XII SIGE ISSN: 1983 7402 São José dos Campos, 28 de setembro a 01 de outubro de 2010

Geração Remota de Números Aleatórios Reais para Aplicações Envolvendo Segurança em Sistemas Móveis Dedicados

Fabricio da Silva Soares, Mario T. Shimanuki e Wagner Chiepa Cunha
ITA – Instituto Tecnológico de Aeronáutica – Praça Marechal Eduardo Gomes, 50 – São José dos Campos – SP – Brasil

***Resumo* — Sistemas móveis dedicados, tais como *Smart Phones*, *Hand Held*, PDA *(Personal Digital Assistant)*, etc., quando utilizados para executar aplicações seguras (aplicações envolvendo criptografia) necessitam de uma fonte de geração de números aleatórios (por exemplo, no processo de geração dos pares de chaves do algoritmo RSA). Números aleatórios gerados via *softwares* de uso geral, oferecem um baixo grau dispersão. Neste contexto, no presente trabalho foi utilizada a bateria de testes NIST e Diehard em seqüências de números aleatórios para analisar as características estatísticas e a complexidade linear do gerador. O trabalho, também propõe uma arquitetura para a Geração Remota de Números Aleatórios Reais (*Remote True Randon Number Generator* - RTRNG).**

***Palavras-Chave* — Segurança de Informações, Geração Remota de Números Aleatórios Reais (RTRNG) e Computação Confiável.**

I. INTRODUÇÃO

O poder de processamento dos sistemas móveis dedicados vem crescendo nos últimos anos. Uma das características dos sistemas móveis dedicados é que estes foram concebidos para que não sejam incorporados a módulos ou *hardwares* externos. Podemos utilizar esses somente através dos canais de E/S disponíveis (*Wi-Fi*, *Bluetooth*, *slots* de cartão externo, portas USB, etc.)

Caso o sistema móvel dedicado esteja executando um aplicativo envolvendo criptografia assimétrica (VoIP cifrado, armazenamento e transferência de arquivos cifrados, etc), para se ter um maior nível de segurança, o gerador de números aleatórios deve ser um gerador eficiente.

Uma seqüência de números gerada deterministicamente por *software* é necessariamente determinística (pseudo-aleatória). De acordo com John von Newmann [1] “*Quem quer que seja que considere métodos aritméticos para produzir números aleatórios está, claro, num estado de pecado”*.

De acordo com [2], o conceito de segurança está intimamente ligada à dificuldade de se distinguir, em termos computacionais, uma seqüência pseudo-aleatória de outra realmente aleatória e com distribuição uniforme.

Fabricio da Silva Soares, fabricio@fabricio.net.br, Mario T. Shimanuki, explorer@ita.br, Wagner Chiepa Cunha, chiepa@ita.br, Tel +55-12-3947-5878, Fax +55-12-3947-6930.

Segundo [3] "*A segurança de todo o sistema é tão forte quanto seu elo mais fraco. Tudo deve ser seguro: os algoritmos criptográficos, os protocolos, o gerenciamento de chaves, entre outros. Se o algoritmo é ótimo, mas o gerador de números aleatórios é deficiente, um criptoanalista experiente atacará o sistema através da geração de números aleatórios. [...] A criptografia é somente uma parte da segurança, e usualmente uma parte muito pequena*".

Diante do exposto, é inegável a necessidade de se ter uma fonte segura de geração de números aleatórios em aplicações envolvendo segurança. Neste trabalho são apresentados os testes NIST [4] e Diehard [5] realizados para a verificação dos seguintes geradores de números pseudo-aleatórios:

1. Linguagem C/C++: função *rand()*;
2. Biblioteca OpenSSL: *RAND_pseudo_bytes()*;
3. Linguagem Java: método *Math.rand()*;
4. BlackBerry Java: classe *Random*;

Para a geração de números pseudo-aleatórios com a função *rand()* foi utilizado a biblioteca do pacote de *software* MS Visual Studio 2010 para C/C++. O OpenSSL (*Open Secure Sockets Layer*) é uma biblioteca com primitivas criptográficas, foi utilizado a função *RAND_pseudo_bytes()*, do OpenSSL versão 1.0.0. O método *Math.rand()* foi utilizado com o pacote Java JDK 6.

A RIM (*Research In Motion*), fabricante dos *Smartphones* BlackBerry, disponibiliza em seu sítio um ambiente *Open Source* de desenvolvimento de aplicativos para os seus aparelhos, denominado *BlackBerry Java Application Development v5.0* [6].

Os mesmo testes também foram realizados no seguinte gerador de números aleatórios reais:

1. Alea I: *True Random Number Generator* (TRNG) Alea I.

Para se conseguir uma cadeia de números aleatórios reais, foi utilizado um Gerador de Números Aleatórios Reais (TRNG) baseado em *hardware* (Alea I da empresa Araneus). Alea I é capaz de gerar 100 *kbits/s* [7], este utiliza um semicondutor polarizado reversamente para gerar uma banda Gaussiana de ruído branco. Este ruído é amplificado e digitalizado utilizando um conversor analógico digital (A/D).

A cadeia de *bits* advindos do conversor A/D é então processada pelo microcontrolador, combinando a entropia de uma cadeia de amostras e removida a polarização causada por imperfeições no gerador de ruído e no conversor A/D.

Foram geradas cadeias com 10.000.000 de números de 32 *bits* inteiros sem sinal, utilizando cada um dos geradores de números pseudo-aleatórios / aleatórios supracitados. Estas cadeias foram particionadas em blocos de acordo com as recomendações para os testes Diehard [5] e NIST [4], antes dos testes de aleatoriedade.

II. Bateria de Testes em Geradores de Números Aleatórios

Na atual versão (sts-2.1), a bateria de testes NIST (*National Institute of Standards and Technology*) reúne dezesseis testes de aleatoriedade. Alguns criados por Donald Knuth, como o teste serial e outros por George Masaglia como o teste de posto de matriz binária [8].

A bateria de testes NIST, fornece como resultado valores *p* (*p-values*). Se $p \leq 0{,}01$ significa que a seqüência não é aleatória com uma confiança de 99%. Sendo que se o valor *p* calculado é maior que 0,01 (ou 1%), considera-se que a seqüência passou no teste.

Para facilitar a compreensão dos resultados dos testes aplicados nesse trabalho, a bateria de testes NIST foi renomeada da seguinte forma:

1. **NIST01:** Teste de freqüência;
2. **NIST02:** Teste de freqüência no interior de um bloco;
3. **NIST03:** Teste de séries;
4. **NIST04:** Teste de série mais extensa do 1s em um bloco;
5. **NIST05:** Teste de posto de matriz binária;
6. **NIST06:** Teste de transformada discreta de Fourier;
7. **NIST07:** Teste de equiparação ao modelo sem superposição;
8. **NIST08:** Teste de equiparação ao modelo com superposição;
9. **NIST09:** Teste estatístico universal de Maurer;
10. **NIST10:** Teste de Complexidade Linear;
11. **NIST11:** Teste Serial;
12. **NIST12:** Teste de entropia aproximada;
13. **NIST13:** Teste de somas cumulativas (CUSUM);
14. **NIST14:** Teste de excursões aleatórias;
15. **NIST15:** Teste variante de excursões aleatórias.

George Masaglia criou a bateria de testes de aleatoriedade Diehard com o objetivo de substituir os testes convencionais criados por Donald Knuth [9].

A bateria de testes Diehard, semelhante ao NIST, também fornece como resultado valores *p*, que deveriam ser uniformes em [0,1) se o arquivo de entrada contivesse *bits* aleatórios realmente independentes.

Os valores de *p* (*p-values*) são obtidos por $p=F(x)$, onde F é a distribuição considerada para a variável aleatória X da amostra, usualmente a normal. No entanto, a distribuição considerada é apenas uma aproximação assintótica, motivo pelo qual a adequação é pior nas extremidades. Desta forma, quando o fluxo de *bits* falha em determinado teste, obtém-se *p* de 0 ou 1 em seis ou mais itens do teste.

A bateria de testes Diehard foi renomeada da seguinte forma:

1. **DIEH01:** Teste de espaçamento de aniversários;
2. **DIEH02:** Teste de permutação de 5 inteiros com superposição;
3. **DIEH03:** Testes de posto de matrizes binárias;
4. **DIEH04:** Testes de fluxo de *bits*;
5. **DIEH05:** Testes OPSO, OQSO e DNA;
6. **DIEH06:** Teste de contagem de 1s;
7. **DIEH07:** Teste do pátio de estacionamento;
8. **DIEH08:** Teste de distância mínima;
9. **DIEH09:** Teste das esferas 3D;
10. **DIEH10:** Teste de compressão;
11. **DIEH11:** Teste de somas sobrepostas;
12. **DIEH12:** Teste de séries;
13. **DIEH13:** Testes do jogo de dados.

III. Análise de Resultados da Bateria de Testes NIST e Diehard

O número de falhas ocorrido em cada teste do NIST e Diehard aplicados nas cadeias com 10.000.000 números aleatórios gerados nessa pesquisa são ilustrados, respectivamente, na **Tab. I** e **Tab. II**:

Tab. I: Resultado dos testes do NIST

Teste	C/C++	OpenSSL	Java	BlackBerry	Alea I
NIST01	1	0	0	0	0
NIST02	1	0	0	0	0
NIST03	falhou	0	0	0	0
NIST04	1	0	0	0	0
NIST05	1	0	0	0	0
NIST06	1	0	0	0	0
NIST07	148	0	0	0	3
NIST08	1	0	0	0	0
NIST09	1	0	0	0	0
NIST10	0	0	0	0	0
NIST11	2	0	0	0	0
NIST12	1	0	0	0	0
NIST13	2	0	0	0	0
NIST14	falhou	0	0	falhou	0
NIST15	falhou	0	1	falhou	2
Falhas	160	0	1	2	5

Legenda:

- (verde) Houve 5 ou menos falhas (passou no teste);
- (vermelho) Houve 6 ou mais falhas (falha no teste).

Para determinar o sucesso ou a falha de um gerador de números aleatórios, foi adotado o mesmo critério de [2] para a bateria de testes do NIST e de Diehard. Esse critério determina que, um bom gerador de números aleatórios apresenta menos de 6 falhas em cada um dos testes.

Tab. II: Resultado dos testes de Diehard

Teste	C/C++	OpenSSL	Java	BlackBerry	Alea I
DIEH01	9	0	0	0	1
DIEH02	0	0	2	0	1
DIEH03	19	3	1	0	0
DIEH04	20	1	0	2	1
DIEH05	75	5	15	13	3
DIEH06	19	2	2	1	1
DIEH07	10	0	0	1	1
DIEH08	1	0	0	0	0
DIEH09	20	1	1	1	5
DIEH10	falhou	0	0	0	0
DIEH11	falhou	0	0	2	0
DIEH12	falhou	0	0	0	1
DIEH13	falhou	0	0	0	0
Falhas	173	12	21	20	14

Legenda:

- Houve 5 ou menos falhas (passou no teste);
- Houve 6 ou mais falhas (falha no teste).

A **Tab. I** mostra os resultados dos testes do NIST. As cadeias numéricas criadas pelos geradores de números aleatórios da Linguagem C/C++ não continham dados consistentes para realizar algumas baterias de testes do NIST. Por esse motivo, algumas sequências apresentaram graves falhas durante os testes (destacados nas tabelas como “falhou”).

Os demais geradores apresentaram resultados satisfatórios, sendo que o gerador de números aleatórios da Biblioteca OpenSSL não apresentou nenhum erro nas baterias testadas. Do mesmo modo que nos testes de [2], nenhuma variante apresentou algum tipo de falha no teste de Complexidade Linear do NIST.

Comparando os resultados obtidos pelos testes do NIST e de Diehard, tanto a Biblioteca OpenSSL quanto o *hardware* Alea I geram boas sequências de números aleatórios. Foi possível observar que, conforme [9], o número de falhas apresentadas na bateria de testes do NIST é bem menor do que na bateria de testes Diehard, assim, devido a sua eficiência e acurácia a bateria de testes Diehard é uma forte candidata a substituto da bateria de testes NIST.

IV. Arquitetura para Geração Remota de Números Aleatórios Reais (RTRNG)

O TCG (*Trusted Computing Group*) [10] é uma organização criada para produzir, definir e promover especificações abertas para o desenvolvimento de plataformas computacionais seguras. Este grupo tem como objetivo desenvolver o conceito e a especificação de uma plataforma computacional confiável.

O TCG criou a especificação para o *chip* TPM (*Trusted Module Platform*) [11, 12 e 13]. Atualmente o TPM é produzido pelas seguintes companhias: Atmel; Broadcom; Infineon (Infineon TPM); Intel (Intel Manageability Engine - iTPM); Sinosun; Nuvoton (Winbond) STMicroelectronics, Toshiba; e ITE (ITE TPM).

A **Fig. 1** ilustra as funcionalidades gerais de um TPM. O TPM efetua funções criptográficas, funções de mão única (funções *hash*), funções de segurança física, provêem identificação do sistema e possui um gerador de números aleatórios.

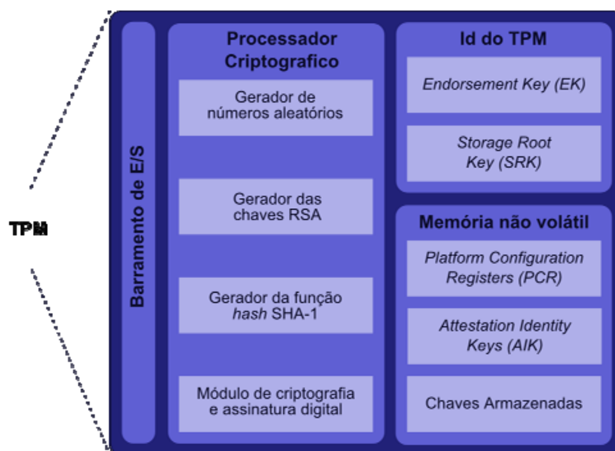


Fig. 1. Funcionalidades de um TPM

O *Endorsement Key* (EK) é uma chave RSA de 2048 *bits* que é criada durante o processo de fabricação do *chip* e este não pode ser trocada. Mecanismos de segurança impedem que a chave privada seja visualizada fora do *chip*. A chave pública é utilizada para atestar e enviar informações sensíveis ao *chip*.

Semelhante ao EK o *Storage Root Key* (SRK) é um par de chaves RSA, também com 2048 *bits*, no entanto ela é criada pelo dono (*owner*) do TPM, durante o processo de inicialização do mesmo.

O *Platform Configuration Register* (PCR) são 16 registradores de 160 *bits* destinados a armazenar a função *hash* SHA-1 de determinados *hardwares* e *softwares*. Este tem o propósito de garantir que estes não foram alterados.

O *Attestation Identity Key* (AIK) é a chave de atestação de identidade. Utilizada para atestar a veracidade das “Chaves Armazenadas” (chaves RSA).

A Fig. 2 ilustra um diagrama de blocos simplificado da interligação do TPM e do TRNG com um microcontrolador.

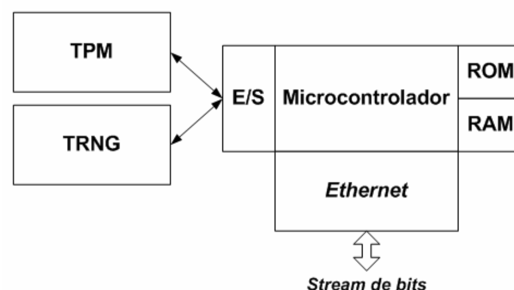


Fig. 2. Interligação do TPM com u microcontrolador

O TPM irá assegurar a confiança das informações, através de assinatura digital, e o TRNG irá gerar o *stream* de *bits*.

A **Fig. 3** apresenta uma implementação da arquitetura proposta (RTRNG) baseada no TPM AT97SC3203S da empresa Atmel com o gerador de números aleatórios reais (TRNG) Araneous Alea I, controlados pelo *Single Computer Board* picoFlash da empresa JK Microsystems.

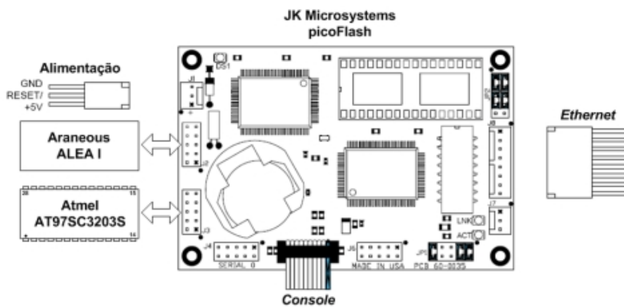


Fig. 3. RTRNG utilizando o TPM Atmel AT97SC3203S

O *Single Computer Board* picoFlash possui as seguintes características: DOS e *Web Server* embarcados, pilha TCP/IP, sistema de arquivos *flash*, processador 186 de 40 MHz, 512K de memória *flash*, 512K de memória RAM, conector *Ethernet* 10Base-T, 16 canais digitais de E/S, 2 portas seriais, *watchdog* e *timer* de 16 *bits*.

A comunicação do TPM Atmel AT97SC3203S com os dispositivos externos é feita através do barramento SMBus [14]. O protocolo SMBus é um protocolo de comunicação por dois fios (**Fig. 4**). O sinal de *clock* (SMBCLK) deve ser fornecido pelo sistema, a transferência de dados é feita por uma porta de E/S bidirecional (SMBDAT). O sistema deve prover um outro sinal de *clock* ao terminal AVRCLK do TPM.

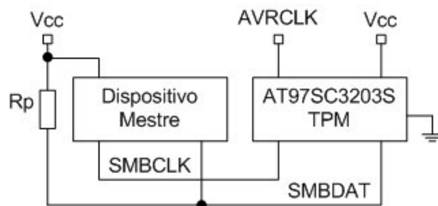


Fig. 4. Topologia do barramento SMBus

O TPM AT97SC3203S da Atmel possui um gerador de números aleatórios reais baseado em um evento físico com alta entropia (ruído térmico, desintegração radioativa, etc.) [15]. Desta forma, o TPM pode garantir uma fonte de números aleatórios reais para a geração das chaves RSA, além de possuir o módulo de "criptografia e assinatura digital", utilizados para garantir a identidade e integridade do sistema.

Esse TPM possui um acelerador criptográfico capaz de processar assinaturas RSA de 2048 *bits* em 500 *ms* e de 1024 em 100 *ms*. O processamento da função *hash* SHA1 é de 50 μs por blocos de 64 *bytes*. Todos em conformidade com a FIPS 140-2 [16].

Apesar do TPM possuir a capacidade de gerar chaves RSA (estes criados com uma fonte de aleatoriedade real), na arquitetura proposta, optou-se por enviar somente os números aleatórios gerados pelo Alea I, evitando-se assim que o RTRNG tenha conhecimento de todos os pares de chaves RSA gerados para os clientes.

O RTRNG irá efetuar as seguintes tarefas:

1. Envio de um cabeçalho assinado (identificação do servidor remoto) contendo informações sobre o *stream* de *bits*.
2. Envio de um *stream* de *bits* aleatórios via *ethernet* ao cliente.

A **Fig. 5**. ilustra uma solicitação de um *stream* de *bits* ao RTRNG.

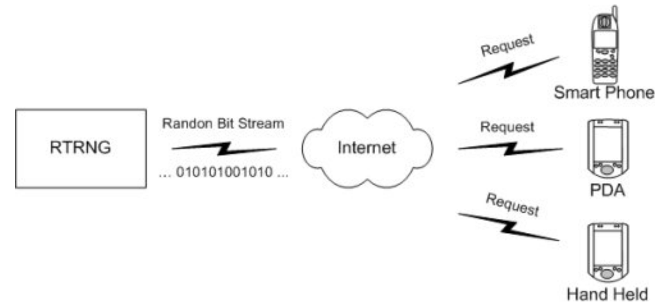
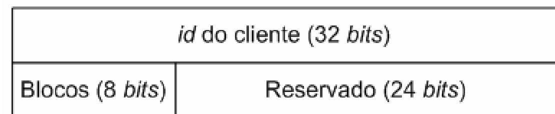


Fig. 5. Solicitação pelo cliente de um *stream* de *bits*

O RTRNG possui um par de chaves RSA gerado pelo TPM (Pri_{RTRNG} e Pub_{RTRNG}), visto que estas chaves foram gerados pelo TPM, Pri_{RTRNG} jamais poderá ser exportada, garantindo-se desta forma a identidade do sistema.

Como Pub_{RTRNG} é conhecido por todos, a solicitação de um *stream* de *bits* (*request*) é feito cifrando-se com Pub_{RTRNG} o campo mostrado na **Fig. 6**.



id do cliente (32 *bits*)	
Blocos (8 *bits*)	Reservado (24 *bits*)

Fig. 6. Campo de requisição

Onde:

id do cliente: identificação do cliente;
Bloco: quantidade dos blocos solicitados, onde cada bloco corresponde a 512 *bits* (perfazendo-se um máximo de $2^8 \times 512$ *bits* por requisição);
Reservado: campo reservado para utilização futura.

O campo "Blocos" diz respeito à quantidade de blocos que o cliente quer receber, e não a quantidade de blocos necessários para a utilização na aplicação. Desta forma existe a garantia que nem mesmo o servidor ficará sabendo do número aleatório utilizado pelo cliente.

O RTRNG irá decifrar a solicitação utilizando Pub_{RTRNG}, e enviar o *stream de bits* (**Fig. 7**).

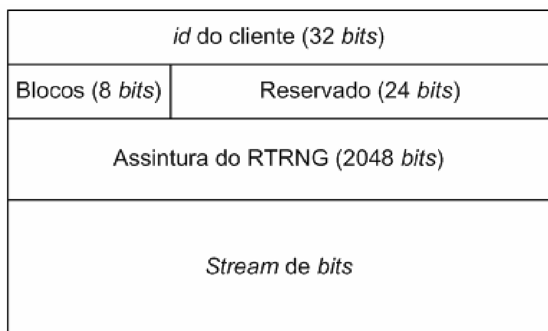


Fig. 7. Campo do *Stream* de *bits*

Onde:

> *id* do cliente: identificação do cliente;
> Bloco: quantidade dos blocos solicitados pelo cliente;
> Reservado: campo reservado para utilização futura;
> Assinatura do RTRNG: cabeçalho assinado provando a autenticidade do servidor;
> *Stream* de *bits*: quantidade de bits aleatórios solicitado.

Caso o cliente receba um pacote com uma assinatura inconsistente, ele irá descartar o pacote, visto que este pode ter advindo de uma fonte não confiável.

Após o início do recebimento do *Stream* de *bits*, o cliente irá utilizar parte deste para compor o seu número aleatório.

V. CONCLUSÕES

As baterias de testes realizadas (NIST e Diehard) mostraram que os números aleatórios gerados por linguagens não correlatos a segurança têm um grau de dispersão menor que os números gerados por uma fonte real de aleatoriedade (Alea I) ou ferramentas desenvolvidos para a segurança (OpenSSL).

Pôde ser observada que os geradores de números aleatórios produzidos por linguagens de uso geral, oferecem um grau de falha relativamente grande. O pior gerador analisado foi o gerador de números aleatórios da linguagem C/C++.

Somente as cadeias de números aleatórios geradas pela biblioteca OpenSSL e pelo *hardware* Alea I apresentaram valores considerados aceitáveis para a utilização em sistema de criptografia computacionais.

Aplicações envolvendo criptografia, especificamente aqueles algoritmos que necessitam de uma fonte de aleatoriedade para o desenvolvimento do mesmo (por exemplo o algoritmo RSA para a geração dos números primos), necessitam de uma atenção em especial neste quesito.

Para eliminar esta lacuna, foi proposta uma arquitetura para a geração de números aleatórios reais para suprir estas informações em sistemas móveis dedicados.

REFERÊNCIAS

[1] NEWMANN. J. von. **Comic Sections**. in: D. MacHale. Dublin. 1993.
[2] MENNA, L.M. **Análise de geradores de números pseudo-aleatórios**. 2005. 113 f. Tese de mestrado. Instituto Tecnológico de Aeronáutica. São José dos Campos: 2005.
[3] SCHNEIER, B. **Applied cryptography**. 2ed. New York: John Wiley & Sons, 1996.
[4] National Instituteof Standards and Technology. **A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random** Number Generators for Cryptographic Applications. NIST Special Publication 800-22rev1a. 2010. Disponivel em: <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf>. Acesso em: 20/06/10.
[5] MARSAGLIA. G. **The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness**. 1995. Disponível em: <http://www.stat.fsu.edu/pub/diehard/>. Acesso em: 20/06/10.
[6] BlackBerry developers. **BlackBerry Developer Zone**. Disponível em: < http://br.blackberry.com/developers/>. Acesso em: 20/06/10. Acesso em: 20/06/10.
[7] Araneus Alea I. **True Random Number Generator**. Disponível em: < http://www.araneus.fi/products-alea-eng.html>
. **NIST Statistical Test Suite**. Disponível em: <http://csrc.nist.gov/groups/ST/toolkit/rng/ documents/sts-2.1.zip>. Acesso em: 20/06/10.
[8] National Instituteof Standards and Technology. **NIST Statistical Test Suite**. Disponível em: <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/sts-2.1.zip>. Acesso em: 20/06/10.
[9] MARSAGLIA, G. TSANG, W.W.. Some Difficult-to-pass Tests of Randomness. In: COMPUTER SCIENCE AND STATISTICS SYMPOSIUM ON THE INTERFACE, 16, 1984, Atlanta. **Proceedings...** Atlanta: Elsivier Press, 1984. p. 11. Disponível em: <http://www.jstatsoft.org/v07/i03/paper>. Acesso em: 20/06/10.
[10] Trusted Computing Group. **TCG architecture overview**, version 1.4. Disponível em: <http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf>. Acesso em: 20/06/10.
[11] Trusted Computing Group. **TPM main: part 1 design principles (specification version 1.2 level 2 revision 103)**. Disponível em: < http://www.trustedcomputinggroup.org/files/resource_files/ACD19914-1D09-3519-ADA64741A1A15795/mainP1DPrev103.zip>. Acesso em: 20/06/10.
[12] Trusted Computing Group. **TPM main: part 2 tpm structure (specification version 1.2 level 2 revision 103)**. Disponível em: < http://www.trustedcomputinggroup.org/files/resource_files/8D3D6571-1D09-3519-AD22EA2911D4E9D0/mainP2Structrev103.pdf>. Acesso em: 20/06/10.
[13] Trusted Computing Group. **TPM main: part 3 commands (specification version 1.2 level 2 revision 103**. Disponível em: < http://www.trustedcomputinggroup.org/files/static_page_files/ACD28F6C-1D09-3519-AD210DC2597F1E4C/mainP3Commandsrev103.pdf>. Acesso em: 20/06/10.
[14] Kinney, S. L. **Trusted platform module basics**. Burlington: Newnes, 2006.
[15] SHIMANUKI, M.T. CUNHA, W.C. **Inicialização segura de sistemas computacionais dedicados**. XI SIGE. São José dos Campos. 2009.
[16] Balacheff, B., Chen, L., Pearson, S., Plaquin, D., Proudler, G. **Trusted computing platform**. New Jersey: Prentice Hall PTR, 2003.