para os algoritmos SHA-0 e SHA-1 (WANG; YIN; YU, 2005), 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.

Com esta preocupação, o NIST criou em novembro de 2008 a Competição de Algoritmos *Hash* Criptográficos, que irá selecionar um algoritmo dentre os vários inscritos na competição para suceder os algoritmos da família SHA.

CAPÍTULO 2 – COMPETIÇÃO DA NIST

Foram reportadas falhas encontradas nas funções SHA-1 e MD5 pela comunidade científica, que geram ataques de colisão com sucesso. 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, onde os especialistas da área escolheram 5 dos 14 algoritmos presentes na segunda etapa, dos quais haviam 64 inscritos no início da competição. O *timeline* da competição, que contêm informações mais específicas de seu processo se encontra na seção 3.1.

2.1 Timeline da Competição

A competição surgiu, de acordo com o NIST (2008a) (tradução nossa),

> "em resposta a uma vulnerabilidade no SHA-1 anunciado em fevereiro de 2005, o NIST realizou um workshop de *hash* criptográfico em 31 de outubro de 2005 para avaliar o estado das suas funções *hash* aprovadas. Enquanto o NIST continuava a recomendar a transição da função SHA-1 para a família de funções aprovadas SHA-2 (SHA-224, SHA-256, SHA-384 e SHA-512), o NIST decidiu também que seria prudente desenvolver a longo prazo uma ou mais funções de *hash* através de um concurso público, semelhante ao processo de desenvolvimento para o *Advanced Encryption Standard* (AES)".

Na Tabela 2 está descrito o *timeline* contendo os períodos da competição proposta pelo NIST para o desenvolvimento das novas funções de *hash*. Como pode ocorrer imprevistos na competição, este *timeline* é apenas uma tentativa proposta, podendo ser alterada pelo NIST quando houver necessidade (NIST, 2008a).

Tabela 2 – *Timeline* provável da Competição do NIST

2007	
Jan – Mar.	Publicação dos requerimentos mínimos preliminares, requerimentos da submissão e critério de avaliação para os comentários públicos.
27/04/07	Período para comentários público terminado.
Abr – Jun	Análise dos comentários.

Out – Dez	Finalização e publicação dos requerimentos mínimos para aceitação, requerimentos de submissão e o critério de avaliação das funções de *hash* candidatas. Requisição para submissão das funções de *hash.*
2008	
Out – Dez	Prazo final para submissão das funções de *hash.*
2009	
Abr – Jun	Revisão das funções submetidas, e seleção dos candidatos que atende os requerimentos básicos da submissão. Primeira conferência para anúncio dos candidatos da primeira etapa da competição. Chamada para comentários públicos dos candidatos da primeira etapa.
2010	
Abr – Jun	Término do período para comentários públicos.
Abr – Jun	Segunda conferência das funções de *hash* candidatas. Discussão dos resultados das análises dos candidatos. Candidatos podem identificar possíveis melhorias para seus algoritmos.
Jul – Set	Análise dos comentários públicos sobre os candidatos e seleção dos finalistas. Preparação do relatório para explicar a seleção. Anúncio dos finalistas e publicação do relatório de seleção.
Out – Dez	Candidatos finalistas podem anunciar qualquer ajuste de seus algoritmos. Última etapa começa.
2011	
Out – Dez	Período para comentários público da etapa final terminado.
2012	
Jan – Mar	Terceira e final conferência das funções de *hash* candidatas. Candidatos finalistas discutem sobre os comentários de suas submissões.
Abr – Jun	Análise dos comentários públicos e seleção do vencedor. Preparação do relatório para descrever a seleção final. Anúncio da nova função de *hash*.
Jul – Set	Projeto revisado do padrão de *hash*. Publicação do padrão de projeto para análise e comentários públicos.
Out – Dec	Período para comentários públicos terminado. Análise destes comentários. Envio do padrão proposto para a *Secretary of Commerce* para assinatura.

2.2 Situação Atual da Competição

Atualmente a competição para a escolha da nova função de *hash* realizado pelo NIST se encontra em sua terceira e ultima etapa. A terceira etapa da competição começou no final de 2010, onde cinco algoritmos finalistas foram escolhidos para esta última etapa, dos 14 algoritmos da segunda etapa. É previsto que o algoritmo finalista seja anunciado e publicado pelo NIST em 2012 (NIST, 2008b).

As funções escolhidas para esta terceira etapa são: BLAKE, Grøstl, JH, Keccak e Skein. Neste projeto foi escolhida a função de *hash* denominada Keccak, desenvolvida por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche.

CAPÍTULO 3 – ALGORITMO KECCAK

Neste projeto foi escolhido o algoritmo Keccak dentre os cinco finalistas, pois apresenta uma estrutura relativamente simples e versátil, o que facilita não só na implementação do mesmo, mas também na compatibilidade da arquitetura do *smart card* que é utilizado no projeto. *Smart cards* disponibilizam pouco poder de processamento e recursos como memória, devido ao seu corpo com dimensões de alguns centímetros, por isso é necessário que o algoritmo seja simples.

Além disso, não é foco deste trabalho acertar qual será o vencedor da competição, mas sim criar mais um parâmetro para comparação em um dispositivo de baixo recurso que é o *smart card*.

O algoritmo Keccak pertence à família de funções de esponja, que são funções de *hash* que utilizam a construção em esponja (descrita na seção 4.2) para transformação ou permutação de tamanho fixo para construção de uma função que a partir de uma entrada de qualquer tamanho gere um saída de tamanho arbitrário (DAEMEN *el al.*, 2009). 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 segurança comprovada contra todos os ataques genéricos existentes (DAEMEN *et al.*, 2009).

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.

3.1 Função de Permutação

O algoritmo 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.

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 terá 64 bits.

Esta função realiza um número de rodadas n_r 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 n_r depende da largura de permutação, que é dada por $n_r = 12 + 2*l$, onde $2^l = w$. Seguindo o exemplo acima, se *w* for igual a 64, o número de rodadas seria igual a 24, pois $2^l = 2^6$, tal que $n_r = 12 + 2 * 6$, ou seja, em cada uma das 24 rodadas seriam realizadas as cinco etapas de operações presentes na função de permutação.

As cinco etapas que manipularão 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. 5 apresenta o pseudocódigo da função de permutação, onde a matriz *A* contendo blocos de informação e o *RC* (*round constant*) 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…n_r-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)

  χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),  forall (x,y) in (0…4,0…4)

  ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

Figura 5 – As operações de cada rodada e suas etapas de *Keccak-f[b]*

O parâmetro *RC* (definido na Tabela 2) presente no pseudocódigo da função de permutação é um vetor contendo 24 valores no formato hexadecimal que são utilizados na etapa ι.

Tabela 3 – *Round Constants* de RC[i]

RC[0]	0x0000000000000001	**RC[12]**	0x000000008000808B
RC[1]	0x0000000000008082	**RC[13]**	0x800000000000008B
RC[2]	0x800000000000808A	**RC[14]**	0x8000000000008089
RC[3]	0x8000000080008000	**RC[15]**	0x8000000000008003
RC[4]	0x000000000000808B	**RC[16]**	0x8000000000008002
RC[5]	0x0000000080000001	**RC[17]**	0x8000000000000080
RC[6]	0x8000000080008081	**RC[18]**	0x000000000000800A
RC[7]	0x8000000000008009	**RC[19]**	0x800000008000000A
RC[8]	0x000000000000008A	**RC[20]**	0x8000000080008081
RC[9]	0x0000000000000088	**RC[21]**	0x8000000000008080
RC[10]	0x0000000080008009	**RC[22]**	0x0000000080000001
RC[11]	0x000000008000000A	**RC[23]**	0x8000000080008008

Em cada rodada n_r 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 (DAEMEN *et al,* 2011b).

3.2 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 (DAEMEN *et al.,* 2011b). A construção em esponja é uma construção iterada utilizada para a construção de uma função *F* com entrada de tamanho variável e saída de tamanho arbritário baseado em uma permutação *f* que opera em um número fixo de bits *b,* onde *b* é a largura da permutação e representa a soma dos parâmetros *r* e *c* (DAEMEN *et al.*, 2009).

A função de esponja *Keccak[r,c]* 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 terá 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 no desempenho do algoritmo e na segurança do *hash* gerado, onde quanto maior o valor de *c*, mais seguro o *hash* gerado, porém exige maior 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 (DAEMEN *et. al*, 2011c), que são:

- *n* = 224: Keccak[r = 1152, c = 448]
- *n* = 256: Keccak[r = 1088, c = 512]
- *n* = 384: Keccak[r = 832, c = 768]
- *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. 6.

```
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 6 – Função de Esponja *Keccak[r,c]*

Na fase de inicialização é criada uma matriz de estado S de tamanho 5x5 que será 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 (DAEMEN *et al,* 2011b).

CAPÍTULO 4 – 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. (RANKL, 2007). Existem cartões com outras dimensões, como por exemplo, o cartão SIM, ou então GSM-SIM, muito utilizado atualmente em dispositivos móveis de telecomunicação (celulares, *smartphones,* etc.) para a identificação, controle e armazenamento dos dados do cliente.

O *smart card* apesar de ter recursos restritos vêm se tornando muito utilizado em diversas áreas como transporte, médicas e principalmente em corporações bancárias e telecomunicações. Áreas como as corporações bancárias requerem grande segurança e sigilo de informações, por isso um *smart card* deve possuir métodos para garantir a segurança das informações contidas dentro do cartão.

4.1 Justificativa da Arquitetura Escolhida

Além da segurança, outro requisito de relevância que é considerado na competição é do desempenho das funções em diferentes arquiteturas, em hardware e software. As funções são testadas tanto em arquiteturas que provêm de alto desempenho computacional quanto em arquiteturas com poucos recursos. Um dos dispositivos utilizados atualmente que fornecem baixos recursos são os *smart cards*.

Neste contexto o projeto teve como objetivo a implementação da função de *hash* em um dispositivo *smart card* com suporte a tecnologia Java Card, para explorar esta arquitetura de baixo desempenho que vem sido muito utilizada e requer segurança dos dados contidos nele.

4.2 Categorias de Smart Cards e Seu Funcionamento

Comumente é possível encontrar três tipos de smart cards:

- Cartões de memória – têm apenas a capacidade de armazenar dados. A comunicação com o cartão é feita através de contatos metálicos existentes sobre o corpo do cartão, descritos na Fig. 7.

- Cartões microprocessados – contém um processador incorporado capaz de executar operações e aplicativos. A comunicação é similar aos cartões de memória.
- Cartões sem contato – pode ser do primeiro ou segundo tipo descrito, porém possui uma antena incorporada em seu corpo de plástico e sua comunicação é feita através de radio frequência, tanto para a transferência de arquivos como para transferência de energia.

Na Fig. 7 (CARDOSO, 2008) podemos ver o módulo contendo os contatos utilizados para a transferência de dados e energia para o cartão. Geralmente um módulo contém oito contatos, sendo utilizados efetivamente apenas cinco deles.

Figura 7 – Módulo com contatos do smart card

Abaixo são descritos a definição dos contatos enumerados na Fig. 7.

1. VCC – Contato para alimentação elétrica
2. RST – Reset
3. CLK – Clock
4. AUX1 – Auxiliar 1 – não utilizado
5. GND – Ground
6. STP – Uso padrão ou proprietário
7. I/O - Contato para entrada e saída
8. AUX2 - Auxiliar 2 – Não utilizado

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 (*Electrically-Erasable Programmable Read-Only Memory*) é capaz de armazenar algumas centenas de Kbytes. Existem também smart cards que possuem memórias ROM (*Read Only Memory*) e RAM (*Random Access Memory*), 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, em inglês), que é o protocolo no nível de aplicação definido pela ISO 7816.

4.3 Protocolo APDU

O *Application Protocol Data Unit* (APDU) é o protocolo utilizado com o propósito de realizar a comunicação entre o *host* e o cartão. A comunicação consiste na transmissão de pacotes de dados de comandos e respostas. Os dados que são enviados do *host* (através de um leitor) para o cartão são denominados *Command*-APDU (C-APDU) e os dados de resposta são chamados de Response-APDU (R-APDU). Estes dois formatos de APDUs possuem uma estrutura definida pelo ISO/IEC 7816-4, e para cada C-APDU existe um R-APDU correspondente. Os dois APDUs apresentam estruturas diferentes, porém a forma de representação dos dados (que é a utilização de valores hexadecimais) é igual.

4.3.1 APDU de Comando

A estrutura de um C-APDU é descrito na Fig. 8. A estrutura é composta por sete campos, onde estes são divididos em corpo (*body*) e cabeçalho (*header*), onde os campos do corpo não são de uso obrigatório.

Header (Obrigatório)				Body (Opcional)		
CLA	INS	P1	P2	Lc	DATA	Le

Figura 8 – Estrutura do APDU de comando

No cabeçalho são definidos quatro campos, que são: *CLA*, *INS*, *P1* e *P2*. Estes campos têm seu preenchimento obrigatório para que a comunicação entre leitor e cartão seja realizada.

O campo *CLA* é chamado de classe de instruções e tem uma tamanho de 1 byte. Este campo é utilizado para identificar uma aplicação e sua classe de instruções. O campo *INS* é o campo de instrução e possui um tamanho de 1 byte. Como o nome sugere, ele é utilizado para identificar uma instrução específica de uma classe. Os campos de parâmetro *P1* e *P2*, ambos com tamanho de 1 byte cada, são utilizados para prover informações sobre o comando enviado.

No corpo da estrutura do C-APDU, os três campos *Lc*, *Data* e *Le* são de uso opcional, apenas quando há a necessidade de transmitir algum dado que será utilizado pela aplicação. O campo *Lc* tem o tamanho de 1 byte e seu valor define o tamanho em bytes dos dados do campo *Data* que serão enviados pelo leitor. O campo *Data* carrega os dados que serão transmitidos para o cartão e pode variar entre 0 e 255 bytes. Por fim, o campo *Le* controla o tamanho esperado pelo *host* da resposta do R-APDU, caso não seja definido um valor neste campo, o tamanho do R-APDU pode ter um valor variável.

4.3.2 APDU de Resposta

A estrutura de um R-APDU é descrito na Fig. 9. A estrutura é composta por três campos, onde estes são divididos em corpo (*body*) e trailer, onde o campo do corpo não é de uso obrigatório.

Body (Opcional)	Trailer (Obrigatório)	
DATA	SW1	SW2

Figura 9 – Estrutura do APDU de resposta

No corpo da estrutura do R-APDU o campo *Data* é de uso opcional, caso a aplicação necessite retornar algum dado para o *host*. Este campo tem um tamanho referente ao valor definido no campo *Le* do C-APDU (caso o mesmo tenha sido definido).

No trailer os campos *SW1* e *SW2* de tamanho de 1 byte cada são parâmetros que retornam estados dos processamentos dos comandos da aplicação. Por exemplo, caso o cartão retorne nos campos *SW1* e *SW2* os valores *90* e *00* (valores hexadecimais) respectivamente, significa que a execução do comando C-APDU foi completado com sucesso.

A comunicação em *smart cards* que suportam a tecnologia Java Card também é feita através do uso do protocolo APDU, por isso há a necessidade de conhecer este protocolo que será utilizado posteriormente para a execução dos *applets* implementados em Java e implantados dentro do cartão.

CAPÍTULO 5 – JAVA CARD

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 específicas ou instruções de baixo nível para a implementação de aplicativos em smart cards.

Java card é a tecnologia que permite que os *smart cards* sejam capazes de executar pequenos aplicativos 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, em inglês) e fornece classes e métodos para auxiliar no desenvolvimento de aplicações. (CHEN, 2000). 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 ela 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. (CHEN, 2000).

Na Fig. 10 (CHEN, 2000) podemos ver a arquitetura da tecnologia Java Card, nos quais estão presentes o JCRE e a JCVM, que são as duas estruturas principais que possibilitam o suporte e execução de aplicativos em Java nos *smart cards*.

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 (CHEN, 2000).

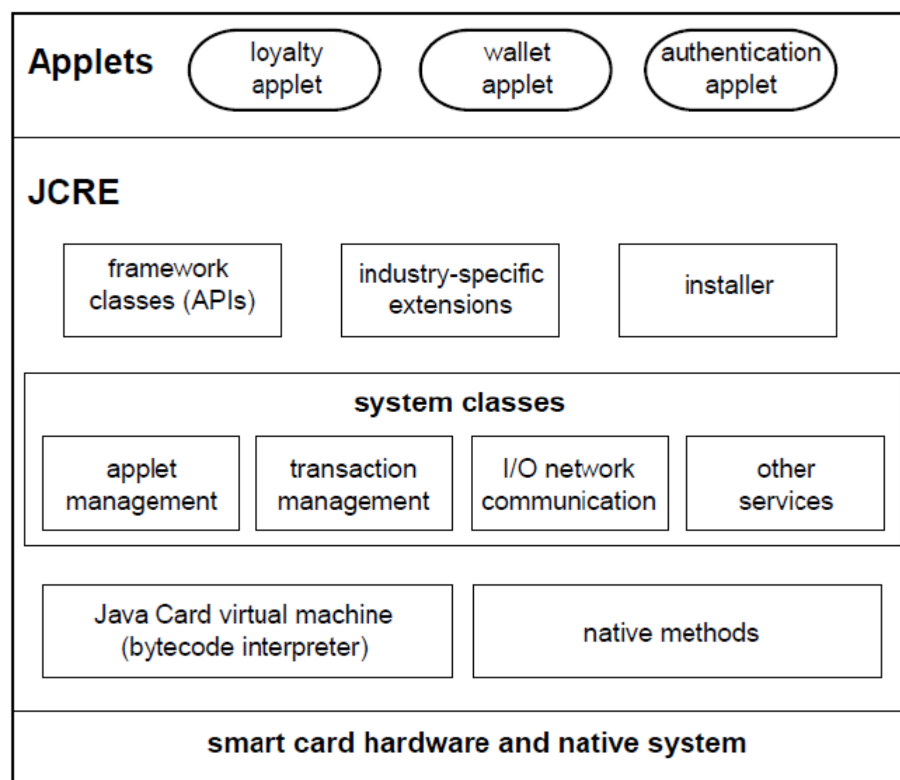

Figura 10 – Arquitetura da tecnologia Java Card

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 4 são citados alguns dos recursos suportados e não suportados em cartões que utilizam a plataforma Java Card 2.2 (CHEN, 2000).

Tabela 4 – Recursos Java suportados e não suportados

Recursos Java Suportados	**Recursos Java Não Suportados**
Tipos primitivos boolean, byte e short	Threads
Arrays unidimensionais	Carga dinâmica de classes
O tipo integer (opcional)	Coletor de Lixo e Finalization
Pacotes Java	Tipos primitivos double, float e long
Classes, interface e exceptions	Char e Strings
Recursos Orientado à Objeto	Arrays Multidimensionais

5.1 Aplicativo em Java Card

Como já dito anteriormente, um aplicativo ou *applet* (programa java) para *smart cards* é desenvolvido utilizando a linguagem Java tradicional, porém com algumas regras e restrições definidas pelo JCRE. Os *applets* podem ser incorporados no *smart card*, e geralmente são armazenadas na memória EEPROM.

Os aplicativos em Java usualmente são identificados por *strings* no padrão Unicode, porém em Java Card a identificação e seleção dos aplicativos é feita através da utilização de um *Application Identifier* (AID). O AID é uma sequência de bytes com tamanho entre 5 a 16 bytes e identifica um aplicativo de maneira única. Na Fig. 11 é descrita a estrutura que compõe um AID, que é definida pela ISO 7816-5.

Application Identifier (AID)	
Registered Application Provider Identifier (RID)	Proprietary Application Identifier Extension (PIX)
5 bytes	0 a 11 bytes

Figura 11 – Estrutura padrão do AID

A ISO controla a distribuição dos RIDs para as companhias, onde cada companhia tem seu único RID. As companhias então definem o valor do PIX da AID de acordo com sua necessidade.

Para o desenvolvimento do aplicativo é seguido uma estrutura de implementação típica, como pode ser vista na Fig. 12. Esta estrutura é composta por cinco métodos básicos que compõem o ciclo de vida do aplicativo, que são: *register, install, select, deselect* e *process.*

Todo *applet* deve estender o *javacard.framework.Applet*, que contém todos os métodos usados pelo JCRE. O método *install* é chamado pelo JCRE durante a instalação do *applet* no cartão e cria uma instância do mesmo, reservando recursos no *smart card* para sua execução. Após a instalação, o método *register* é chamado para registrar o *applet* instalado em uma tabela de *applets* para torná-lo selecionável. O método *select* é utilizado para inicializar/selecionar um *applet* específico que se encontra na tabela de *applets*. O método *deselect* é utilizado para “limpar” os recursos utilizados pelo *applet* selecionado. Por fim, o

método *process* recebe o APDU de comando e controla os métodos do *applet*, de acordo com o APDU recebido.

```
import javacard.framework.*;

public class MeuApplet extends Applet {
    private MeuApplet() {
        register();
    }

    public void install() {
        new MeuApplet();
    }

    public void select(){ ... }

    public void deselect(){ ... }

    public void process(APDU apdu) { ... }
}
```

Figura 12 – Estrutura de um aplicativo em Java Card

A Fig. 13 descreve o processo de desenvolvimento de um aplicativo em Java com suporte a tecnologia Java Card.

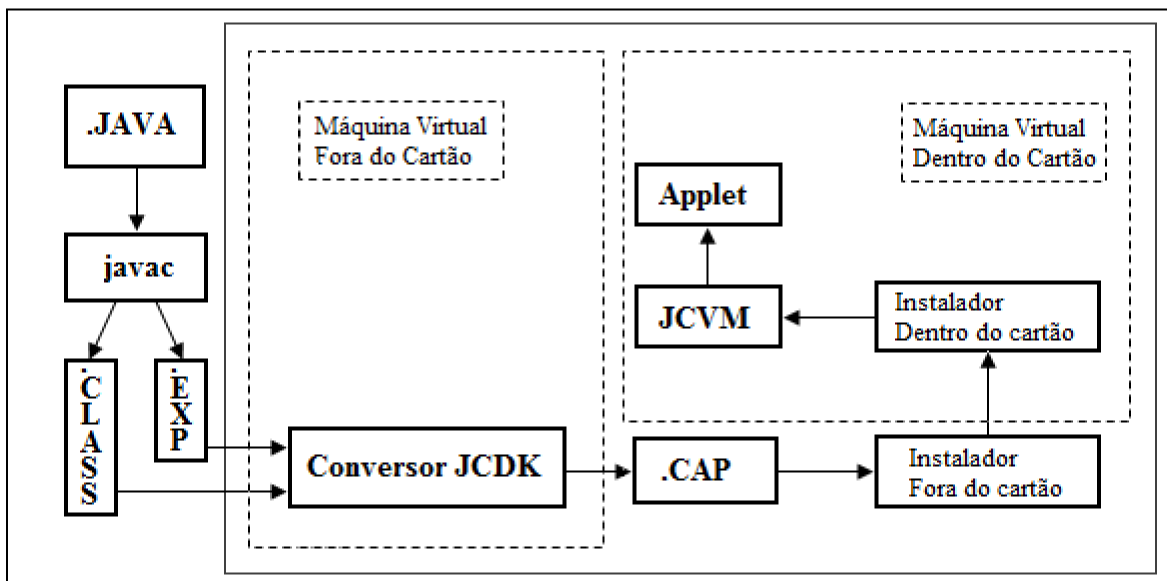

Figura 13 – Ciclo de Desenvolvimento de um aplicativo em Java Card

O *applet* é desenvolvido em qualquer IDE com suporte ao Java. Após a implementação, é feito o processo de compilação deste *applet* pela Máquina Virtual do Java

(que está fora do cartão, ou seja, reside em um PC), que gera um arquivo de classes contendo os *bytecodes* junto com um arquivo de exportação. Este arquivo de classes então é convertido e verificado pela plataforma de desenvolvimento Java Card (JCDK), que gera um arquivo convertido no formato CAP (formato utilizado pelo Java Card). Por fim o arquivo CAP é instalado no cartão (utilizando um software instalador), onde a JCVM (dentro do cartão) executa o *applet* caso o mesmo seja selecionado.

5.2 Algoritmos Criptográficos em Smart Cards

A principal característica dos smart cards, segundo Rankl (2007, p. 32) (tradução nossa) é de que "eles podem armazenar uma quantidade relativa de dados de maneira segura e oferecer um ambiente seguro para execução dos programas". Para isso são incorporadas dentro dos cartões funções criptográficas básicas, como algoritmos simétricos e assimétricos, algoritmos de *hash* e algoritmos geradores de chaves. Na Tabela 6 são descritas funções típicas encontradas em *smart cards* (RANKL, 2007, p. 27).

Tabela 5 – Algoritmos criptográficos geralmente encontrados em *smart cards*

Tipo de Algoritmo	**Algoritmo**
Algoritmos criptográficos simétricos	AES, DES, IDEA
Algoritmos criptográficos assimétricos	DAS, ECDSA, RSA
Algoritmos de *hash*	HMAC, MD5, SHA-1 e SHA-256
Geradores de chaves p/ algoritmos simétricos	Vários
Geradores de chaves p/ algoritmos assimétricos	Vários
Geradores de números aleatórios	Vários
Detecção de erros	CRC e Reed-Solomon

CAPÍTULO 6 – TECNOLOGIAS EMPREGADAS NO PROJETO

Neste projeto foi utilizado um leitor de cartões *GemPC Twin* com interface USB, fabricado pela Gemalto e mostrado pela Fig. 14, que foi utilizado para a importação dos aplicativos dentro do cartão e para o envio de dados. Quanto ao *smart card*, utilizamos um com suporte à Java Card desenvolvido pela NXP *Semiconductors*¸ modelo NXP P541G072. Para o desenvolvimento do algoritmo de hash é utilizado o ambiente de desenvolvimento Eclipse, juntamente com o plugin *JCOP Tools*, plugin desenvolvido inicialmente pela IBM e atualmente é de autoria da NXP Semiconductors. Todas as tecnologias empregadas no projeto (exceto a IDE Eclipse) foram disponibilizadas pelo Laboratório de Sistemas Integráveis Tecnológico (LSITEC) em parceria com o UNIVEM e este projeto.

Figura 14 – Leitor de cartões com interface USB

6.1 Java Card NXP P541G072

O *smart card* NXP P541G072 utilizado no desenvolvimento do projeto e descrito na Fig. 15 dá suporte a múltiplas aplicações e também trabalha em *Dual Interface*, pode-se utilizar tanto os contatos metálicos quanto rádio frequência para a comunicação ou transmissão de dados entre cartão e terminal. Além disso, ele disponibiliza métodos criptográficos como o *Triple Data Encryption Standard* (3DES) e o RSA, e funções de *hash* conhecidas, como o MD5 e o SHA.

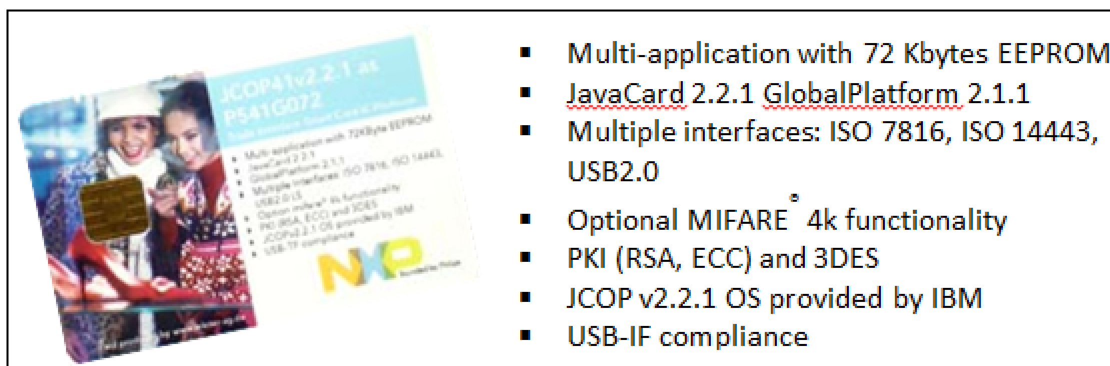

Figura 15 – Smart Card utilizado no projeto

Este *smart card* utiliza como seu Sistema Operacional a versão 2.2.1 do *Java Card Open Platform* (JCOP), que foi desenvolvida pela IBM e atualmente está sob autoria e responsabilidade da NXP *Semiconductors*. Com este sistema o *smart card* é capaz de suportar aplicações em Java que utilizam a tecnologia Java Card v2.2.1. O NXP P541G072 utiliza o modelo de hardware P5CT072, onde tem sua arquitetura descrita na Fig. 16 (PHILIPS, 2004).

Algumas características deste modelo são:

- Três tipos de memórias: RAM de 4608 bytes para armazenamento temporário de dados; ROM de 160 Kbytes é escrita no tempo de fabricação e contém o S.O. do cartão e todos os aplicativos do fabricante; EEPROM de 72 Kbytes onde são armazenadas os aplicativos criados e dados relevantes com retenção destas informações por no mínimo 20 anos.
- Dois coprocessadores para execução de funções criptográficas 3DES e AES.
- Coprocessador para execução do RSA e ECC.
- Velocidade de transferência de dados: 9600 (padrão), 19200, 38400, 115200, 230400 bit/seg. à 3,57 MHz (contato); 106000, 212000, 424000 bit/seg. (sem contato).

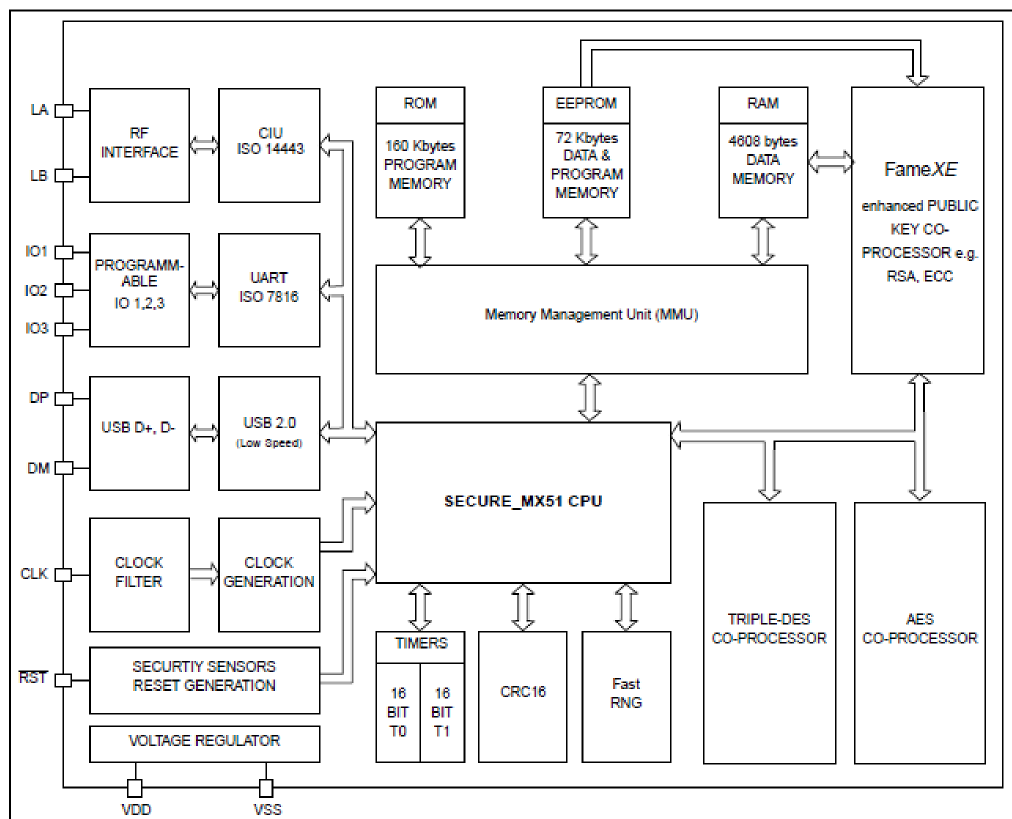

Figura 16 – Arquitetura do hardware smart card NXP P541G072

6.2 IDE Eclipse e JCOP Tools

Para a implementação da função Keccak na linguagem Java utilizamos 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.

Todo o processo de conectar o leitor com o JCOP *shell*, autenticar o cartão e o leitor e assegurar um canal de comunicação segura entre o cartão e o terminal é feita automaticamente pelo plugin JCOP *Tools,* dando-nos apenas o trabalho de implantar e executar o *applet* dentro do cartão. O processo de instalação, seleção e execução de um *applet* dentro do cartão é realizado através de comandos enviados para o cartão utilizando o JCOP *shell*, e que são descritos abaixo:

- É utilizado o comando *upload* como visto no trecho abaixo para realizar a importação do *applet* para dentro do cartão. Este comando importa para o cartão o arquivo CAP gerado que representa o *applet* e seu pacote (*package*).

```
cm>  upload  -b  250  "C:\workspace\KeccakJC\bin\jc\hash\keccak\
javacard\keccak.cap"
```

- Após importado, o *applet* é instalado na memória EEPROM do cartão e registrado na tabela de *applets* (utilizando seu AID) utilizando o comando *install,* visto abaixo.

```
cm> install -i 4b656363616b46756e63 -q C9#() 4b656363616b4b6563
63616b46756e63
```

- Após a instalação com sucesso o *applet* pode ser selecionado através de seu AID, através do comando */select.*

```
cm> /select 4B656363616B46756E63
```

- Após o *applet* ser selecionado, é possível executá-lo pelo cartão e/ou enviar dados utilizados pelo *applet* através de comandos APDU. Para ambos os casos é utilizado o comando *send*, onde são passados valores hexadecimais no formato do APDU de comando.

```
cm> send 80100000010A00
```

Todos estes comandos são comandos de alto nível (relativamente) disponibilizados pelo plugin JCOP *Tools*, que tem o trabalho de interpretá-los e convertê-los em comandos APDUs, ou seja, transformá-los em um conjunto ou sequência de valores hexadecimais para então enviar estes conjuntos ao cartão.

CAPÍTULO 7 – DESENVOLVIMENTO DO KECCAK EM JAVA

Esta seção apresenta as etapas de implementação do algoritmo Keccak na linguagem Java. É apresentada a função de permutações e a função de esponja, e suas etapas em especifico. 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 Keccak completa está descrita no Apêndice A.

7.1 Desenvolvimento da Função de Esponja

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*).

7.1.1 Etapa de Padding

A Fig. 17 representa a etapa de *padding* implementada na linguagem Java.

```
//padding
x = (short) (rbyte - (Mlen % (rbyte)));
byte[] P = new byte[Mlen + x];
Util.arrayCopy(M,(short)0,P,(short)0,Mlen);
if((Mlen % rbyte) != 0){
    P[Mlen] = (byte) 0x01;
    P[P.length-1] |= (byte) 0x80;
}
```

Figura 17 – Etapa de padding

Esta etapa faz o preenchimento da informação de entrada armazenada no vetor *M* para que ela se torne múltipla de *r*. Para isso, é criado um novo vetor *P* com tamanho múltiplo de *r*, e recebe os dados de *M*. Após recebido, *P* é “concatenado” com o valor *01* (Hexadecimal) seguidos de *n 0s* seguido por o valor *80* final. Isso torna a mensagem aplicável na etapa de absorção.

7.1.2 Etapa de Absorção

A Fig. 18 representa a etapa de absorção implementada na linguagem Java.

```
//absorbing
short S[] = new short[25];
byte[] Pi = new byte[rbyte];
short[] tmp = new short[rbyte/2];
short Plen = (short) P.length;
for (i = 0; i < Plen / rbyte; i++){
    Util.arrayCopy(P, (short) (i*Plen), Pi, (short) 0, rbyte);
    tmp = byteToShort(Pi);
    for (x = 0; x < rbyte/2; x++){
        S[x] ^= tmp[x];
    }
    S = Keccak400(S);
}
```

Figura 18 – Etapa de absorção

Esta etapa consiste em receber a mensagem já preenchida na etapa de *padding* e quebra-la em *Pi* blocos de tamanho *r.* Os dados de cada bloco *Pi* são inseridos no vetor de estados *S,* e então é aplicado a função de permutação sobre o vetor de estados. Esta etapa é executada enquanto existir blocos *Pi* para serem processados pela função de permutação.

Após finalizar esta etapa é realizada a etapa de compressão.

7.1.3 Etapa de Compressão

Por fim é realizada a etapa de compressão, que recebe o vetor de estados *S* com os dados processados na etapa de absorção, e gera o *hash* da mensagem de entrada. Para isso é criado um vetor *Z* com tamanho do *hash* especificado (no caso 224 bits), onde este vetor *Z* é

preenchido com o valor do vetor de estados *S*. Caso o tamanho do *hash* não seja atingido apenas com os valores do vetor de estados, é reaplicada a função de permutação para gerar novos valores necessários para o preenchimento completo de *Z*. Ao final deste processo é retornado o *hash* da mensagem.

A Fig. 19 representa a etapa de compressão implementada na linguagem Java.

```
//squeezing
x = outputLength;
short[] Z = new short[outputLength/16];
short Zlen = (short) Z.length;
while (x > 0){
    for (i = 0; i < Zlen; i++){
        Z[i] = S[i];
    }
    x -= r;
    if (x > 0){
        S = Keccak400(S);
    }
}
return shortToByte(Z);
```

Figura 19 – Etapa de compressão

7.2 Desenvolvimento da Função de Permutação

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 $n_r = 20$. A função possui cinco etapas com propósitos específicos, que são: θ, ρ, π, χ e ι, onde estas etapas serão executas 20 vezes.

7.2.1 Etapa θ

A Fig. 20 representa a etapa *θ* implementada na linguagem Java.

```
//Theta Step
for (x = 0; x < 5; x++){
    ram_B[x] = (short) (A[x] ^ A[5 + x] ^ A[10 + x]
                        ^ A[15 + x] ^ A[20 + x]);
}

for (x = 0; x < 5; x++){
    ram_tmp[0] = (short) (ram_B[index[x+4]]
                         ^ rot(ram_B[index[x+1]],(short)1));
    for (y = 0; y < 25; y += 5 ){
        A[x + y] ^= ram_tmp[0];
    }
}
```

Figura 20 – Etapa θ

Esta etapa consiste em realizar a operação XOR em cada "coluna" (se assumir o vetor como uma matriz 5x5) do vetor *S*, e armazená-las em um vetor *B*, que a seguir são realizadas rotações de 1 bit a esquerda e operações XOR entre os valores de *B* referentes as posições anterior e sucessor da posição atual.

7.2.2 Etapas ρ e π

A Fig. 21 representa as etapas *ρ* e *π* implementadas na linguagem Java.

```
//Rho and Pi Steps
ram_tmp[0] = A[1];
short t=0;
for (x = 0; x < 24; x++){
    t = indexPi[x];
    ram_B[0] = A[t];
    A[t] = rot(ram_tmp[0], rotationConstants[x] );
    ram_tmp[0] = ram_B[0];
}
```

Figura 21 – Etapas ρ e π

A etapa ρ consiste em realizar um número de rotações sobre os blocos de dados contidos no vetor de estados, onde esse número é determinado pelo vetor de rotações, representado na Fig. 22.

```
//Rotation Constants
private static final short rotationConstants[] = {
     1,  3,  6, 10, 15,
    21, 28, 36, 45, 55,
     2, 14, 27, 41, 56,
     8, 25, 43, 62, 18,
    39, 61, 20, 44
};
```

Figura 22 – Vetor de rotações utilizado na etapa ρ

A etapa π realiza permutações de dados, ou seja, a troca de posições entre os blocos do vetor. Estas etapas são realizadas juntas, pois a etapa π não altera os blocos, apenas realiza a troca de posição dos blocos como já dito anteriormente.

7.2.3 Etapa χ

A Fig. 23 representa a etapa χ implementada na linguagem Java.

```
//Chi Step
for (y = 0; y < 25; y += 5){
    ram_B[0] = A[y + 0];
    ram_B[1] = A[y + 1];
    ram_B[2] = A[y + 2];
    ram_B[3] = A[y + 3];
    ram_B[4] = A[y + 4];
    for (x = 0; x < 5; x++)
    {
        A[y + x] = (short) (ram_B[x] ^
             ((~ram_B[index[x+1]]) & ram_B[index[x+2]]));
    }
}
```

Figura 23 – Etapa χ

Esta etapa consiste em realizar operações lógicas XOR, AND e NOT entre cada “linha” (se assumir o vetor como uma matriz) do vetor de estados.

7.2.4 Etapa ι

A Fig. 24 representa a etapa *ι* implementada na linguagem Java.

```
//Iota Step
A[0] ^= roundConstants[round];
```

Figura 24 – Etapa ι

Esta etapa consiste simplesmente em realizar uma operação XOR entre o primeiro bloco do vetor de estados e um dos valores do vetor de constantes representado na Fig. 25, onde o valor utilizado é correspondente ao número da rodada atual.

```
//Round Constants
private static final short[] roundConstants = {
    (short)0x0001, (short)0x8082, (short)0x808A, (short)0x8000, (short)0x808B,
    (short)0x0001, (short)0x8081, (short)0x8009, (short)0x008A, (short)0x0088,
    (short)0x8009, (short)0x000A, (short)0x808B, (short)0x008B, (short)0x8089,
    (short)0x8003, (short)0x8002, (short)0x0080, (short)0x800A, (short)0x000A
};
```

Figura 25 – Vetor de constantes da rodada utilizado na etapa ι

CAPÍTULO 8 – ANÁLISE E RESULTADOS DA FUNÇÃO KECCAK

Este capítulo apresenta a análise da função Keccak e seu comportamento dentro do cartão. Em termos de operações lógicas, a função *Keccak-f* [*b*] utiliza aproximadamente (DAEMEN *et al,* 2011a):

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

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

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

Em relação à utilização de memória, temos os dados representados na Tabela 6. Estes valores foram retirados com o uso de comandos dentro do cartão, disponibilizados pelo Java Card.

Tabela 6 – 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 7 descreve uma comparação com as funções de *hash* MD5 e SHA, ambas disponibilizadas pelo cartão através da tecnologia Java Card.

Executamos estas funções dentro do cartão, passando um valor de entrada padrão.

Tabela 7 – 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 7, 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 segundo. Este tempo elevado é indesejável, tornando seu uso (no momento atual) impraticável.

CONCLUSÕES

Na implementação da função não obteve problemas para adaptá-la à tecnologia Java Card, em grande parte por suportar a linguagem Java, o que facilitou em muito a implementação. Vale ressaltar que utilizamos 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 conseguimos o tempo de processamento descrito na seção 9, que é de aproximados 37 segundos, contra os 60 segundos onde a função Keccak apenas utilizava a memória EEPROM do cartão. 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. N

Uma das etapas do projeto proposto foi a etapa de comparação dos resultados obtidos com trabalhos correlatos, que não pôde ser concluída com o nível de detalhe esperado, devido ao fato que as informações coletadas são básicas, mostrando apenas informações de tempo de execução e utilização da memória pela função. Para realizar a comparação com trabalhos correlatos, seriam necessárias informações referentes aos ciclos/bytes (número de ciclos de clock para processar um byte) e *troughtput* (quantidade de dados processados em determinado tempo) do *smart card*.

Estas duas informações são usualmente utilizadas para indicar o desempenho das funções criptográficas nas arquiteturas utilizadas, essencial para realizar comparações de desempenho entre arquiteturas de hardware diferentes.

Para recolher tais informações seria necessário criar um programa de *benchmark* (programa que avalia o desempenho de um CPU, por exemplo) específico para *smart cards*, devido ao fato de não existir nenhum programa funcional, público e aberto. Tais programas existem apenas para uso interno de empresas ou organizações, ou seja, não são distribuídos no mercado.

No entanto, a implementação descrita neste projeto executa de fato em smart cards, diferentemente do trabalho correlato (GOUICEM, 2010) encontrado, que não especifica se realmente a função Keccak foi implementada em uma arquitetura de smart card real ou apenas foi aferido o número de ciclos para o microcontrolador comumente adotado como unidade de processamento e controle em *smart cards*.

Isto tem impacto direto nos resultados, uma vez que não são considerados atrasos da arquitetura do *smart card* completo, como mecanismos de entrada e saída (comunicação) e os dispositivos de memória. Elementos estes que são considerados gargalos em qualquer arquitetura, o que pode levar a distorção dos resultados.

REFERÊNCIAS

BISHOP, Matt. **Computer Security:** Art and Science. Boston: Addison-Wesley, 2003.

BURNETT, Steve; PAINE, Stephen. **Criptografia e segurança:** O Guia Oficial RSA. Tradução de Edson Fumankiewicz. Rio de Janeiro: Campus, 2002.

CARDOSO, Thiago Xavier. **Smart Card & Java Card.** 2008. Disponível em <http://www.gta.ufrj.br/ ensino/eel879/trabalhos_vf_2008_2/thiago/index.html>. Acesso em 20 jun. 2010.

CHEN, Zhiqun. **Java Card Technology for Smart Cards:** Architecture and Programmer's Guide. Addison-Wesley, 2000.

DAEMEN, Joan *et al.* **Cryptographic Sponges.** 2009. Disponível em <http://sponge. noekeon.org/>. Acesso em 20 nov. 2010.

DAEMEN, Joan *et al.* **Keccak Implementation Overview.** Version 3.1, 2011a. Disponível em < http://keccak.noekeon.org/Keccak-implementation-3.1.pdf>. Acesso em 20 ago. 2011.

DAEMEN, Joan *et al.* **The Keccak Reference.** Version 3, 2011b. Disponível em <http://keccak.noekeon.org/Keccak-reference-3.0.pdf>. Acesso em 10 mai. 2011.

DAEMEN, Joan *et al.* **The Keccak SHA-3 Submission.** Version 3, 2011c. Disponível em <http://keccak.noekeon.org/Keccak-submission-3.pdf>. Acesso em 20 mai. 2011.

FILHO, Antonio Mendes Da Silva. **Segurança da informação:** Sobre a Necessidade de Proteção de Sistemas de Informações. Revista Espaço Acadêmico, [S.1], nº 42, nov. 2004. Disponível em: <http://www.espacoacademico.com.br/042/42amsf.htm>. Acesso em: 14 mai. 2011.

GOUCIEM, Mourad. **Comparison of Seven SHA-3 Candidates Software Implementations on Smart Cards.** Disponível em <http://eprint.iacr.org/2010/531.pdf>. Acesso em: 20 set. 2011.

MENEZES, Alfred J.; OORSCHOT, Paul C.; VANSTONE, Scott A.; **Handbook of Applied Cryptography.** CRC Press, 1996. Disponível em: <http://www.cacr.math.uwaterloo.ca/hac/>. Acesso em: 14 mai. 2011.

MORENO, Edward David; PEREIRA, Fábio Dacêncio; CHIARAMONTE, Rodolfo Barros. **Criptografia em Software e Hardware.** São Paulo: Novatec, 2005.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). **Tentative Timeline of the Development of New Hash Functions.** 2008a Disponível em: <http://csrc.nist.gov/groups/ST/hash/timeline.html>. Acesso em: 11 dez. 2010.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). **Cryptographic Hash Algorithm Competition.** 2008b Disponível em: <http://csrc.nist.gov/groups/ST/hash/sha-3/index.html>. Acesso em: 11 dez. 2010.

PHILIPS SEMICONDUCTORS. P5CT072 **Secure Dual Interface PKI Smart Card Controller.** 2004. Disponível em: <http://www.classic.nxp.com/acrobat_download2/other/identification/sfs085513.pdf>. Acesso em 10 out. 2011.

RANKL, Wolfgang. **Smart Cards Applications:** Design Models for using and programming smart cards. Chichester: John Wiley & Sons, 2007.

SCHNEIER, Bruce. **Applied Cryptography:** Protocols, Algorithms, and Source Code in C. 2nd Edition. New York: John Wiley & Sons, 1996.

SUN MICROSYSTEMS. **Java Card v2.2.2 API.** Disponível em <http://www.win.tue.nl/pinpasjc/docs/apis/jc222/index.html?overview-summary.html>. Acesso em 15 jun. 2010.

WANG, Xiaoyun *et al*. **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD.** Crypto'04, 2004. Disponível em: <http://eprint.iacr.org/2004/199.pdf>. Acesso em: 14 mai. 2011.

WANG, Xiaoyun; YIN, Yiqun Lisa; YU, Hongbo. **Finding Collisions in the Full SHA-1.** Crypto'05, 2005. Disponível em: <http://www.informatik.uni-trier.de/~ley/db/conf/crypto/crypto2005.html>. Acesso em: 14 mai. 2011.

APÊNDICE A – Função Keccak implementada em Java

```
package javacard.hash.keccak;

import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.APDU;
import javacard.framework.Util;
import javacard.framework.JCSystem;
/** The Keccak Function, design by Guido Bertoni,
 * Joan Daemen,Michael Peeters and Gilles Van Assche.
 *
 * For more information, visit que keccak
 * website: http://keccak.noekeon.org/
 *
 * Implementation by Fernando Yokota */
public class Keccak extends Applet {
	//definicao do campo CLA do APDU
	final static byte CLA_KECCAK = (byte) 0x80;
	//definicao do campo INS do APDU
	final static byte INS_KECCAKF = (byte) 0x10;
	//criacao de vetores na memoria RAM, mais rapida que EEPROM
	short[] ram_B = JCSystem.makeTransientShortArray( (short) 5, JCSystem.CLEAR_ON_DESELECT);
	short[] ram_tmp = JCSystem.makeTransientShortArray((short) 1, JCSystem.CLEAR_ON_DESELECT);

	public static void install(byte[] bArray, short bOffset, byte bLength) {
		new Keccak().register(bArray, (short) (bOffset + 1), bArray[bOffset]);
	}
	public void process(APDU apdu) {
		// Good practice: Return 9000 on SELECT
		if (selectingApplet()) {
			return;
		}
		//vetor buf recebe o apdu de comando
		byte[] buf = apdu.getBuffer();
		//vetor que recebe hash
		byte[] hash;
		switch (buf[ISO7816.OFFSET_INS]) {
			case INS_KECCAKF:
				//tamanho dos dados de é recebido
				short lgth = apdu.setIncomingAndReceive();
				if (lgth == 0){
					ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
				}
				byte[] data = new byte[lgth];
				//copia os dados recebidos para o vetor data
				Util.arrayCopy(buf, ISO7816.OFFSET_CDATA, data, (short) 0, lgth);
				//executa a funcao de hash sobre data
				hash = KeccakF(data);
				//copia o hash para o buffer do apdu
				Util.arrayCopyNonAtomic(hash, (short)0, buf, (short)0, (short)hash.length);
				//envia buf como apdu de resposta juntamente com o hash
				apdu.setOutgoingAndSend((short)0, (short)hash.length);
				break;
			default:
				ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
		}
	}
```

```
//capacity
final static short c = (short) 256;
//bitrate
final static short r = (short) (400-c);
//bitrate em bytes
final static short rbyte = (short) (r/8);
//tamanho do hash
final static short outputLength = (short) 224;

//funcao de esponja
public byte[] KeccakF(byte[] M){
	short i = 0;
	short x = 0;
	short Mlen = (short) M.length;
	//padding
	x = (short) (rbyte - (Mlen % (rbyte)));
	byte[] P = new byte[Mlen + x];
	Util.arrayCopy(M,(short)0,P,(short)0,Mlen);
	if((Mlen % rbyte) != 0){
		P[Mlen] = (byte) 0x01;
		P[P.length-1] |= (byte) 0x80;
	}
	//absorbing
	short S[] = new short[25];
	byte[] Pi = new byte[rbyte];
	short[] tmp = new short[rbyte/2];
	short Plen = (short) P.length;
	for (i = 0; i < Plen / rbyte; i++){
		Util.arrayCopy(P, (short) (i*Plen), Pi, (short) 0, rbyte);
		tmp = byteToShort(Pi);
		for (x = 0; x < rbyte/2; x++){
			S[x] ^= tmp[x];
		}
		S = Keccak400(S);
	}
	//squeezing
	x = outputLength;
	short[] Z = new short[outputLength/16];
	short Zlen = (short) Z.length;
	while (x > 0){
		for (i = 0; i < Zlen; i++){
			Z[i] = S[i];
		}
		x -= r;
		if (x > 0){
			S = Keccak400(S);
		}
	}
	return shortToByte(Z);
}
```

```
//converte vetor de bytes para vetor de shorts
private static short[] byteToShort(byte[] b) {
	short[] c = new short[b.length/2];
	byte aux[] = new byte[2];
	for (short i = 0; i < c.length; i++){
		for (short j = 0; j < 2; j++){
			aux[j] = b[2*i + j];
		}
		c[i] = (short) ((((short) aux[1]) & 0xFF) |  ((((short) aux[0]) & 0xFF) << 8));
	}
	return c;
}

//converte vetor de shorts para vetor de bytes
private static byte[] shortToByte(short[] b){
	byte[] c = new byte[b.length*2];
	short y = 0;
	for (short x=0; x<c.length;x+=2){
		c[x+1] = (byte) (b[y] & 0xFF);
		c[x] = (byte) ((b[y] >> 8) & 0xFF);
		y++;
	}
	return c;
}

//metodo de rotacao de bits
private static short rot(short n, short i){
	return (short) ((n << i) | (n >>> (16-i)));
}

//Round Constants
private static final short[] roundConstants = {
	(short) 0x0001, (short) 0x8082, (short) 0x808A, (short) 0x8000, (short) 0x808B,
	(short) 0x0001, (short) 0x8081, (short) 0x8009, (short) 0x008A, (short) 0x0088,
	(short) 0x8009, (short) 0x000A, (short) 0x808B, (short) 0x008B, (short) 0x8089,
	(short) 0x8003, (short) 0x8002, (short) 0x0080, (short) 0x800A, (short) 0x000A
};

//Rotation Constants
private static final short rotationConstants[] = {
	1,  3,  6, 10, 15,
	21, 28, 36, 45, 55,
	2, 14, 27, 41, 56,
	8, 25, 43, 62, 18,
	39, 61, 20, 44
};

//Index utilizado na etapa pi
private static final short indexPi[] = {
	10,  7, 11, 17, 18, 3,
	5, 16,  8, 21, 24, 4,
	15, 23, 19, 13, 12, 2,
	20, 14, 22,  9,  6, 1
};

//Index utilizado na funcao de permutacao
private static final short index[] = {0, 1, 2, 3, 4, 0, 1, 2, 3, 4};
```

```
		//funcao de permutacao
		private short[] Keccak400(short[] A){
			short x = 0;
			short y = 0;
			//short[] B = new short[5];
			short round = 0;
			for (round = 0; round < 20; round++){
				//Theta Step
				for (x = 0; x < 5; x++){
					ram_B[x] = (short) (A[x] ^ A[5 + x] ^ A[10 + x] ^ A[15 + x] ^ A[20 + x]);
				}
				for (x = 0; x < 5; x++){
					ram_tmp[0] = (short) (ram_B[index[x+4]]
							^ rot(ram_B[index[x+1]], (short)1));
					for (y = 0; y < 25; y += 5 ){
						A[x + y] ^= ram_tmp[0];
					}
				}
				//Rho and Pi Steps
				ram_tmp[0] = A[1];
				short t=0;
				for (x = 0; x < 24; x++){
					t = indexPi[x];
					ram_B[0] = A[t];
					A[t] = rot(ram_tmp[0], rotationConstants[x] );
					ram_tmp[0] = ram_B[0];
				}
				//Chi Step
				for (y = 0; y < 25; y += 5){
					ram_B[0] = A[y + 0];
					ram_B[1] = A[y + 1];
					ram_B[2] = A[y + 2];
					ram_B[3] = A[y + 3];
					ram_B[4] = A[y + 4];
					for (x = 0; x < 5; x++)
					{
						A[y + x] = (short) (ram_B[x] ^
								((~ram_B[index[x+1]]) & ram_B[index[x+2]]));
					}
				}
				//Iota Step
				A[0] ^= roundConstants[round];
			}
			return A;
		}
}
```