SÍNDROME-FORTUNA: UMA ABORDAGEM VIÁVEL PARA A GERAÇÃO DE NÚMEROS PSEUDOALEATÓRIOS NO LINUX

DANIEL REZENDE SILVEIRA

SÍNDROME-FORTUNA: UMA ABORDAGEM VIÁVEL PARA A GERAÇÃO DE NÚMEROS PSEUDOALEATÓRIOS NO LINUX

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: JEROEN ANTONIUS MARIA VAN DE GRAAF

Belo Horizonte

Junho de 2010

Silveira, Daniel Rezende
S587s Síndrome-Fortuna: Uma abordagem viável para a geração de números pseudoaleatórios no Linux / Daniel Rezende Silveira. — Belo Horizonte, 2010
xxiv, 54 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Jeroen Antonius Maria van de Graaf

1. Sistemas Operacionais. 2. Criptografia. 3. Geração de Números Aleatórios. 4. NP-Completo. I. Título.

CDU 519.6*34

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Síndrome-Fortuna:Uma abordagem viável para a geração de números pseudoaleatórios no linux

DANIEL REZENDE SILVEIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. JEROEN ANTONIUS MARIA VAN DE GRAAF - Orientador
Departamento de Computação - UFOP

PROF. SÉRGIO VALE AGUIAR CAMPOS - Co-orientador
Departamento de Ciência da Computação - UFMG

Anderson C A Nascimento
PROF. ANDERSON CLAYTON ALVES NASCIMENTO
Departamento de Engenharia Elétrica - UNB

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

DR. WILTON SPEZIALI CALDAS
Desenvolvimento e Pesquisa - IVision

Belo Horizonte, 18 de junho de 2010.

Para Gislayne e Joaquim.

Agradecimentos

Agradeço a minha família pela compreensão e apoio; aos meus orientadores pela oportunidade de crescer e aprender muito; a Deus.

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."
(John von Neumann)

Abstract

Random numbers are used in various areas of computing, such as genetic programming, simulations and cryptography. In the latter, the random number generator takes a vital role, producing the initial secrecy for cryptographic protocols. This secret must be unknown to any adversary and will be used to ensure that the information remains secure. This work presents a random number generator based on the intractability of an NP-Complete problem, from the area of error-correcting codes, that use a non-heuristic approach for entropy collection. The generator, implemented in the Linux kernel, shows a good trade-off between eficiency and security and can be used as an alternative system interface for secure random number generation.

Keywords: Security, Cryptography, Random Number Generators, NP-Completeness, Entropy, Syndrome Decoding.

Resumo

Números aleatórios são utilizados em várias áreas da computação, como programação genética, simulações e criptografia. Nesta última, o gerador de números aleatórios exerce um papel fundamental, produzindo o segredo inicial para os protocolos criptográficos. Este segredo deve ser desconhecido de qualquer adversário e será utilizado para garantir que a informação criptografada permaneça segura. Este trabalho apresenta um gerador de números aleatórios baseado na intratabilidade de um problema NP-Completo, da teoria de códigos corretores de erros, que utiliza uma abordagem de coleta de entropia que dispensa heurísticas. O gerador, implementado no kernel do Linux, possui uma boa relação desempenho/segurança e pode ser utilizado como interface alternativa do sistema para a geração de números aleatórios seguros.

Palavras-chave: Segurança, Criptografia, Geradores de números aleatórios, NP-Completo, Entropia, Decodificação do Síndrome.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

Neste capítulo apresentaremos as motivações originárias para o estudo dos geradores de números aleatórios, o estado atual da pesquisa existente na área, os objetivos principais do trabalho e a organização do restante da dissertação.

1.1 Motivação

Os geradores de números aleatórios são a base de diversos sistemas criptográficos. Os valores gerados devem ser imprevisíveis por possíveis adversários e sua geração deve ser rápida e eficiente, para utilização em diversos cenários. Alguns exemplos da utilização de números aleatórios em sistemas criptográficos são os primos p e q no algoritmo RSA, a chave no algoritmo AES, a chave privada a no algoritmo DSA, os números de sequência no estabelecimento de conexões TCP, etc.

Em um cenário ideal, os números aleatórios utilizados por esses algoritmos seriam gerados por dispositivos capazes de medir fenômenos imprevisíveis, naturalmente caóticos, como as variações de pressão e temperatura do ambiente ou o decaimento do número atômico de uma substância radioativa. Entretanto, esse tipo de hardware não está disponível na maioria dos casos e, mesmo quando disponíveis, a geração dos valores é, quase sempre, muito lenta em relação à necessidade dos sistemas.

A solução adotada em diversos sistemas computacionais é a utilização de geradores de números pseudo-aleatórios – PRGs – que são algoritmos determinísticos que expandem rapidamente uma pequena quantidade de bits aleatórios, chamada semente, em uma longa sequência de valores indistinguíveis, dentro de certos limites, de valores verdadeiramente aleatórios. Para a produção da semente inicial, o sistema utiliza fontes de aleatoriedade real – chamadas fontes de entropia – como a temporização de interrupções de teclado, mouse e disco, ou mesmo algum hardware específico.

1.2 Contexto

A qualidade dos geradores de números aleatórios e sua aplicabilidade aos protocolos e aplicativos de segurança têm sido amplamente estudadas nos últimos anos.

Gutterman et al. [2006] fazem uma análise detalhada do gerador de números aleatórios do Linux, apontando falhas de segurança e problemas de implementação. Naquele trabalho são descritos também problemas relacionados à geração de números aleatórios em sistemas com baixa entropia. É criticada, por exemplo, a geração de chaves SSL na distribuição Linux OpenWRT utilizada em roteadores wireless. Devido à baixa aleatoriedade disponível nesse sistema, as chaves geradas são consideradas fracas e passíveis de serem quebradas em um ataque.

Uma discussão abrangente dos aspectos gerais dos PRGs, assim como um guia para o projeto e implementação de um PRG é proposto por Ferguson & Schneier [2003]. Aspectos relativos às fontes de entropia dos sistemas operacionais em dispositivos móveis são discutidos por Krhovjak et al. [2009].

Barak & Halevi [2005] apresentam uma definição rigorosa e a análise da segurança de geradores de números pseudoaleatórios. O trabalho sugere a separação do processo de extração de entropia do processo de geração dos números pseudo-aleatórios. É proposto um gerador que utiliza apenas uma entrada: a semente, que é o estado do gerador. Periodicamente a entropia é extraída dos eventos do sistema e é feita uma operação de XOR para alterar a semente do gerador. Essa construção é bem mais simples que a maioria dos PRGs existentes e ainda assim se mostra segura.

Francillon & Castelluccia [2007] abordam a geração de números pseudo-aleatórios criptograficamente seguros em sensores de redes sem fio. Nesse tipo de dispositivo a geração desses números se revela um problema, pois não existem as fontes de aleatoriedade utilizadas pela maioria dos PRGs. Naquele trabalho foi implementado um PRG, denominado TinyRNG, que utiliza os bits de erro de transmissão entre os sensores para gerar aleatoriedade. O trabalho demonstra que o erro de transmissão segue uma distribuição aleatória e que é um fator difícil de ser avaliado e manipulado por um invasor, sendo, portanto, considerada uma boa fonte de entropia.

1.3 Objetivos

O objetivo deste trabalho é implementar no núcleo do sistema Linux um gerador de números aleatórios baseado em uma arquitetura resistente a ataques, utilizando uma função geradora formalmente segura. Esse gerador deverá implementar uma interface não-blocante para fornecimento de bits aleatórios, com boas características de segu-

rança e desempenho satisfatório.

Para isso, utilizaremos a arquitetura do gerador Fortuna, proposto por Ferguson & Schneier [2003], cujo objetivo principal é ser resistente a diversos tipos de ataques conhecidos. Esse gerador utiliza o algoritmo AES, que, apesar de muito eficiente, não possui prova formal de segurança. Assim, vamos substituir esse algoritmo pela função geradora, também eficiente, baseada no problema NP-Completo da decodificação do vetor síndrome, descrita por Fischer & Stern [1996].

1.4 Organização da dissertação

Esta dissertação foi organizada em seis capítulos. O capítulo 2 apresenta os principais conceitos relacionados a geradores de números aleatórios e diversos tipos de geradores existentes, contextualizando o problema. O capítulo 3 apresenta a teoria utilizada na construção de geradores criptograficamente seguros. O capítulo 4 descreve a proposta do gerador Síndrome-Fortuna. O capítulo 5 revela detalhes da implementação e faz uma análise dos resultados obtidos. O capítulo 6 apresenta as conclusões obtidas e as sugestões de trabalhos futuros.

Capítulo 2

Geradores de números pseudoaleatórios

Como evidenciou o matemático von Neumann [1951] em seu trabalho sobre a geração de números aleatórios:

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

Não existem métodos aritméticos para geração de valores imprevisíveis, uma vez que estes presumem indeterminação e a aritmética é, essencialmente, determinística. Por isso, os geradores estudados neste trabalho são chamados *pseudo*aleatórios, indicando que os valores produzidos não são verdadeiramente indeterminados, mas possuem propriedades matemáticas que os permitem serem utilizados como tal, dentro de certos limites.

Na seção 2.1 vamos apresentar o conceito de aleatoriedade utilizado nos geradores de números pseudoaleatórios. Nas seções 2.2, 2.3 e 2.4 mostraremos alguns exemplos de cada tipo de gerador apresentando detalhes de projeto e implementação.

No restante desta dissertação iremos nos referir aos geradores de números pseudoaleatórios – *pseudorandom number generators* (PRGs) – apenas como geradores. Sempre que não estiver explícito o contrário, faremos referência a geradores criptograficamente seguros, isto é, projetados para serem utilizados em algoritmos criptográficos.

2.1 Conceito de Aleatoriedade

Os geradores de números pseudoaleatórios surgiram, inicialmente, para utilização em simulações e análise numérica. Estas aplicações demandavam dos geradores eficiência e,

principalmente, *uniformidade* nos valores produzidos. Com a evolução da criptografia surgiu a necessidade de criação de chaves secretas, que permitiriam usuários comunicar de forma segura. A partir de então, um outro requisito se tornou fundamental aos geradores de números pseudoaleatórios: a *imprevisibilidade* das sequências geradas.

A única forma de garantir que o segredo inicial de um gerador seja indeterminado é utilizando fontes de aleatoriedade reais. Essas fontes são parte importante na construção dos geradores e serão estudas nas seções 2.1.1 e 2.1.2.

2.1.1 Quantidade de Entropia

As fontes externas de aleatoriedade coletam dados de eventos imprevisíveis[1], como variações de pressão e temperatura, ruídos de ambiente e temporização das movimentações de mouse e de digitação.

Antes de utilizar a aleatoriedade coletada, entretanto, faz-se necessário *estimá-la*. O objetivo é evitar que o estado interno do gerador seja atualizado com valores potencialmente fáceis de serem descobertos por adversários. Em 1996 foi descoberta uma falha no gerador de números aleatórios do navegador *Netscape* que permitia que as chaves criadas e utilizadas nas conexões SSL fossem descobertas em cerca de um minuto. O problema, revelado no trabalho de Goldberg & Wagner, era a utilização do tempo do sistema e do *process id* como fontes de aleatoriedade. Mesmo quando o navegador utilizava chaves de sessão de 128 bits, consideradas seguras, estas possuiam no máximo 47 bits realmente aleatórios, muito pouco para evitar que um adversário utilizando força bruta conseguisse descobrir o valor da chave.

Entropia de Shannon Para permitir essa medição da aleatoriedade coletada das fontes externas, é utilizado o conceito de *entropia* – ou quantidade de incerteza – introduzido por Shannon [1948] em seu trabalho fundamental sobre a teoria das comunicações.

Shannon demonstrou que para uma fonte aleatória X que pode assumir n possíveis valores $\{x_i : i = 1, ..., n\}$, a quantidade de entropia – ou informação – fornecida $H(X)$ é:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

[1] O conceito de imprevisibilidade utilizado está relacionado à incapacidade humana de prever certos fenômenos, dada sua essência caótica ou quântica.

onde $p(x_i)$ é a função de probabilidade de massa do evento x_i e a base do logaritmo é a escala do sistema de numeração utilizado. Para representação em bits, $b = 2$.

Assim, conforme a probabilidade de ocorrência de seus eventos, cada fonte de aleatoriedade fornecerá uma quantidade de entropia diferenciada. Se considerarmos, por exemplo, um teclado de 105 teclas, no qual cada entrada é pressionada com igual probabilidade $p = \frac{1}{105}$, podemos dizer que cada chave fornecerá:

$$H(X) = -\sum_{i=1}^{105}(\frac{1}{105})\log_2(\frac{1}{105}) = -\log_2(\frac{1}{105}) = 6,71 \text{ bits de entropia}$$

Entretanto, sabemos que os teclados são utilizados para digitar textos em linguagens bem definidas, sendo que algumas letras, como as vogais, por exemplo, são bem mais comuns que outras. O exemplo citado só forneceria os 6,71 bits de entropia por tecla se o digitador estivesse vendado e se esforçando para não digitar teclas repeditas.

Min-Entropy Como a quantidade de entropia deriva da probabilidade de *cada* um dos eventos aleatórios, torna-se inviável realizar seu cálculo exato, uma vez que essas probabilidades podem ser difíceis de modelar ou mesmo desconhecidas. Assim, é utilizada uma estimativa de entropia, denominada *min-entropy*, que tem um cálculo simplificado, porém conservador:

$$H_{min}(X) = \min_{i=1}^{n}(-\log_b p(x_i)) = -\log_b(\max_{i=1}^{n} p(x_i))$$

Assim, a *min-entropy* é definida com base no evento de maior probabilidade e, portanto, será sempre menor ou igual à entropia de Shannon, que toma como base uma média ponderada de todas as probabilidades. No exemplo anterior, se uma das teclas tivesse uma probabilidade de ocorrência superior às demais $p(x_i) = 50\%$, teríamos:

$$H_{min}(X) = -\log_2(\frac{1}{2}) = 1 \text{ bit de entropia}$$

O objetivo ao utilizar a *min-entropy* é estabelecer um piso mínimo de entropia. No exemplo acima, como uma das teclas tinha probabilidade muito superior às demais, a estimativa de entropia foi reduzida à equivalente ao sorteio de uma moeda, no qual só há dois eventos, cada um com probabilidade de 50%.

A estimativa de entropia é um dos pontos críticos no projeto de geradores de números aleatórios, pois o nível de segurança destes está relacionado diretamente à precisão da estimativa realizada. Como veremos adiante, a abordagem de coleta e utilização de entropia proposta por Ferguson & Schneier [2003], e adaptada neste trabalho, procura minimizar o problema de uma possível imprecisão na estimativa de entropia.

2.1.2 Fontes de Entropia nos Sistemas Computacionais

Atualmente não existe uma lista definitiva de fontes de aleatoriedade seguras para os diversos cenários possíveis, apesar de haver esforço neste sentido [Barker & Kelsey, 2006]. Em regra, cada sistema deve utilizar as fontes de ruído existentes em seu contexto e deve protegê-las de forma que possíveis adversários não possam prever ou medir suas saídas.

Apresentaremos nesta seção algumas fontes de entropia e suas formas de operação e extração de bits.

Interrupções de Hardware Esta é uma das fontes de aleatoriedade mais comuns em sistemas interativos como computadores pessoais, dispositivos móveis, servidores, etc. A coleta de entropia geralmente se baseia em um relógio de tempo real de alta precisão, que é medido a cada interrupção sofrida pelo sistema operacional. As interrupções podem ser de teclado, mouse, rede ou outros dispositivos que estejam conectados ao sistema. Além das temporizações, são utilizados como fonte de ruído as informações da interrupção propriamente dita, como, por exemplo, a tecla pressionada, o cabeçalho de um pacote de rede ou as coordenadas do mouse. A utilização dessas informações pode levar a problemas de segurança, como mostrou Gutterman et al. [2006], pois, caso um adversário consiga quebrar o gerador, ele poderá obter informações sensíveis do usuário.

A quantidade de entropia obtida nesses eventos é algo muito difícil de modelar e estimar. Algumas pessoas podem manter o tempo de digitação entre teclas praticamente constante, variando poucos milissegundos, ou os sistemas podem ficar ociosos por horas, sem nenhuma tecla pressionada. A frequência de varredura do teclado também limita as medições de tempo, reduzindo a entropia coletada. Caso o sistema operacional esteja operando sobre uma plataforma virtual, as interrupções de hardware podem ser acumuladas em um buffer antes de serem entregues ao sistema hóspede, reduzindo, também, a entropia fornecida. Além disso, existe a possibilidade de o adversário obter informações adicionais sobre a

fonte, coletando com um microfone, por exemplo, os sons do teclado. Todas essas restrições sugerem muita cautela na estimativa de entropia fornecida por fontes desse tipo.

Conversão analógico-digital Qualquer conversão de sinais analógicos para strings de bits impõe erros que podem ser utilizados como fontes de entropia. Conversores analógico-digitais capturam formas de onda, quantificam e digitalizam o sinal em seqüências onde grupos de bits representam a força do sinal em um momento discreto do tempo, como as sequências do tipo PCM - Pulse Code Modulation. Devido à natureza dos componentes analógicos que capturam os sinais, os bits menos significativos da sequencia digital tendem a carregar erros de medição que podem servir como fontes de entropia. Esses conversores podem ser construídos a partir de microfones ou componentes de entrada de placas de som, sintonizadores de TV e rádio.

Na prática, esse tipo de gerador é difícil de ser encontrado em desktops e servidores, sendo mais apropriado para dispositivos embarcados onde não existem outras alternativas para coletar entropia. Krhovjak et al. [2009] demonstram em seu trabalho métodos para extração de bits utilizando conversores analógico-digitais em aparelhos móveis.

Free-running Oscillators Esse tipo de fonte é muito utilizada na construção de dispositivos de hardware para geração de bits aleatórios. Baseia-se na comparação de estados de circuitos que oscilam de forma livre, independente e sujeita a ruídos térmicos ou ruídos tipo "shot"[2]. Um tipo comum de oscilador é o *Ring Oscillator*, que é um conjunto ímpar de portas tipo NOT conectadas em cadeia que oscilam livremente de acordo com a freqüência e os ruídos presentes no circuito.

Pode-se encontrar geradores baseados em *ring oscillators* nos processadores da Via™e em alguns chipsets antigos da Intel™. Apesar da propriedades quânticas envolvidas na oscilação dos circuitos, este tipo de fonte não está imune a ataques, conforme demonstram Markettos & Moore [2009]. Os autores conseguiram reduzir drasticamente a entropia fornecida por um gerador baseado em osciladores por meio da injeção de freqüência no circuito. Além disso, o adversário pode conseguir capturar os bits de entropia fornecidos pela fonte, por meio de uma falha de proteção de memória, por exemplo. Nesse caso, os dados continuam perfeitamente aleatórios, mas não contêm nenhuma entropia do ponto de vista do adversário.

[2]Ruído gerado devido à natureza quântica da passagem de elétrons em junções de materiais

Além das dificuldades de coletar dados realmente aleatórios, existem diversos outros problemas práticos na implementação e utilização de fontes de aleatoriedade. Em primeiro lugar, as fontes podem não estar disponíveis todo o tempo. Se o gerador tem que esperar pela digitação de alguma tecla ou a movimentação do mouse, então os aplicativos que dependem do gerador podem parar. Por isso, os PRGs devem sempre fazer uso do maior número de fontes possíveis, evitando depender de uma ou outra exclusivamente.

É interessante observar que existem geradores chamados *True-Random Number Generators*[3] (TRNGs), construídos inteiramente a partir de dispositivos físicos. Tais geradores não utilizam técnicas de expansão dos dados aleatórios, e produzem sua saída apenas com os valores coletados das fontes de entropia. Quando disponíveis, são frequentemente utilizados para formação do segredo inicial de outros PRGs.

As principais características dos TRNGs são a baixa eficiência, devido à baixa velocidade de geração de aleatoriedade das fontes externas; o não-determinismo, ou seja, os valores gerados são, em essência, indeterminados; e a ausência de ciclos, ou seja, as seqüências geradas não podem se repetir.

2.2 Geradores não-criptográficos

Os geradores apresentados a seguir foram desenvolvidos para utilização em análise numérica, simulações e jogos. Estão presentes em diversos softwares de uso geral e são implementados, também, como bibliotecas das linguagens de programação. Esses geradores não são robustos o suficiente para serem utilizados em aplicações criptográficas, uma vez que um adversário pode utilizar os valores produzidos para estimar, ou mesmo prever, os valores que serão gerados no futuros.

2.2.1 Linear congruential generator

Os geradores lineares são os mais utilizados em aplicações não criptográficas. A saída é produzida da seguinte forma:

$$X_{i+1} = aX_i + c \pmod{n}$$

onde a, c e n são constantes e X_0 é a semente do gerador. A seleção desses valores deve ser cuidadosa para garantir o período máximo $n - 1$. Se os valores forem, por exemplo, $a = 1$ e $c = 0$, a saída será uma sequência repetida de valores X_0.

[3]Também chamados de *Non-deterministic Random Bit Generators* ou *Hardware Random Number Generators*

2.2.2 Linear feedback shift register

O *linear feedback shift register* (LFSR) é composto de um registrador com n elementos, cada um capaz de armazenar um bit, sendo a_n o elemento de entrada e a_0 o elemento de saída. Um relógio controla a operação de *shift* que gera um fluxo dos dados entre os elementos. A cada ciclo do relógio uma operação de *feedback* é realizada.

A cada unidade de tempo o conteúdo do elemento a_0 é colocado na saída, o conteúdo de cada elemento i é movido para o elemento $i-1$ e o novo conteúdo do elemento $n-1$ é a saída da função *feedback*.

A função *feedback* é uma operação XOR do conteúdo de um subconjunto dos elementos do registrador. Esse subconjunto, também chamado de sequência *tap*, é representado por um polinômio na forma $\sum c_i x^i + 1$ onde $i \leq i \leq n$ e $c_i \in \{0, 1\}$, sendo que $c_i = 1$ representa que o i-ésimo elemento está na sequência *tap*

A figura 2.1 mostra um gerador com $n = 5$ e o polinômio $x^4 + x^3 + x^2 + 1$.

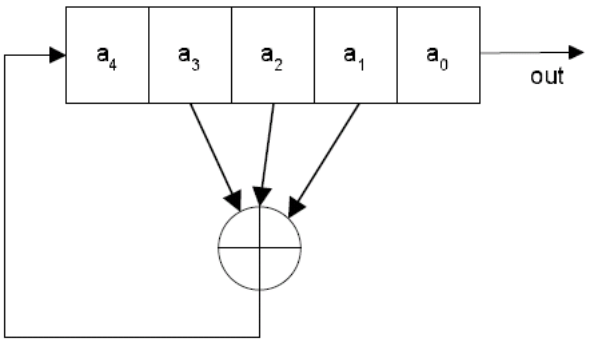

Figura 2.1. Linear Feedback Shift Register.

Se o primeiro elemento do registrador é utilizado na entrada da função *feedback* (ou seja, $c_0 = 1$), o gerador é chamado *não-singular*. Se o LFSR é não-singular, o polinômio é primitivo e o conteúdo inicial do registrador é diferente de zero, então o gerador produz a saída com o período máximo possível $2^n - 1$ [Vanstone et al., 1997, p. 195].

Conforme mostram Ferguson & Schneier [2003], vários geradores podem ser construídos a partir de combinações de LFSRs.

2.2.3 Generalized feedback shift register

O *generalized linear feedback shift register* (GFSR) é um refinamento do LFSR. Particularmente, o GFSR é não-singular, tem polinômio primitivo e seu grau é 3. As vantagens desse tipo de gerador são uma melhor distribuição dos valores, maior período e maior eficiência.

Definição Uma série $a_i \in \{0,1\}^w, i \in \mathbb{N}$ é gerada por um GFSR de polinômio x^p+x^q+1 se e somente se

$$a_i = a_{i-p+q} \oplus a_{i-p}\,, \quad i = p, p+1, ...$$

Essa definição é similar ao LFSR, exceto pelas seguintes diferenças:

- O elemento de memória no GFSR é uma palavra com w bits e no LFSR é apenas um bit.
- O período da sequência de saída do GFSR depende da semente inicial. O mecanismo para criar a semente de forma a maximizar o ciclo é descrito em detalhes por Lewis & Payne [1973].

A figura 2.2 ilustra um GFSR.

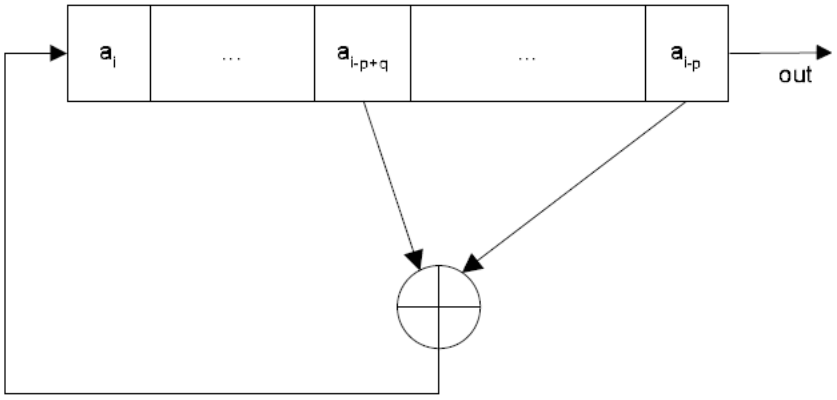

Figura 2.2. Generalized Feedback Shift Register.

2.2.4 Twisted generalized feedback shift register

O *twisted generalized feedback shift register* (TGFSR) é uma melhoria ao GFSR. Neste gerador o ciclo aumenta de $2^p - 1$ para $2^{pw} - 1$, e a dependência da semente inicial é eliminada. Além disso, o polinômio pode ser de qualquer grau.

Definição Uma série $a_i \in \{0,1\}^w, i \in \mathbb{N}$ e uma matriz $A_{w\times w}$ é gerada por um TGFSR de polinômio $x^p + x^q + 1$ se e somente se

$$a_i = a_{i-p+q} \oplus a_{i-p} \cdot A\,, \quad i = p, p+1, ...$$

O produto $a_{i-p} \cdot A$ pode ser implementado com operações *shift-right* e *XOR*

Uma variação muito usada do TGFSR é o *Mersenne Twister* [Matsumoto & Nishimura, 1998], que tem período de $2^{19937} - 1$ e é bastante eficiente. Este gerador é utilizado como o gerador de números aleatórios da linguagem de programação *python*

A figura 2.3 ilustra um TGFSR.

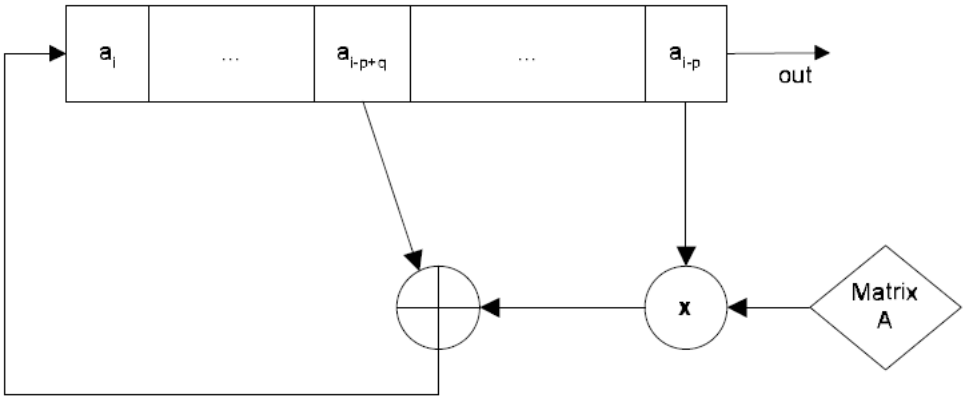

Figura 2.3. Twisted Generalized Feedback Shift Register.

2.3 Geradores criptográficos

Os geradores criptográficos[4] são projetados para que quaisquer adversários, com poderes computacionais limitados, não possam diferenciar os valores produzidos de valores verdadeiramente aleatórios. Para atingir esse objetivo, os geradores devem produzir sequências que sejam uniformes e, além disso, não contribuam para a descoberta de valores futuros.

Os geradores criptográficos podem ser divididos em duas categorias: os baseados em primitivas criptográficas e os baseados na intratabilidade de problemas computacionais. Essa distinção se faz necessária, devido às diferenças na eficiência e na formalidade das provas de segurança dos geradores.

Vamos apresentar três geradores. Os dois primeiros baseados na dificuldade de se fatorar números inteiros e o último baseado em um primitiva criptográfica.

2.3.1 Shamir

Shamir [1981] foi o primeiro a publicar um PRG e provar, com base na dificuldade de fatoração, que a saída gerada é *imprevisível* por um adversário qualquer. O gerador

[4]Também chamados de *Cryptographically Secure Pseudo-Random Number Generators*

proposto utiliza os princípios da criptografia de chaves públicas RSA e, como utiliza exponenciação modular de números grandes, sua eficiência é reduzida.

Os parâmetros do gerador são $N = p \cdot q$, onde p e q são números primos grandes e secretos, uma semente S e uma sequência de chaves $K_1, K_2, ...$, tal que todo K_i é primo relativo a $\varphi(N) = (p-1) \cdot (q-1)$. A sequência de chaves K_i não é determinante na segurança do esquema e pode ser formada pelos primos ímpares (por exemplo, $3, 5, 7, 11...$). O algoritmo 1 mostra o funcionamento do gerador.

[Algoritmo] 1 Gerador Shamir.

Entrada: Dois primos grandes p e q. Uma semente S em $[1, pq-1]$. Uma sequência K_i, tal que para todo K_i, $gcd(K_i, pq) = 1$
Saída: Imprime seqüência de bits aleatórios

1: $N \leftarrow p \cdot q$
2: **para** $i = 1$ até ∞ **faça**
3: $\quad x \leftarrow S^{1/K_i} \pmod N$
4: $\quad$ **imprima** x
5: **fim para**

O cálculo da raiz $K_{\text{i-ésima}}$ de S é simples, dados p e q, e difícil, caso estes valores sejam desconhecidos. Shamir prova que a computação de R_1 dados N, S e $R_2, ...R_l$ é tão difícil quanto computar R_1 dados somente N e S. Isto significa que conhecer partes da sequência de valores não contribui para o conhecimento de outras partes. A segurança do esquema baseia-se na dificuldade de calcular a raiz modular de grandes valores e é equivalente à segurança do RSA.

2.3.2 Blum-Blum-Shub

Blum et al. [1983] propuseram um gerador criptográfico denominado Blum-Blum-Shub (BBS). O gerador tem um design simplificado em relação ao apresentado por Shamir e, por isso, seu desempenho é consideravelmente superior.

Os parâmetros do BBS são $N = p \cdot q$, onde p e q são números primos grandes e secretos tais que $p \equiv q \equiv 3 \pmod 4$. A semente do gerador BBS S_0 é formada a partir de um número s, primo relativo a N, da seguinte forma $S_0 = s^2 \pmod N$.

A sequência de saída é produzida utilizando os k bits menos signficativos de $S_i = S_{i-1}^2 \pmod N$, onde $k \leq \log_2 \log_2(S_i)$. Tipicamente $k = 1$, ou seja, apenas o bit menos significativo é utilizado. O algoritmo 2 descreve o gerador.

A segurança do BBS, assim como no gerador proposto por Shamir, está na dificuldade de fatorar N [Vazirani & Vazirani, 1984].

[Algoritmo] 2 Gerador Blum-Blum-Shub.

Entrada: Dois primos grandes p e q, tais que $p = q = 3 \pmod 4$. Um segredo aleatório s em $[1, pq - 1]$ tal que $gcd(s, pq) = 1$
Saída: Imprime seqüência de bits aleatórios

1: $N \leftarrow p \cdot q$
2: $x_0 \leftarrow s^2 \pmod N$
3: **para** $i = 1$ até ∞ **faça**
4: $\quad x_i \leftarrow x_{i-1}^2 \pmod N$
5: $\quad$ **imprima** Bit menos significativo de x_i
6: **fim para**

2.3.3 ANSI X9.17

O algoritmo a seguir é um padrão americano de processamento de informação (Federal Information Processing Standard – FIPS) para geração de chaves e vetores de inicialização, mas é frequentemente usado como um PRG de propósito geral. O algoritmo foi proposto pelo American National Standards Institute (ANSI) em 1985 e utiliza o método criptográfico 3DES.

Os parâmetros do gerador são uma semente s, um inteiro m e uma chave DES-EDE k. O algoritmo 3 mostra um pseudocódigo do gerador.

[Algoritmo] 3 Gerador ANSI X9.17.

Entrada: Uma semente secreta s de 64 bits, um inteiro m, uma chave DES-EDE k.
Saída: Imprime seqüência de bits aleatórios

1: $I \leftarrow E_k(D)$, onde D é uma representação da data/hora do sistema com 64 bits
2: **loop**
3: $\quad out \leftarrow E_k(I \oplus seed)$
4: $\quad seed \leftarrow E_k(I \oplus out)$
5: $\quad$ **imprima** out
6: **fim loop**

A segurança do gerador está na dificuldade de inverter a cifra 3DES E_k, sobre a qual não há prova formal de unidirecionalidade.

2.4 Geradores criptográficos com atualização de entropia - Linux

Conforme vimos, todo gerador criptográfico faz uso de um estado inicial não-determinístico e secreto para produzir sua sequência de valores imprevisíveis. Não faz parte da definição desses geradores, entretanto, a forma como esse valor será pro-

duzido, uma vez que a geração dessa semente requer acesso a dispositivos físicos reais, capazes de produzir entropia.

Essa tarefa recai, em geral, sobre os sistemas operacionais, devido ao controle que esses têm sobre os *drivers* de dispositivos e outros eventos que podem contribuir com aleatoriedade real. Para fornecer dados aleatórios de forma segura, os sistemas operacionais modernos implementam complexos geradores de números aleatórios, que possuem um sistema próprio de coleta, armazenamento e utilização de entropia.

Vamos descrever a seguir o gerador de números aleatórios do kernel Linux versão 2.6.30 (LRNG), que foi estudado em detalhes neste trabalho, para viabilizar a implementação do gerador Síndrome-Fortuna, apresentado no capítulo 4.

2.4.1 Algoritmo

O gerador é composto de três blocos de dados aleatórios, denominados *pools*, um primário e dois secundários. O primeiro, denominado *input_pool*, tem 4096 bits e é responsável pela coleta da entropia do sistema e o fornecimento desta aos pools secundários. Estes, com 1024 bits cada, chamados *blocking_pool* e *nonblocking_pool* funcionam como sementes do algoritmo gerador, um para geração blocante, determinada por uma estimativa de entropia, e o outro para geração não-blocante. A figura 2.4 ilustra o funcionamento básico do sistema.

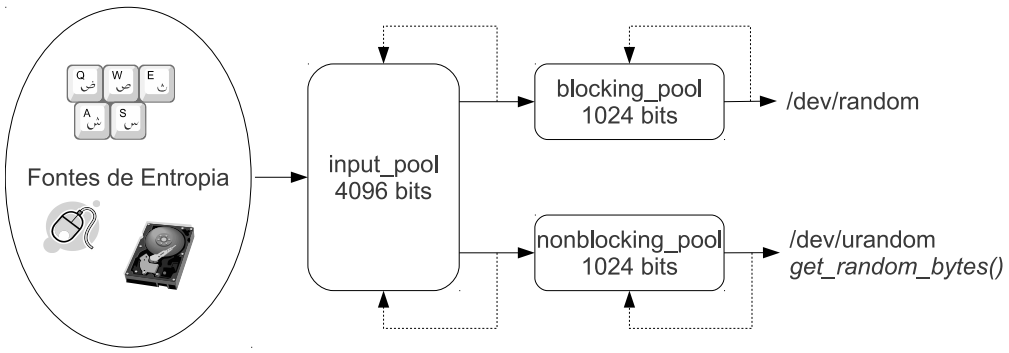

Figura 2.4. Funcionamento do gerador do Linux 2.6.30. As setas contínuas indicam a extração de bits dos pools e as setas pontilhadas indicam o processo de realimentação. A funções de extração e realimentação são implementadas por uma uma variação de TGFSR em conjunto com o SHA-1.

São fornecidas duas interfaces visíveis aos usuários do sistema operacional, na forma de drivers de dispositivos: */dev/random* e */dev/urandom*. Ambas são implementadas pelo mesmo algoritmo, utilizando, respectivamente, os pools *blocking_pool* e *nonblocking_pool*. A principal diferença entre os dispositivos é que o primeiro utiliza um estimador de entropia e só fornece bits até a quantidade de entropia capturada.

A segunda interface, */dev/urandom*, fornece tantos bits quantos solicitados, iterando sobre o pool não blocante. Existe ainda uma terceira interface, acessível apenas para funções internas do kernel, chamada *get_random_bytes()*, implementada sobre o pool não blocante.

O algoritmo gerador pode ser descrito, em linhas gerais, com quatro funções, que controlam a adição de entropia no pool primário e a transferência desta para os pools secundários e para a saída. Estas funções serão apresentadas a seguir, com pequenas simplificações para facilitar o entendimento.

2.4.1.1 Funções de controle

A primeira função, denominada *add_timer_randomness*, apresentada no algoritmo 4, controla a entrada de dados no gerador, sendo responsável pela adição de entropia ao *input_pool*. Essa função é chamada a cada interrupção de disco, de entrada (*input core*) ou de drivers externos, configurados para fornecer entropia (precisam ativar o flag IRQF_SAMPLE_RANDOM). Seu funcionamento é bastante simplificado, uma vez que ela é chamada durante o contexto da interrupção.

A função constrói uma estrutura *sample*, utilizando a variável *jiffies*, que conta o número de interrupções de relógio do sistema, a saída da função *get_cycles()* que retorna o número de ciclos da CPU e a variável *num*, que contém informações sobre a interrupção, tecla pressionada ou operação de disco. Essa estrutura é utilizada para estimar a entropia adicionada (vide 2.4.2), creditá-la ao *input_pool* e realizar a operação de embaralhamento dos dados no pool primário do sistema. Caso a entropia do pool primário esteja acima de um limiar definido de 3584 bits, somente 1 evento é processado, a cada 4096 chamadas, diminuindo a utilização de CPU.

[Algoritmo] 4 Linux *add_timer_randomness*.

Entrada: *state*: estrutura com temporizações anteriores do dispositivo;
num: dado relativo ao evento ocorrido;
TRICKLE_THRESHOLD: limiar para descartar eventos.

1: **se** *input_pool.entropy* > TRICKLE_THRESHOLD **então**
2: Prossiga somente 1 em cada 4096 chamadas
3: **fim se**
4: *sample* ← struct(*jiffies*, *get_cycles()*, *num*)
5: *entropy* ← estimate_entropy(*state*, *sample*)
6: credit_entropy_bits(*input_pool*, *entropy*)
7: mix_pool_bytes(*input_pool*, *sample*)

A função principal do gerador, que controla a extração de bits dos pools, tanto do primário quanto dos secundários, é chamada *extract_entropy*. Cada interface, blocante

ou não blocante, utiliza de forma diferente essa função para produzir sua saída. Na implementação da interface */dev/random*, por exemplo, verifica-se primeiro a quantidade de entropia disponível antes de se começar a extrair os dados. Caso não se consiga extrair a quantidade solicitada, coloca-se o processo em espera, até que a estimativa de entropia seja acrescida. A interface */dev/urandom* e a função *get_random_bytes* utilizam diretamente a função *extract_entropy*, que está descrita no algoritmo 5.

[Algoritmo] 5 Linux *extract_entropy*.

Entrada: *nbytes*: número de bytes solicitados;
pool: pool de onde os dados serão extraídos;
POOL_SIZE: tamanho dos pools secundários em bytes (128);
EXTRACT_SIZE: tamanho da porção extraída por vez em bytes (10).
Saída: *out*: ponteiro para área de memória que receberá *nbytes* bytes aleatórios

1: **se** (*pool* é secundário) E (*pool.entropy* < *nbytes*) E (*pool.entropy* < POOL_SIZE) **então**
2: *bytes* ← min(*nbytes*, POOL_SIZE)
3: *tmp* ←extract_entropy(*input_pool*, *bytes*)
4: mix_pool_bytes(*pool*, *tmp*)
5: credit_entropy_bits(*pool*, *bytes*)
6: **fim se**
7: **para** $i = 0$ até *nbytes* **faça**
8: *out*[*i*] ← extract_buf(*pool*)
9: i += $EXTRACT\_SIZE$
10: **fim para**
11: debit_entropy_bits(*pool*, *nbytes*)

Além de produzir a saída do gerador, a função *extract_entropy* controla a transferência de entropia entre o pool primário e os secundários e faz o controle de crédito e débito de entropia entre os pools. A cada chamada, a função verifica se a quantidade de dados solicitados é maior que a entropia existente no pool (blocking ou nonblocking) e, se o pool não estiver com a entropia máxima possível (POOL_SIZE), o algoritmo faz um "reseed" buscando dados do pool primário. Após este reseed, a função produz a saída, extraindo porções de 10 bytes do pool solicitado.

As funções de extração *extract_buf* e de adição *mix_pool_bytes* são utilizadas no processo de transferência da entropia. As funções *credit_entropy_bits* e *debit_entropy_bits* fazem o crédito e débito de entropia nos pools em questão. Estas funções não serão descritas neste trabalho, por serem auxiliares à operação do gerador e sem impacto no entendimento global de seu funcionamento.

2.4.1.2 Extração de entropia

A função de extração é implementada com um algoritmo hash SHA-1, associado a um processo de *feedback*, para garantir a segurança dos estados anteriores (*forward security*). Para facilitar o entendimento, dividimos o algoritmo em quatro etapas, ilustradas na figura 2.5. Na primeira etapa é calculado o hash SHA-1 sobre o conteúdo integral do pool. Após isto o valor do hash é adicionado ao pool em uma operação de *feedback*. Na terceira etapa o hash é calculado novamente, desta vez sobre 64 bytes do pool, sem reiniciar o contexto da função, com o objetivo de produzir uma saída diferente do valor de feedback. Na quarta e última etapa o algoritmo realiza um *folding* dos dados, utilizando a saída do hash de 160 bits, para formar uma saída de 80 bits.

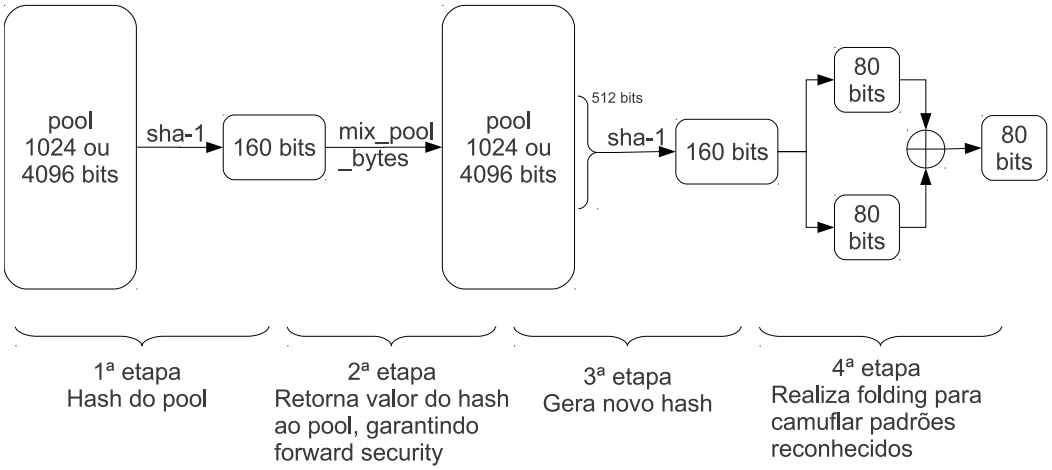

Figura 2.5. Extração de entropia de um pool primário (4096 bits) ou secundário (1024 bits) para um buffer de saída de 80 bits. Operação dividida em 4 etapas, conforme implementado no kernel Linux 2.6.30.

2.4.1.3 Adição de entropia

Toda entrada de entropia, em qualquer dos pools, é realizada pela função *mix_pool_bytes*. Esta função utiliza uma variação de um TGFSR com polinômios primitivos $x^{103}+x^{76}+x^{51}+x^{25}+x^{1}+1$ para o pool primário e $x^{26}+x^{20}+x^{14}+x^{7}+x^{1}+1$ para os pools secundários. A função utilizada não é unidirecional, entretanto, esta característica não prejudica o funcionamento do gerador, uma vez que as saídas desta função nunca são enviadas ao usuário externo sem antes passar pela função unidirecional de extração *extract_buf*. A figura 2.6 ilustra o funcionamento.

Conforme pode ser observado na figura 2.6, cada byte da entrada é suficiente para atualizar 4 bytes do pool. Assim, o pool primário, que tem 128 bytes de tamanho,

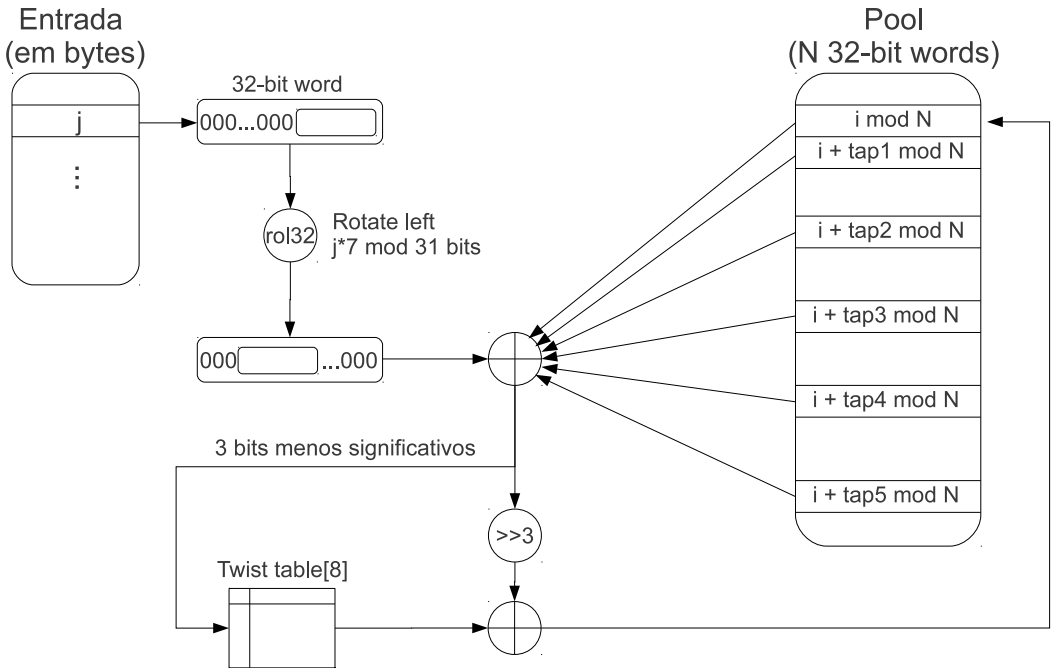

Figura 2.6. Adição de entropia em um pool do gerador. Cada byte j é transformado em uma palavra de 32 bits e rotacionado *j*7 mod 31* bits à esquerda. Após isto, esta *word* é "cifrada" pelo resultado do ou exclusivo bit a bit das posições do pool correspondentes aos "taps" do polinômio primitivo. O resultado segue, então, o procedimento usual de um TGFSR, sendo os 3 bits menos significativos utilizados para indexar um elemento da *twist-table* e os 29 restantes fazem ou exclusivo bit a bit com este elemento. O resultado final desta operação é colocado na posição i do pool.

precisa de 32 bytes de entrada para ser completamente renovado. Cada chamada da função *add_timer_randomness*, conforme vimos, monta uma estrutura com 16 bytes de tamanho em arquiteturas de 32 bits e 20 bytes em arquiteturas de 64 bits. Ou seja, bastam 2 eventos para que todos os dados do pool primário sejam atualizados, mesmo que estas operações acrescentem pouca ou nenhuma entropia ao pool.

2.4.2 Estimativa de Entropia

Uma estimativa da entropia é calculada para cada evento adicionado ao *input_pool*. O somatório da entropia de cada evento compõe uma estimativa global de entropia para o pool primário do sistema. Esta estimativa é utilizada, então, no processo de transferência de dados entre o pool primário e os secundários, para garantir atualizações com quantidades de aleatoriedade suficientes para impedir que adversários enumerem os estados possíveis. Este tipo de atualização é chamada *atualização catastrófica*. A estimativa é utilizada, também, para bloquear a interface */dev/random*, quando o

usuário requisitar mais dados aleatórios que o gerador tiver acumulado.

A estimativa da entropia de cada evento é realizada na função *add_timer_randomness*, utilizando as diferenças entre a temporização do evento no momento atual n e no instante anterior $n-1$. São utilizados os deltas de primeira, segunda e terceira ordem das temporizações dos eventos, conforme demonstrado a seguir:

$$\begin{aligned} t_n &\leftarrow jiffies \\ \delta_n &\leftarrow t_n - t_{n-1} \\ \delta_n^2 &\leftarrow \delta_n - \delta_{n-1} \\ \delta_n^3 &\leftarrow \delta_n^2 - \delta_{n-1}^2 \\ mindelta &\leftarrow min(|\delta_n|, |\delta_n^2|, |\delta_n^3|) >> 1 \\ entropy &\leftarrow min(\log_2(mindelta), 11) \end{aligned} \tag{2.1}$$

A estratégia utilizada pelo estimador de entropia leva em consideração que os eventos podem ocorrer com temporização aproximada por polinômios de grau 0, 1 ou 2, e, por isso, serem previsíveis a adversários. Caso esta hipótese seja verdadeira, um adversário poderia prever as temporizações utilizando o próprio modelo polinomial de baixo grau, fazendo com que a entropia adicionada ao sistema fosse nula.

Para evitar este tipo de ataque, possível quando as fontes fornecem eventos com temporizações polinomiais, o estimador de entropia considera apenas valores de temporização que *não* se encaixam nos polinômios, apresentando diferenças, ou deltas, em relação ao modelo considerado. É utilizado o menor delta, considerando os três graus de polinômio utilizados. Este valor é, ainda, reduzido em 1 bit e é calculado seu logaritmo, que não pode ser maior que 11.

2.4.3 Inicialização do gerador

A inicialização do gerador no kernel Linux 2.6.30 é realizada por meio de duas chamadas à função *mix_pool_bytes*. A primeira passa como parâmetro a hora do sistema, com resolução de 8 bytes, fornecida pela função *ktime_get_real()*. A segunda chamada passa como parâmetro a estrutura *utsname* com 325 bytes de tamanho, contendo o nome do sistema operacional, a versão, o nome dado à instalação e outras informações. Conforme vimos, a quantidade de informação passada a função *mix_pool_bytes* é suficiente para modificar toda a extensão dos pools, embora a entropia real adicionada seja consideravelmente menor.

Para evitar o problema da inicialização com baixa entropia, o sistema fornece,

nos dispositivos */dev/random* e */dev/urandom*, uma interface para se escrever dados aleatórios nos pools. Assim, os desenvolvedores do kernel sugerem, no comentário inicial do código do gerador, que antes de se desligar o sistema, sejam capturados 512 bytes de dados aleatórios da interface */dev/urandom* para um arquivo-semente em disco, e que estes dados sejam escritos no gerador após a inicialização, simulando continuidade na operação do gerador.

Esse arquivo-semente deve ser recriado a cada inicialização bem-sucedida do sistema, evitando que, em caso de desligamento súbito, o sistema retorne a um estado anterior já utilizado. Esta estratégia é sugerida por Ferguson & Schneier [2003] e está descrita nos comentários iniciais do código fonte do gerador aleatório do Linux. Cabe ressaltar que o kernel, por si próprio, não toma este cuidado, fornecendo apenas as interfaces para que os desenvolvedores das distribuições o façam.

Em todas as versões de Linux para desktops que temos conhecimento esta solução é implementada por meio de scripts de inicialização, mesmo que de forma temerária, como sugere o comentário encontrado na linha 48 do script */etc/init.d/urandom* do Ubuntu 10.04: *"# Hm, why is the saved pool recreated at boot? [pere 20090903]"*. Apesar do questionamento, o script recria o pool corretamente, após a inicialização, evitando o que poderia representar uma falha grave de segurança para o gerador.

Existe, ainda, a situação em que o arquivo-semente não está disponível, como na primeira inicialização de um computador recém fabricado, ou em um sistema sendo executado a partir de um *Live CD*[5]. Nestes casos não existe estado anterior, gravado em disco, e o sistema deve buscar outras formas de entropia para permitir uma inicialização segura. As soluções neste cenário, dependem da plataforma do sistema em questão.

Um dispositivo móvel, por exemplo, poderia utilizar as taxas de erros de transmissão de sinais, conforme mostrado por Francillon & Castelluccia [2007] ou outros meios, como entradas de som e vídeo [Krhovjak et al., 2009]. Já um sistema Live CD poderia solicitar alguma interação do usuário, ou um arquivo de dados externo qualquer, um CD de música. Estas alternativas inserem um sério contratempo à praticidade desses sistemas, mas permitir a inicialização somente com a informação de temporização não parece aconselhável. Isto por que, no caso do Live CD, as informações da estrutura *utsname* são plenamente previsíveis e, por isso, contém nenhuma entropia.

[5]Live CDs são distribuições completas e funcionais que cabem integralmente em um CD-ROM

Capítulo 3

Construção de um gerador criptograficamente seguro

A construção de um gerador de números aleatórios criptograficamente seguro requer atenção para algumas propriedades importantes, que visam evitar ataques de diversos tipos. Essas propriedades foram estabelecidas no trabalho de Kelsey et al. [1998] e, mais recentemente, formalizadas por Barak & Halevi [2005] em um modelo e uma arquitetura para a construção de PRGs criptograficamente robustos. Ferguson & Schneier [2003] também discutem modelos de ataques a PRGs e as correspondentes estratégias de defesa.

3.1 Função Unidirecional

Intuitivamente, uma função f é unidirecional se ela é simples de computar mas difícil de inverter, isto é, dado x, o valor de $f(x)$ pode ser computado em tempo polinomial mas qualquer algoritmo viável que receber como entrada $f(x)$ pode gerar uma saída y tal que $f(y) = f(x)$ apenas com probabilidade negligenciável.

A existência de funções unidirecionais, ou *one-way functions*, não foi provada. Sabe-se que se $P = NP$ elas certamente não existem, mas não está claro se elas existem caso $P \neq NP$. Entretanto, há vários exemplos de funções que acredita-se ser unidirecionais na prática, como as funções hash da família SHA, utilizadas no PRG do Linux. Há, também, funções que se conjectura serem unidirecionais, como a função de decodificação do vetor síndrome, utilizada neste trabalho, ou o cálculo de logaritmo discreto módulo um grande número primo, ou o problema da soma dos subconjuntos. Essas funções pertencem à classe NP e, desde que $P \neq NP$ e sob alguma suposição de intratabilidade, são unidirecionais.

A principal diferença entre os dois tipos de funções unidirecionais, além da base teórica, é o custo computacional. As funções baseadas em problemas matemáticos intratáveis demandam, em geral, uma quantidade maior de cálculos por bit gerado. Conforme demonstrado por Impagliazzo et al. [1989] a existência de funções unidirecionais é condição necessária e suficiente para a existência de geradores de números pseudo-aleatórios, motivo pelo qual ela sempre estará presente na construção de qualquer PRG.

No gerador proposto neste trabalho utilizamos a estratégia proposta por Fischer & Stern [1996] para a construção de uma função geradora baseada no problema NP-Completo da decodificação do vetor síndrome, da teoria de códigos corretores de erros.

3.1.1 O problema da decodificação do vetor síndrome

Em teoria de códigos, a *decodificação* é o processo de tradução de mensagens em *palavras* pertencentes a um determinado *código*. A decodificação é utilizada na transmissão de mensagens em canais de comunicação ruidosos, que podem produzir erros. Um *código linear binário* é um código corretor de erros que utiliza os símbolos 0 e 1, no qual cada *palavra* do código pode ser representada como uma combinação *linear* das demais, e cada palavra tem um *peso*, isto é, um número de bits 1, definido.

Um código linear binário (n, k, d) é um subespaço de $\{0,1\}^n$ com 2^k elementos no qual cada palavra tem peso menor ou igual a d. A taxa de informação do código é k/n e ele pode corrigir até $\lfloor \frac{d-1}{2} \rfloor$ erros. Todo código pode ser definido por uma matriz de paridade de tamanho $n \times (n-k)$ que, quando multiplicada (módulo 2) por um vetor $x = \big(x_1, x_2, \ldots, x_n\big)$ integrante do código, produz um resultado $s = \big(0, 0, \ldots, 0\big)$, de tamanho $(n-k)$, denominado *síndrome*. Inversamente, quando a palavra multiplicada pela matriz de paridade não pertence ao código o valor do síndrome é diferente de zero.

Uma matriz de paridade preenchida aleatoriamente define um código linear binário aleatório. Para este código não existe algoritmo eficiente conhecido que possa encontrar a palavra mais próxima de um vetor, dado um síndrome diferente de zero. Outro problema difícil, conhecido como *decodificação do vetor síndrome*, ou *syndrome decoding*, é encontrar uma palavra de determinado peso a partir de seu síndrome.

Este último problema é da classe NP-Difícil e pode ser descrito da seguinte forma: seja uma matriz binária $A = \{a_{ij}\}$ de dimensões $n \times (n-k)$, um vetor síndrome não-nulo s e um inteiro positivo w. Encontre um vetor binário x com peso $|x| \leq w$, tal

que:

$$(x_1, x_2, \ldots, x_n) \cdot \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n-k} \end{pmatrix} = (s_1, s_2, \ldots, s_{n-k}) \pmod 2$$

O caso em que buscamos o vetor x com peso $|x| = w$ é NP-Completo [Berlekamp et al., 1978].

O fato de um problema ser da classe NP garante que não existe algoritmo de tempo polinomial para resolver o pior caso. Muitos problemas, entretanto, podem ser resolvidos de forma eficiente no caso médio, tornando necessário um estudo mais detalhado de cada instância do problema.

3.1.1.1 Limite de Gilbert-Warshamov

Para avaliar a dificuldade de uma instância específica do problema da decodificação do vetor síndrome utilizaremos um conceito estudado extensivamente e revisado por Chabaud [1994], segundo o qual as instâncias mais difíceis do problema da decodificação do vetor síndrome para códigos aleatórios são as de pesos na vizinhaça do limite de Gilbert-Warshamov para as dimensões do código.

O limite de Gilbert-Warshamov λ de um código (n, k, d) é definido pela relação $1 - k/n = H_2(\lambda)$ onde $H_2(x) = -x \log_2 x - (1 - x) \log_2(1 - x)$ é a função binária de entropia.

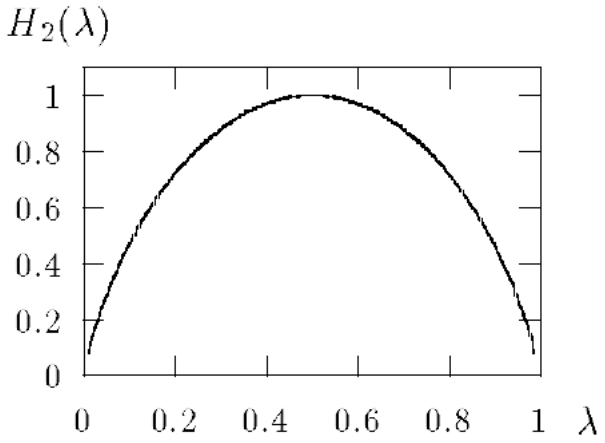

Figura 3.1. Limite de Gilbert-Warshamov, definido pela função binária de entropia.

Conforme a análise de Fischer & Stern [1996], existe, em média, um vetor para

cada síndrome quando o peso do vetor está em torno do limite de Gilbert-Warshamov para a dimensão do código. Ou seja, a dificuldade de encontrar uma palavra é uma função do peso crescente até o limite de Gilbert-Warshamov, e decrescente após o mesmo. Assim, pode-se definir uma instância difícil do problema da decodificação do síndrome quando o peso do vetor está próximo do limite de Gilbert-Warshamov.

3.1.1.2 Definição formal do problema

Definição Uma função $f : \{0,1\}^* \to \{0,1\}^*$ é considerada fortemente unidirecional se as condições a seguir são válidas:

- *Fácil de Computar*: existe um algoritmo determinístico de tempo polinomial A tal que, para cada entrada x, é produzida uma saída $A(x) = f(x)$.
- *Difícil de Inverter*: para todo algoritmo probabilístico de tempo polinomial A' e todo polinômio positivo $p(n)$ suficientemente grande,

$$Pr(A'(f(X_n)) \in f^{-1}(f(X_n))) < \frac{1}{p(n)}$$

onde X_n é uma variável aleatória distribuída uniformemente sobre $\{0,1\}^n$

Vamos considerar agora uma coleção de funções relativas ao problema de decodificação do síndrome.

Definição Seja ρ em $]0,1[$ e δ em $]0,1/2[$. Uma coleçao $SD(\rho,\delta)$ é um conjunto de funções f_n tal que:

$$\begin{aligned} D_n &= \{(M,x), M \in \lfloor \rho n \rfloor \times n, x \in \{0,1\}^n / |x| = \delta n\} \\ f_n &: D_n \to \{0,1\}^{\lfloor \rho n \rfloor \cdot (n+1)} \\ &(M,x) \to (M, M \cdot x) \end{aligned}$$

Conforme asseveram Fischer & Stern [1996], instâncias do problema nas quais o peso δn é muito pequeno ou próximo a $n/2$ não são difíceis. As instâncias da coleção SD que acredita-se serem unidirecionais, apesar de não existir prova neste sentido, ocorrem quando o peso δ está nas proximidades do limite de Gilbert-Warshamov. Assim, temos a seguinte suposição de intratabilidade:

Suposição de Intratabilidade Seja ρ em $]0,1[$. Então, para todo δ em $]0,1/2[$, tal que $H_2(\delta) < \rho$, a coleção $SD(\rho,\delta)$ é fortemente unidirecional.

Observe que se $H_2(\lambda) = \rho$ e $\delta < \frac{\lambda}{2}$, a intratabilidade de $SD(\rho, \delta)$ é um caso particular da dificuldade de decodificação abaixo da metade da distância mínima. Assim, a suposição se torna mais forte que as suposições usuais para o problema da decodificação [Goldreich et al., 1993].

Aplicações criptográficas baseadas na complexidade de problemas conhecidos foram extensivamente estudadas e implementadas nas últimas décadas, e as suposições de intratabilidade são um passo natural na construção dessa aplicações. À luz do estado atual do conhecimento em teoria da complexidade, não se pode esperar construir tais algoritmos sem qualquer suposição de intratabilidade [Goldreich, 2001, p. 19].

3.2 Ciclo e Nível de Segurança

Qualquer gerador de números pseudo-aleatórios, enquanto não atualizado por fontes de entropia, pode entrar em ciclo após um determinado número de iterações, repetindo valores já produzidos. A ocorrência de ciclos é inerente à geração algorítmica dos valores pseudo-aleatórios a partir de um estado interno *finito* e, caso não seja tratada, pode prejudicar os protocolos e aplicativos criptográficos que dependem do gerador.

Considerando que qualquer PRG pode ter seu estado interno representado por um vetor de k bits, atualizado a cada iteração, pelo paradoxo do aniversário, ou *birthday paradox*, podemos dizer que após *aproximadamente* $2^{k/2}$ iterações a probabilidade de ocorrer uma colisão de estados do gerador supera 50%. Ou seja, um PRG com estado interno de 128 bits pode entrar em ciclo após a geração de 2^{64} valores, fator bem menor que o necessário para se obter, por força bruta, o estado interno do mesmo: 2^{128}.

Além dos ciclos, o projetista de um PRG deve estar atento ao nível de segurança desejado para o gerador. O nível de segurança, conforme definido em Barker & Kelsey [2006], é um valor associado à quantidade de trabalho necessária para se quebrar o algoritmo ou sistema criptográfico. No gerador apresentado acima, caso não fosse imposto nenhum limite na quantidade de bits requisitados, o nível de segurança cairia de 128 para 64, pois após 2^{64} passos o gerador provavelmente passaria a operar em ciclo, produzindo valores conhecidos do adversário. Em geral, os PRGs modernos limitam a quantidade de requisições entre as atualizações da semente, para evitar a ocorrência de ciclos e aumentar o nível de segurança do sistema.

3.3 Requisitos de um PRG criptograficamente seguro

De modo geral, um gerador deve ser protegido contra ataques externos e internos. Deve-se assumir que o adversário conhece o código do gerador e pode ter informação *parcial* sobre as fontes de entropia utilizadas para atualizar seu estado. Os requisitos de segurança para um PRG são:

- Resiliência: os valores produzidos pelo gerador devem parecer aleatórios para um adversário sem conhecimento do estado interno do gerador. Ou seja, após a geração de k bits, um adversário não pode prever o valor do bit $k+1$ com probabilidade $p > \frac{1}{2}$.
- Forward Security: os valores produzidos no passado devem parecer aleatórios para um adversário, mesmo que este tenha conhecimento do estado interno do gerador em um tempo futuro.
- Backward Security/Break-in recovery: os valores produzidos no futuro devem parecer aleatórios para um adversário, mesmo que ele tenha conhecimento do estado interno atual do gerador. Esta propriedade só pode ser satisfeita caso o gerador seja atualizado com entropia suficiente para gerar um novo estado interno que seja desconhecido do adversário.

3.4 Modelos de ataques a PRGs

Existem várias formas de atacar um PRG. A mais simples, chamada *criptanálise direta*, é a tentativa de reconstrução do estado interno do gerador utilizando sua saída. Nesse caso o adversário captura grandes quantidades de dados e tenta inverter a função geradora. A menos que o adversário consiga alguma informação adicional sobre o estado desta, este tipo de ataque dificilmente traz resultados, devido à propriedade unidirecional da função geradora.

A situação se torna mais complicada quando o adversário consegue, por algum meio, obter o estado interno do gerador. Para os propósitos desta discussão não é importante *como* isso pode ter ocorrido: uma falha na implementação, um computador recém-iniciado e ainda sem semente ou a captura da semente aleatória gravada pelo PRG no disco rígido podem ser estratégias para se adquirir o estado interno do gerador.

Uma vez descoberto o estado interno, o adversário pode tentar um ataque de *extensão de comprometimento de estado*, ou *state compromise extension*, visando man-

ter o controle sobre o gerador e ainda descobrir os valores gerados no passado. Um PRG criptograficamente seguro deve proteger os valores gerados no passado e, também, eventualmente, se recuperar para um estado seguro, a fim de gerar valores futuros desconhecidos do adversário.

A proteção dos valores gerados no passado pode ser alcançada por meio da atualização do estado interno do gerador utilizando uma função de feedback unidirecional f, tal que o estado no tempo t seja $E_t = f(E_{t-1}, ...)$. Dessa forma, uma vez comprometido o estado E_t do gerador, os n estados anteriores E_{t-n} estarão seguros, a menos que a função unidirecional possa ser invertida.

A recuperação do gerador para um estado seguro após um comprometimento é uma tarefa mais complexa. A única forma de se recuperar o controle do gerador é atualizando-o para um estado desconhecido do adversário, utilizando entropia coletada das fontes de aleatoriedade. Entretanto, a quantidade de entropia utilizada deve ser suficiente para evitar que o adversário recupere novamente o estado do gerador. Isto é, se atualizarmos o gerador com, por exemplo, 30 bits de entropia, o adversário poderá simplesmente testar todas as entradas possíveis até recuperar o novo estado interno. Para isso serão necessário no máximo 2^{30} tentativas, o que é viável de se realizar.

A melhor defesa para esse tipo de ataque é acumular os bits aleatórios e somente utilizá-los quando a quantidade de entropia for suficiente para realizar uma atualização catastrófica, ou seja, uma atualização que torna inviável, computacionalmente, de se obter o novo estado interno.

Caso o gerador seja projetado para que qualquer ataque exija no mínimo 2^{128} passos, o *pool* de bits aleatórios deverá ter 128 bits de entropia. Entretanto, fazer qualquer estimativa sobre a quantidade de entropia acumulada é extremamente difícil, senão impossível. Essa medição depende de quanto o adversário conhece ou pode conhecer sobre as fontes de aleatoriedade utilizadas ou mesmo de como ele pode influenciá-las. Um adversário pode, por exemplo, após o comprometimento do estado de um gerador, e conhecendo a fórmula de estimativa da entropia, criar eventos de forma a inflar esse cálculo, evitando a atualização catastrófica do gerador e garantindo controle indefinido sobre este.

Capítulo 4

Síndrome-Fortuna

A proposta do presente trabalho é desenvolver um gerador de números aleatórios, com coleta de entropia, baseado na intratabilidade de um problema da teoria de códigos corretores de erros e implementá-lo no kernel do Linux. Demonstraremos que o gerador proposto aplica bons fundamentos de segurança e é baseado em sólidos princípios matemáticos.

O gerador foi projetado com as funções de coleta de entropia e geração de valores aleatórios disjuntas, sendo que cada uma será abordada em uma seção deste capítulo. Serão abordados também o problema da inicialização do gerador após um boot normal e após uma instalação completamente nova de um sistema, além da implementação do algoritmo no kernel do Linux.

4.1 Coleta de Entropia

Ferguson & Schneier [2003] propuseram um gerador de números aleatórios resistente a ataques, denominado Fortuna. A principal contribuição do algoritmo proposto foi o sistema de coleta de aleatoriedade, que elimina a necessidade de estimadores de entropia, utilizados, até então, na maioria dos projetos de geradores que temos conhecimento. Conforme vimos anteriormente, a estimativa de entropia é utilizada para garantir que as atualizações de estado se deem de forma catastrófica, ou seja, com quantidade de aleatoriedade suficiente para impedir que eventuais adversários mantenham controle sobre o estado do gerador.

4.1.1 Acumulador de entropia

O acumulador de entropia do Fortuna captura dados de diversas fontes de entropia que depois são utilizados para atualizar a semente do gerador. Sua arquitetura, conforme veremos a seguir, permite que o sistema se mantenha seguro mesmo que um adversário controle algumas das fontes de entropia.

Os eventos aleatórios capturados pelas fontes de entropia devem ser *uniforme* e *ciclicamente* distribuídos entre os n pools do gerador, conforme a figura 4.1. Essa distribuição dos eventos aleatórios entre os pools irá garantir que a entropia também seja distribuída uniformemente e, conforme veremos na seção 4.1.3, permitirá que quantidades sucessivamente maiores de aleatoriedade sejam utilizadas nas atualizações de estado do gerador.

Cada pool pode receber, em teoria, dados aleatórios ilimitados. Para implementar esse pool ilimitado os dados são comprimidos de forma incremental, utilizando a função hash SHA-256, mantendo assim pools com tamanho constante de 256 bits.

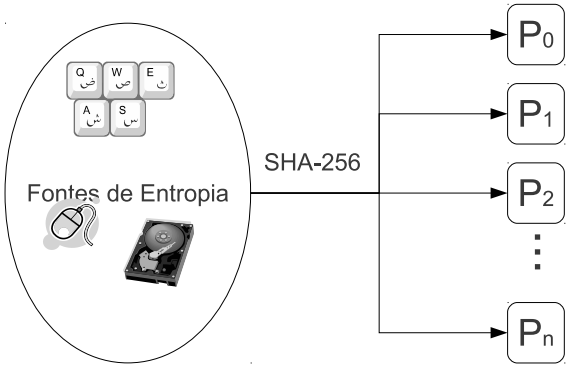

Figura 4.1. Alocação de entropia entre n pools do algoritmo Fortuna. Dados são comprimidos usando a função hash SHA-256.

Quando o pool P_0 tiver acumulado dados aleatórios suficientes, a semente do gerador pode ser atualizada. Uma variável *counter* mantém controle de quantas vezes a semente do gerador já foi atualizada. Esse contador determina quais pools serão usados na atualização a cada vez, sendo que o pool P_i será usado caso 2^i divida *counter*. A tabela 4.1.1 demonstra o esquema de atualização do gerador.

Quanto maior a numeração do pool, mais raramente ele é utilizado na atualização do gerador e, por isso, maior a quantidade de entropia acumulada. Isso permite que o gerador se adapte a cada situação de ataque de forma automática. Caso o adversário tenha pouco controle sobre as fontes de aleatoriedade, ele não poderá prever sequer o estado do pool P_0, e o gerador se recuperará de um comprometimento de estado rapidamente, no próximo *reseed*.

Tabela 4.1. Pools utilizados nas primeiras 8 atualizações do gerador Fortuna.

Counter	Pools utilizados
1	P0
2	P0, P1
3	P0
4	P0, P1, P2
5	P0
6	P0, P1
7	P0
8	P0, P1, P2, P3

O adversário pode, entretanto, controlar várias das fontes de entropia. Neste caso ele provavelmente saberá bastante sobre o pool P_0 e poderá reconstruir o estado do gerador utilizando o estado anterior e as saídas produzidas. Mas quando P_1 for utilizado em um *reseed*, ele conterá duas vezes a quantidade de aleatoriedade de P_0. Quando P_2 for utilizado, este conterá quatro vezes a quantidade de aleatoriedade de P_0, e assim sucessivamente. Enquanto houver uma fonte de aleatoriedade desconhecida pelo adversário, haverá um pool que coletará aleatoriedade suficiente para derrotá-lo. Independentemente de quantas eventos falsos o adversário possa gerar ou de quantos eventos ele conhece o sistema irá se recuperar, sendo a velocidade da recuperação proporcional ao nível de controle do adversário sobre as fontes de entropia.

4.1.2 Estimativa inicial

Apesar de o esquema proposto dispensar a estimativa de entropia ativa, como a utilizada no Linux (vide seção 2.4.2), ainda é necessário realizar uma estimativa inicial de entropia, a fim de determinar qual a quantidade mínima de eventos suficiente para realizar uma atualização de estado catastrófica. Essa estimativa é calculada para o pool P_0 e determinará quando será feita a atualização do estado do gerador.

A estimativa deve ser elaborada com base no nível de segurança projetado, no espaço ocupado pelos eventos e na quantidade de entropia presente em cada evento. Se considerarmos, por exemplo, que cada evento fornece 4 bits de entropia e ocupa 32 bits no pool, para atingirmos um nível de segurança de 256 bits, serão necessários 256 bits/(4 bits/evento) = 64 eventos. Assim, o tamanho mínimo do pool para realizar um reseed será: 64 eventos $*$ (32 bits/evento) = 2048 bits. Na verdade, cada pool é implementado como um estado de um hash SHA-256 e tem um tamanho fixo de 256 bits. O cálculo do "tamanho" do pool é importante para diferenciar entre os tipos de eventos, que podem fornecer quantidades diferentes de entropia em relação ao tamanho

ocupado nos pools.

Ferguson & Schneier sugerem um tamanho mínimo de P_0 de 512 bits, para um nível de segurança de 128 bits. A estimativa inicial de entropia tem papel relevante na segurança do sistema, mas é atenuada pelo fato de que, caso o valor escolhido seja muito baixo, haverá sempre *reseeds* com quantidades superiores de entropia. Caso o valor escolhido seja muito alto, uma eventual recuperação de um estado comprometido poderá demorar, mas fatalmente irá ocorrer.

4.1.3 Requisitos de uniformidade e ciclicidade

Para que o algoritmo funcione adequadamente, os dados das fontes aleatórias devem ser distribuídos de forma uniforme e independente entre os pools. Assim, a escolha de qual pool deve receber cada evento é um ponto importante da arquitetura do gerador.

A forma mais intuitiva de escolha para a alocação dos eventos aleatórios entre os n pools seria manter um contador cíclico $i = \{1, ..., n\}$ na função que acumula a entropia e alocar os dados aleatórios para o pool P_i a cada chamada. Essa alternativa não é a mais segura, entretanto, pois um adversário poderia gerar falsos eventos de forma a controlar o valor da variável i, concentrando os eventos aleatórios verdadeiros em um pool específico, por exemplo.

A proposta de Ferguson & Schneier para a alocação dos eventos é que cada driver mantenha um contador individual e faça a escolha de qual pool deverá receber os dados enviados. Dessa forma, o adversário não poderia influenciar a escolha dos pools gerando eventos falsos, a menos que ele pudesse alterar um valor interno do driver em questão, o que, neste caso, comprometeria a fonte aleatória como um todo.

Como o escopo do projeto não inclui a modificação da arquitetura de drivers do sistema Linux, a escolha dos pools foi implementada dentro da função acumuladora, com um contador cíclico $K_f = 1, ..., n$, para cada fonte de aleatoriedade f.

4.2 Função Geradora

A função geradora implementada neste trabalho foi construída a partir do problema da decodificação do vetor síndrome, utilizando os conceitos apresentados no capítulo 3. Apresentamos a seguir a construção da função geradora.

4.2.1 Construção da função geradora

Vamos construir um PRG baseado em uma instância difícil do problema da decodificação do vetor síndrome, utilizando a coleção de funções $SD(\rho, \delta)$, definida na seção 3.1.1. Inicialmente, vamos mostrar que as funções $f_n^{\rho,\delta}$ da coleção $SD(\rho, \delta)$ expandem suas entradas, isto é, têm o conjunto imagem maior que o conjunto domínio.

O domínio $D_n^{\rho,\delta}$ de $f_n^{\rho,\delta}$ é formado pela matriz M de tamanho $\lfloor \rho n \rfloor \times n$ e do vetor x de tamanho n e com peso δn. Logo, o tamanho do conjunto domínio é $2^{\lfloor \rho n \rfloor \cdot n} \cdot \binom{n}{\delta n}$. Já o conjunto imagem é formado pela mesma matriz M de tamanho $\lfloor \rho n \rfloor \times n$ e de um vetor $y = M \cdot x$ de tamanho $\lfloor \rho n \rfloor$. Assim, o tamanho deste é $2^{\lfloor \rho n \rfloor \cdot n} \cdot 2^{\lfloor \rho n \rfloor}$.

Assim, para que o tamanho do conjunto imagem seja maior que o tamanho do conjunto domínio, precisamos que $2^{\lfloor \rho n \rfloor} > \binom{n}{\delta n}$. Esta condição é satisfeita quando $H_2(\delta) < \rho$, conforme o limite de Gilbert-Warshamov, definido na seção 3.1.1.1. Ou seja, para um n suficientemente grande e δ e ρ adequados, $f_n^{\rho,\delta}$ expande sua entrada em uma quantidade linear de bits.

Dada uma instância com parâmetros fixos ρ e δ da coleção $SD(\rho, \delta)$, podemos construir uma função geradora iterativa $G_{\rho,\delta}$ a partir da função unidirecional $f_n^{\rho,\delta}$. Para isso, utilizaremos um algoritmo A que retorna um vetor x de tamanho n e peso δn a partir de um vetor aleatório com $\log_2 \binom{n}{\delta n}$ bits. Esse algoritmo está descrito na seção 4.2.1.1. O gerador $G_{\rho,\delta}$ está descrito em pseudocódigo no algoritmo 6.

[Algoritmo] 6 *sindrome* – Função geradora iterativa baseada no problema da decodificação do vetor síndrome.

Entrada: $(M, x) \in D_n^{\rho,\delta}$
Saída: Imprime seqüência de bits

1: $y \leftarrow M \cdot x$
2: **Separe** y em dois vetores y_1 e y_2. O primeiro com $\lfloor \log_2 \binom{n}{\delta n} \rfloor$ bits e o segundo com o restante dos bits.
3: **imprima** y_2
4: $x \leftarrow A(y_1)$
5: **Vá para:** 1

A figura 4.2 ilustra o funcionamento do gerador.

4.2.1.1 Geração de palavras de determinado peso

Conforme vimos, a construção do gerador requer um algoritmo que, a cada entrada de tamanho $\lfloor \log_2 \binom{n}{\delta n} \rfloor$, produza um vetor de saída distinto $x \in \{0,1\}^n$ com peso $|x| = \delta n$. Para construí-lo, utilizaremos a função de enumeração lexicográfica mostrada por Fischer & Stern [1996].

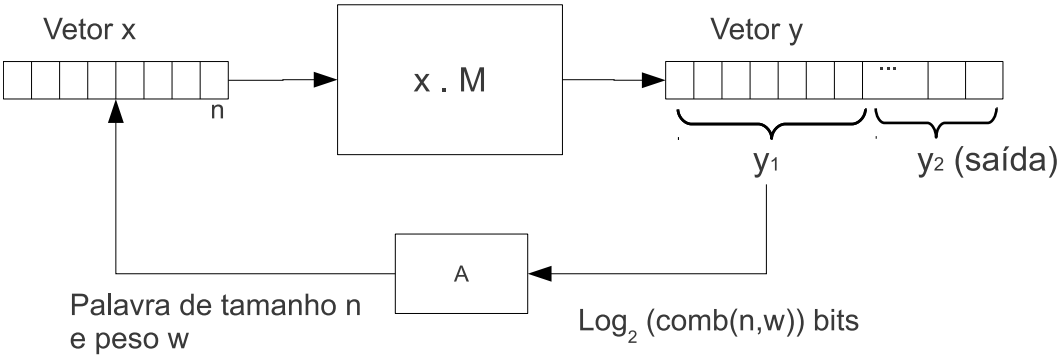

Figura 4.2. Funcionamento do gerador Síndrome.

Vamos utilizar o *Triângulo de Pascal*, que é uma tabela formada pelos coeficientes binomiais $\binom{n}{k}$ no qual n representa a linha e k a coluna. A propriedade de igualdade fundamental do Triângulo de Pascal é a seguinte. Sejam $n, k \in \mathbb{N}^*, n > k$ então:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$$

Cada componente do triângulo $t(n,k) = \binom{n}{k}$ representa o número de palavras existentes com tamanho n e peso k e é igual à soma dos dois componentes imediatamente acima deste $t(n-1,k)$ e $t(n-1,k-1)$. Estes componentes representam, respectivamente, o número de palavras de tamanho n começando com um bit *0* e começando com um bit *1*.

Dessa forma, a partir de um índice i, podemos gerar a palavra de saída fazendo um caminho de baixo para cima no Triângulo de Pascal, começando do componente $t(n,k)$ em direção ao topo. Quando o índice i for menor ou igual ao componente imediatamente acima e à *esquerda* $t(n-1,k)$, geramos um bit *0* e caminhamos para este componente. Quando o índice for maior, geramos um bit *1* e caminhamos para o componente à *direita* $t(n-1,k-1)$, decrementando o índice em $t(n-1,k-1)$. O algoritmo completo está descrito em pseudocódigo ao final desta seção.

Para exemplificar, veja a caminhada para gerar uma palavra de tamanho $n = 4$ e peso $k = 2$, de índice $i = 2$, ilustrada na figura 4.3.

O caminho começa no componente $t(4,2) = \binom{4}{2} = 6$. Como $i = 2 \leq t(3,2) = 3$, o algoritmo gera um bit 0 e caminha para o componente $t(3,2)$. Agora, $i = 2 > t(2,2) = 1$, logo, o algoritmo gera um bit 1, atualiza o índice $i = i - t(2,2) = 1$ e o caminho segue para o componente $t(2,1)$. E assim segue até chegar ao topo do triângulo, formando a palavra $(0,1,0,1)$.

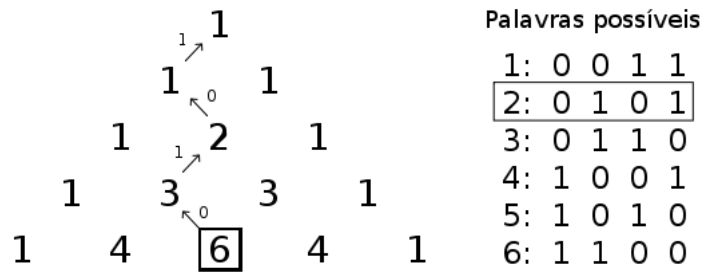

Figura 4.3. Caminhada no Triângulo de Pascal para gerar palavra de índice $i = 2$ dentro de um conjunto de palavras de tamanho 4 e peso 2.

[Algoritmo] 7 *A* – Algoritmo de ordenação lexicográfica que retorna palavras de determinado tamanho e peso.

Entrada: $i \in \{0,1\}^{\lfloor \log_2 \binom{n}{w} \rfloor}, n, w$
Saída: Imprime seqüência de bits com tamanho n e peso w

```
1: t ← Triângulo de Pascal com n linhas
2: c ← t(n, w)
3: enquanto n > 0 faça
4:    se i ≤ c.esquerda então
5:       imprima 0
6:       c ← c.esquerda
7:    senão
8:       imprima 1
9:       i ← i − c.esquerda
10:      c ← c.direita
11:   fim se
12:   n ← n − 1
13: fim enquanto
```

4.2.1.2 Heurística para reduzir espaço de memória

O algoritmo apresentado na seção 4.2.1.1, utiliza uma quantidade considerável de memória. Para armazenar o Triângulo de Pascal para o binômio $\binom{n}{w}$, são necessários $n * w$ posições, cada uma capaz de armazenar o tamanho máximo da combinação $\lceil \log_2 \binom{n}{w} \rceil$.

Para uma configuração com $n = 512$ e $w = 52$, o espaço de memória ocupado por cada posição da tabela de binômios é de 239 bits, obrigando o armazenamento em 32 bytes de memória. O espaço total ocupado chega a $512 * 52 * 32 = 832KB$. Esse valor, dependendo do cenário, pode representar um custo proibitivo para o algoritmo.

Uma heurística para resolver esse problema seria substituir o algoritmo de enumeração lexicográfica por um que recebe o índice i e retorna uma sequência de w palavras aleatórias de tamanho $\log_2 n$. Estas palavras podem ser utilizadas como índices para

determinar as posições dos bits 1 do vetor x desejado. Para que a heurística funcione corretamente, ciclos não podem ocorrer no período w e a função deve produzir sequências diferentes para cada semente i fornecida.

Até onde conhecemos, o gerador TGFSR possui boas propriedades e pode ser um bom candidato. O período do gerador é o máximo possível, isto é, para cada semente são gerados todos os valores possíveis em um determinado período. As propriedades do gerador não garantem, entretanto, que colisões não ocorram, isto é, sementes diferentes podem gerar sequências de valores iguais.

Estes geradores precisam ser melhor estudados para serem utilizados sem prejudicar drasticamente a segurança do esquema, que reside na propriedade bijetiva da função lexicográfica. Ou seja, cada índice $i \in \{0,1\}^{\lfloor \log_2 \binom{n}{w} \rfloor}$ retorna o vetor x correspondente com tamanho n e peso w, e cada vetor x corresponde a somente um índice i. Não podemos garantir o mesmo para as sequências produzidas pelo TGFSR, motivo pelo qual a implementação dessa estratégia requer maiores investigações.

4.3 Síndrome-Fortuna

Descreveremos nesta seção a estratégia adotada para unir os dois algoritmos apresentados, preservando suas propriedades de segurança, e construir um gerador criptograficamente robusto.

A função geradora construída em 4.2.1 baseia-se na dificuldade de se encontrar um vetor x com peso w a partir de seu síndrome, dada uma matriz M aleatória. Assim, a única parte da função que efetivamente compõe o estado interno E, que deve ser mantido secreto, é o vetor x. Vamos utilizar, então, a estratégia de coleta de entropia proposta para atualizar o valor do estado interno E sempre que houver a quantidade mínima de entropia disponível no pool P_0. Dessa forma, vamos garantir que a função geradora receba a entropia gerada pelo sistema operacional e tenha seu estado atualizado ao longo do tempo.

A cada requisição, é feita uma verificação se existe entropia suficiente para uma atualização do estado interno. Esta verificação é condicionada à existência de entropia mínima no pool P_0, conforme uma estimativa inicial, e respeita a distância mínima de tempo entre *reseeds* de 100ms, conforme sugerido por Ferguson & Schneier [2003]. A temporização entre atualizações de estado é realizada para evitar que um adversário possa fazer inúmeras requisições, visando eliminar a entropia nos pools existentes.

A figura 4.4 ilustra o funcionamento completo do gerador.

A estratégia de atualização do gerador Síndrome-Fortuna preserva a resistência

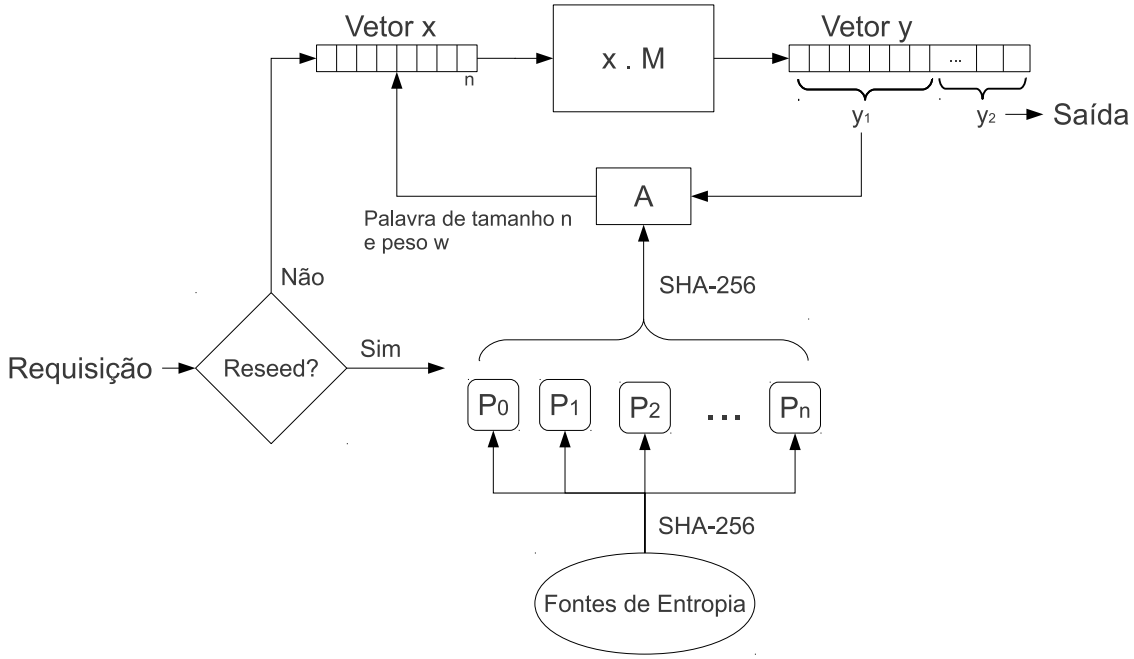

Figura 4.4. Funcionamento do gerador Síndrome-Fortuna. A cada requisição é verificado se o pool P_0 possui entropia mínima e se o tempo entre *reseeds* é superior a 100ms. Em caso positivo o algoritmo realiza uma atualização de estado, incrementando um contador c e escolhendo entre os pools P_i aqueles que satisfazem a condição 2^i divide c. É realizado um hash sha-256 dos pools escolhidos e o resultado é utilizado para indexar uma palavra de tamanho n e peso w para a função geradora. A função geradora realiza a multiplicação entre o vetor escolhido e a matriz M produzindo o síndrome y, que tem parte enviada à saída e parte utilizada para realizar o *feedback* da função geradora, possibilitando a geração iterativa dos dados aleatórios.

do algoritmo contra criptanálise direta, uma vez que a função unidirecional mantém sua operação, sendo somente a semente, o vetor x, eventualmente atualizado antes do processamento da requisição. Independente dessa atualização, qualquer adversário que consiga reconstruir o estado interno, por meio do vetor saída y e da matriz M, terá conseguido resolver um problema considerado, até o momento, na literatura, como computacionalmente intratável.

A propriedade *forward security* é garantida pela operação de feedback, na qual parte do vetor resultado y é utilizada para escolher o novo vetor x. Um adversário pode, no tempo t, conhecer o estado do gerador $E_t = (x_t, M, P_t)$, onde M é a matriz e x_t e P_t representam os valores do vetor x e de todos pools do sistema no tempo t, respectivamente. Nesse caso, o adversário poderá apenas descobrir o último valor utilizado como índice da função de enumeração lexicográfica $y1_{t-1}$. Este valor é uma parte do vetor y_{t-1}, como pode ser visto na figura 4.4. A partir daí, descobrir o valor x_{t-1} requer que o adversário inverta a função geradora utilizando apenas informação

parcial do vetor síndrome. Descobrir o último valor de saída, $y2_{t-1}$, requer, também, inverter a função geradora, uma vez que o vetor parcial $y1_{t-1}$ não contém nenhuma informação sobre o vetor parcial $y2_{t-1}$.

A recuperação para um estado seguro após um comprometimento – *backward security* – é garantida pela atualização eventual do vetor x, pelo sistema de coleta de entropia. Um adversário que controle o estado do gerador $E_t = (x_t, M, P_t)$, poderá mantê-lo até que, em um tempo $t + k$, a quantidade de entropia acumulada nos pools seja suficiente para uma atualização catastrófica. Nesse momento o valor do vetor x será alterado por meio da função SHA-256 dos pools do sistema, conforme visto na figura 4.4. Caso o adversário não tenha conhecimento da entropia adicionada aos pools, o novo estado E_{t+k} estará fora de seu controle.

A quantidade de entropia necessária para um sistema *ideal* se recuperar é de 128 bits. No sistema de coleta de entropia do Fortuna esta quantidade é multiplicada pelo número de pools, uma vez que a entropia é distribuída. Assim, esta quantidade sobe para $128 * n$, onde n é o número de pools. Este valor pode ser relativamente alto, se comparado ao ideal, porém, este é um fator constante, e garante que o sistema irá se recuperar, em um tempo futuro.

No caso do comprometimento acima, considerando a taxa ω de entrada de entropia no sistema, o tempo de recuperação do gerador seria, no máximo, $k = (128 * n)/\omega$. Um adversário poderia tentar ir esgotando a entropia do sistema, enquanto ela é acumulada. Para isso ele precisaria promover reseeds rápidos, de forma a esvaziar os pools antes que eles contenham os 128 bits de entropia suficientes para derrotá-lo. Seria preciso, também, injetar eventos falsos no sistema, por meio de drivers maliciosos, de forma a manter o pool P_0 cheio, permitindo as atualizações de estado.

Este ataque é improvável, dado que o adversário teria que promover 2^n reseeds antes de o sistema coletar $128*n$ bits de entropia real. Entretanto, para evitar qualquer tentativa, é inserido um tempo mínimo entre atualizações de estado, de forma a evitar que reseeds muito frequentes esgotem a entropia do sistema.

4.4 Inicialização do gerador

A inicialização é uma etapa crítica para qualquer gerador de números aleatórios que gerencia a própria entropia. Os problemas relacionados à falta de aleatoriedade na inicialização devem ser sanados conforme as possibilidades de cada cenário. Foge ao escopo deste trabalho, portanto, definir estratégias específicas, uma vez que estas variam para cada arquitetura específica.

Cabe ressaltar, entretanto, que o acumulador de entropia implementado permite a utilização de quaisquer fontes de aleatoriedade. Mesmo uma fonte de baixa qualidade, que pode inserir dados previsíveis nos pools, não tem a capacidade de diminuir a entropia do sistema. Dessa forma, quaisquer fontes disponíveis, mesmo que de qualidade questionável, podem ser incluídas no processo de inicialização, uma vez que elas só podem aumentar a entropia dos pools.

A estratégia de simular continuidade entre as reinicializações mostra-se uma boa forma de mitigar o problema da falta de entropia em grande parte dos casos. Para o gerador Síndrome-Fortuna a gravação do arquivo-semente foi implementada da mesma forma que é feito atualmente no Linux, por meio da gravação em disco durante o desligamento e na inicialização e com a recuperação por meio da função acumuladora de entropia, passando como parâmetro o arquivo-semente.

Capítulo 5

Implementação, análise e resultados

Neste capítulo iremos mostrar detalhes da implementação e fazer uma análise do algoritmo e dos resultados de desempenho obtidos, buscando comprovar a viabilidade da solução proposta. A seção 5.1 mostra algumas decisões de implementação. A seção 5.2 apresenta a metodologia utilizada na análise estatística da aleatoriedade produzida e no desempenho do gerador. As seções 5.3 e 5.4 apresentam o resultados obtidos.

5.1 Implementação

O objetivo da implementação do Síndrome-Fortuna no kernel Linux foi permitir a realização de testes de desempenho e sua comparação com o gerador de números aleatórios nativo do Linux. Assim, na implementação do algoritmo no kernel, a interface */dev/urandom* foi mantida e a interface */dev/random* foi substituída pela função geradora do Síndrome-Fortuna. Foram necessárias alterações em três partes do código original *random.c* para tornar o gerador Síndrome-Fortuna operacional. Apresentaremos abaixo as estruturas de dados utilizadas e as funções alteradas para a integração do gerador ao kernel.

Estruturas de dados O Síndrome-Fortuna utiliza, na função geradora, três estruturas principais: o vetor x, a matriz M e o síndrome y. Estas estruturas foram implementadas por vetores e matrizes em C, utilizando o tipo de dados *unsigned int*, que tem tamanho de 32 bits. O acumulador de entropia foi implementado por um contexto de hash sha-256 para cada pool. Foi implementado, também, para cada pool, um vetor *batch* de 1024 bytes para acúmulo de eventos em tempo de interrupção, antes do processamento do hash. Essa estratégia tem o objetivo de

evitar o cálculo de uma operação de hash a cada evento do sistema, otimizando o funcionamento do gerador [Ferguson & Schneier, 2003].

rand_initialize A função *rand_initialize* inicializa as estruturas do gerador nativo do Linux durante a inicialização do kernel. Inserimos então, nessa função, a inicialização das estruturas do Síndrome-Fortuna. O vetor x e a matriz M são inicializados com valores pseudoaleatórios, utilizando o Mersenne Twister [Matsumoto & Nishimura, 1998], uma variação de TGFSR. É utilizado como semente para este gerador uma estrutura contendo hora e nome do sistema, assim como acontece no gerador do Linux. A opção pelo Mersenne Twister para gerar os valores iniciais do gerador se deu pela boa qualidade estatística dos valores pseudoaleatórios produzidos. A tabela de binômios do gerador é preenchida, conforme a regra básica de formação do Triângulo de Pascal. Os pools do acumulador de entropia têm seu contexto hash inicializado.

add_timer_randomness Para que o gerador Síndrome-Fortuna funcione corretamente, os eventos capturados devem ser acumulados nos pools do sistema. Para isso, foi inserida na função *add_timer_randomness* uma chamada para o acumulador de enventos do Síndrome-Fortuna. Conforme vimos na seção 2.4.1.1, esta função é invocada para cada interrupção do sistema configurada para adicionar entropia. O processo de acumulação de entropia foi implementado com um vetor *batch* para acúmulo de eventos até o tamanho de 1024 bytes. Quando o vetor está vazio, o acumulador realiza apenas uma cópia de memória. Quando o *batch* ultrapassa o valor máximo, o hash dos eventos acumulados é realizado.

random_read Foi alterada a função *random_read*, que implementa a leitura da interface */dev/random*. Esta função retorna a saída da função geradora do Síndrome-Fortuna, implementada conforme a figura 4.4.

Os parâmetros escolhidos para a função geradora foram os sugeridos por Fischer & Stern [1996]. Utilizamos a matriz M com 512x256 bits, o vetor x de tamanho $n = 512$ e peso $w = 52$. Cada iteração do algoritmo rende 18 bits de saída. Para simplificar a implementação, apenas 16 bits são enviados à saída por iteração. O acumulador de entropia foi implementado com 32 pools e com temporização entre reseeds de 100ms. Cada pool foi implementado como um contexto de hash sha-256.

A estratégia de coleta de entropia implementada segue a sugestão de Ferguson & Schneier [2003], utilizando o vetor *batch* como área de memória temporária. O objetivo é evitar a realização de operações de hash a cada interrupção do sistema, o que poderia gerar uma sobrecarga desnecessária. Assim, na implementação

atual, após 1024 bytes de eventos acumulados em cada pool, ou após uma requisição de dados aleatórios, o hash é processado.

Cabe ressaltar que em cenários onde o tempo de interrupção é sensível e não pode sofrer variações, há a possibilidade de implementar as operações de hash utilizando o escalonador de eventos do kernel *kevent*, fora do tempo de interrupção do sistema.

5.2 Metodologia dos experimentos

Uma das formas de avaliar a qualidade de um gerador de números aleatórios é avaliando a qualidade estatística da saída produzida. Os resultados desta análise não garantem, de forma alguma, a segurança de um gerador criptográfico, mas podem revelar falhas no projeto ou implementação de um gerador.

Existem várias baterias de testes estatísticos aceitas pela comunidade científica. Foram utilizadas as baterias *SmallCrush* e *Crush* da biblioteca TestU01, desenvolvida por L'Ecuyer & Simard [2007]. A primeira bateria implementa uma série de 10 testes e é recomendada para uma avaliação incial do gerador. A segunda bateria aplica 32 testes com diversas configurações diferentes, utilizando um total de 2^{35} bits aleatórios.

Para avaliação do desempenho do gerador, fizemos a comparação deste com o gerador Blum-Blum-Shub, que possui uma construção simples, baseada na dificuldade de fatorar inteiros, e com o gerador do kernel Linux 2.6.30 (LRNG). Foi utilizada a biblioteca TestU01 para medir o desempenho dos geradores.

5.3 Resultados Estatísticos

Os resultados dos testes estatísticos são apresentados na forma de *p-values*, que indicam a probabilidade de a amostra Y apresentar o valor amostrado y, considerando verdadeira a hipótese nula H_0:

$$p = P(Y \geq y | H_0)$$

Vamos avaliar, por exemplo, uma amostra Y de 100 sorteios de uma moeda em que 80 vezes foi sorteada "cara" e apenas 20 vezes "coroa". Neste caso, a hipótese nula é que a moeda é justa e, portanto, a distribuição Y é binomial cumulativa. Assim, temos $p = P(Y \geq 80) = 5,6 * 10^{-10}$, ou seja, a probabilidade de uma amostra como a observada ocorrer com uma moeda justa é de $5,6 * 10^{-10}$. Isto não implica em rejeitar tacitamente a hipótese nula mas, conforme o nível de exigência desejado, pode-se considerar que a amostra falhou claramente no teste aplicado.

Nas baterias de testes estatísticos da biblioteca TestU01 os *p-values* fora do intervalo $[10^{-3}, 1 - 10^{-3}]$ são exibidos no resultado final, mas não podem ser consideradas falhas. L'Ecuyer & Simard [2007] arbitram como *falhas claras* os *p-values* fora do intervalo $[10^{-10}, 1 - 10^{-10}]$.

Todos os geradores testados: BBS, Síndrome-Fortuna e o LRNG passaram pelas baterias de testes estatísticos. Todos os *p-values* de todos os testes estiveram no intervalo $[10^{-3}, 1 - 10^{-3}]$, não pemitindo rejeitar a hipótese nula. Isto significa que, estatisticamente, pelos testes aplicados, os valores gerados não podem ser diferenciados de valores verdadeiramente aleatórios.

5.4 Análise de Desempenho

O gerador Síndrome-Fotuna foi avaliado por meio de testes de desempenho e comparado com os geradores BBS e LRNG. Foi utilizado para essa comparação um computador com processador Intel®Pentium®Dual Core T3400 2.16Ghz com 2GB de memória RAM. Foram utilizadas cargas de 100 bytes, 1 kB, 10kB, 100kB e em intervalos de 100kB até completar 1MB de dados aleatórios. Cada gerador teve o tempo de produção dos valores aleatórios cronometrado 3 vezes para cada carga, utilizando a interface própria da biblioteca TestU01.

Os geradores Síndrome-Fortuna e LRNG tiveram sua implementação avaliada enquanto compilados dentro do kernel. Já o algoritmo BBS foi compilado em espaço de usuário e linkado com biblioteca aritmética GMP, que permite a manipulação de números de tamanho variável. O algoritmo BBS foi utilizado apenas como base de comparação de desempenho, motivo pelo qual não o implementamos dentro do kernel.

Para avaliar o impacto no desempenho dos geradores compilados dentro e fora do kernel, fizemos testes com uma implementação do Síndrome-Fortuna compilado em espaço de usuário. Os resultados foram estatisticamente indistinguíveis dos resultados quando o algoritmo estava implementado no kernel. Isto não implica, necessariamente, que o mesmo ocorreria com o algoritmo BBS mas, para os propósitos deste trabalho, consideramos satisfatório o desempenho do BBS linkado com a biblioteca GMP.

Os resultados estão sumarizados na figura 5.1.

As velocidades médias, com os respectivos desvios, estão na tabela 5.1.

Conforme esperávamos, o Síndrome-Fortuna tem desempenho inferior ao do gerador atual do Linux. Entretanto, quando comparado com o gerador BBS, que possui características formais de segurança semelhantes ao Síndrome-Fortuna, o desempenho do gerador é 6,3 vezes superior.

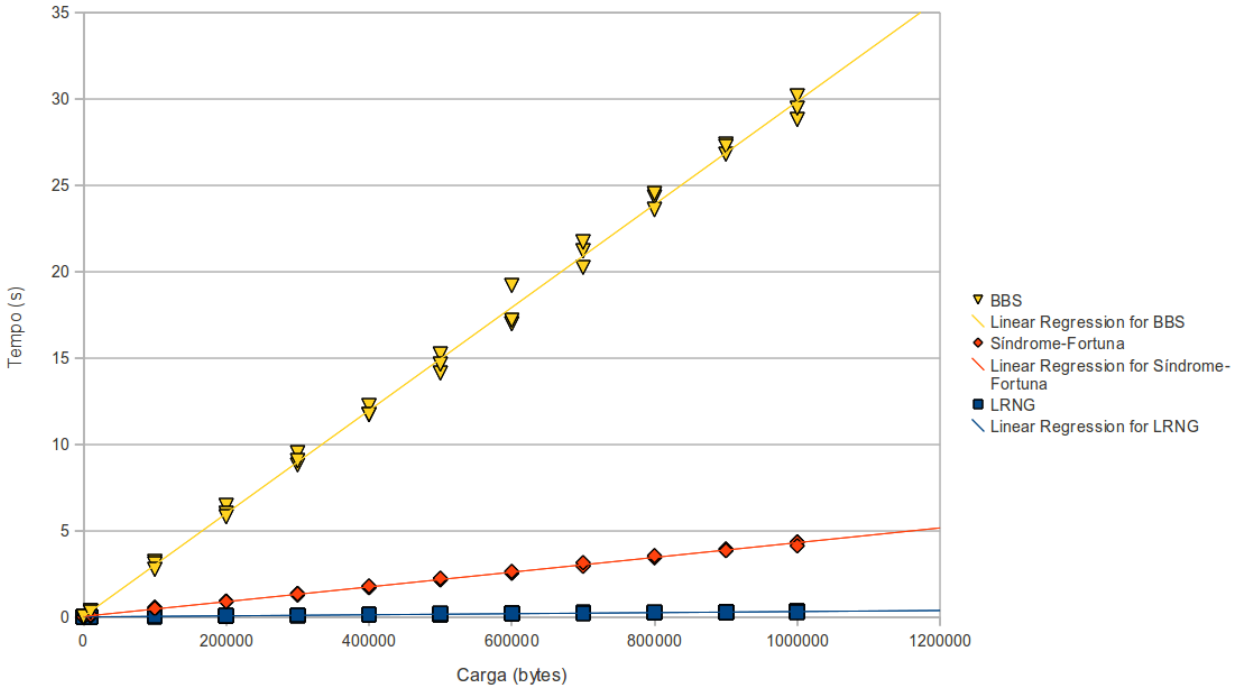

Figura 5.1. Desempenho dos geradores aleatórios Linux (LRNG), Síndrome-Fortuna e Blum-Blum-Shub (BBS).

Tabela 5.1. Desempenho medido nos geradores LRNG, Síndrome-Fortuna e BBS.

Gerador	**Velocidade (em kB/s)**
LRNG	2664,0 ± 818,9
Síndrome-Fortuna	197,1 ± 58,2
BBS	31,4 ± 6,4

Capítulo 6

Considerações Finais

Durante este trabalho foram estudados diversos tipos de geradores de números aleatórios, estatísticos e criptográficos. Foi feita uma investigação detalhada do gerador implementado no Kernel Linux, e foi proposto e implementado um novo gerador, baseado em duas construções existentes. Apresentamos neste capítulo as conclusões deste trabalho e as sugestões para trabalhos futuros.

6.1 Conclusões

Os geradores de números aleatórios são parte importante do complexo conjunto de protocolos e aplicativos responsáveis por garantir a segurança da informação e dos sistemas computacionais. Em um cenário de mudanças rápidas, como o atual, em que a computação alcança lugares inexplorados e novos usuários, o arcabouço de aplicações criptográficas deve se adaptar para fornecer a segurança desejada.

No caso dos geradores de números aleatórios, isto implica em adaptar-se a novas fontes de entropia, novas formas de operação. Não é difícil imaginar cenários onde teclado, mouse e disco rígido estejam cada vez menos presentes, impondo restrições à aleatoriedade dos sistemas. A estratégia implementada para a coleta de entropia no gerador Síndrome-Fortuna é consoante com esta necessidade, uma vez que não realiza a estimativa de entropia e, por isso, pode ser utilizada com qualquer tipo de fonte de aleatoriedade, aumentando a gama de opções disponíveis.

Inversamente, o gerador de números aleatórios do Linux enfrenta, como observamos em sua operação, uma dificuldade para se adaptar a novas fontes de entropia, já que todas devem passar por uma heurística que, caso produza valores imprecisos, pode levar o gerador a estados de baixa entropia. Em uma execução do Linux Ubuntu 9.10, pudemos observar a adição de entropia apenas pelo teclado, mouse e disco rígido,

enquanto uma série de, possivelmente, boas fontes de entropia estavam disponíveis, como placas de rede wireless, interrupções de software, etc.

Quanto à geração dos valores aleatórios, per si, a implementação proposta aplica sólidos princípios matemáticos, derivados da teoria de códigos-corretores de erros, e pode ser considerada tão difícil quanto o problema NP-completo da decodificação do vetor síndrome.

O algoritmo proposto pode ser considerado para utilização em diversos cenários, especialmente em servidores headless, ou em cloud-computing, cenários onde os dispositivos comuns de aleatoriedade não estão presentes e a memória não representa uma restrição à implementação da tabela binomial utilizada pelo algoritmo. Acreditamos que o gerador implementa uma boa relação custo/benefício, fornecendo bits com boas propriedades estatísticas, bom nível de segurança e em velocidades razoáveis.

6.2 Trabalhos Futuros

Acreditamos que o desempenho em tempo e memória do gerador Síndrome-Fortuna pode ser melhorado consideravelmente alterando a função geradora "A", mostrada na figura 4.4. Observamos que grande parte do processamento e, claramente, a maior parte do custo de memória estão relacionados à forma de se obter o vetor x de tamanho n e peso w a partir de um índice aleatório i. A heurística proposta pode apresentar bons resultados, mas os parâmetros específicos do gerador devem ser estudados, de forma a minimizar ao máximo as colisões geradas. Paralelamente, deve ser implementada uma estratégia para detecção e tratamento de colisões.

A estratégia de coleta de entropia permite, como mostramos, a utilização de novas fontes de aleatoriedade, independente de quaisquer estudos detalhados em relação à entropia destas. Uma extensão natural deste trabalho é identificar estas fontes e promover a interligação destas com o sistema de coleta de entropia.

Referências Bibliográficas

Barak, B. & Halevi, S. (2005). A model and architecture for pseudo-random generation with applications to /dev/random. In Atluri, V.; Meadows, C. & Juels, A., editores, *ACM Conference on Computer and Communications Security*, pp. 203--212. ACM.

Barker, E. & Kelsey, J. (2006). Recommendation for random number generation using deterministic random bit generators. Special Publication SP 800-90, National Institute of Standards and Technology (NIST).

Berlekamp, E.; McEliece, R. & Van Tilborg, H. (1978). On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384--386.

Blum, L.; Blum, M. & Shub, M. (1983). Comparison of two pseudo-random number generators. In *Advances in Cryptology–Proceedings of Crypto*, volume 82, pp. 61--78.

Chabaud, F. (1994). On the security of some cryptosystems based on error-correcting codes. In *EUROCRYPT*, pp. 131--139.

Ferguson, N. & Schneier, B. (2003). *Practical Cryptography*. Wiley & Sons.

Fischer, J.-B. & Stern, J. (1996). An efficient pseudo-random generator provably as secure as syndrome decoding. In *EUROCRYPT*, pp. 245--255.

Francillon, A. & Castelluccia, C. (2007). TinyRNG: A cryptographic random number generator for wireless sensors network nodes. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on*, pp. 1--7. Citeseer.

Goldberg, I. & Wagner, D. (1996). Randomness and the Netscape browser. *Dr. Dobb's Journal of Software Tools*, 21(1):66, 68--70.

Goldreich, O. (2001). *Foundations of Cryptography. Volume I: Basic Tools*. Cambridge University Press, Cambridge, England.

Goldreich, O.; Krawczyk, H. & Michael, L. (1993). On the existence of pseudorandom generators. *SIAM J. Computing*, 22(6):1163--1175.

Gutterman, Z.; Pinkas, B. & Reinman, T. (2006). Analysis of the linux random number generator. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 371--385, Washington, DC, USA. IEEE Computer Society.

Impagliazzo, R.; Levin, L. & Luby, M. (1989). Pseudorandom generation from one-way functions. In *Proc. 21st Ann. ACM Symp. on Theory of Computing*, pp. 12--24.

Kelsey, J.; Schneier, B.; Wagner, D. & Hall, C. (1998). Cryptanalytic attacks on pseudorandom number generators. In Vaudenay, S., editor, *FSE*, volume 1372 of *Lecture Notes in Computer Science*, pp. 168--188. Springer.

Krhovjak, J.; Matyas, V. & Zizkovsky, J. (2009). Generating random and pseudorandom sequences in mobile devices. In Schmidt, A. U. & Lian, S., editores, *MobiSec*, volume 17 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 122--133. Springer.

L'Ecuyer, P. & Simard, R. (2007). Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4).

Lewis, T. G. & Payne, W. H. (1973). Generalized feedback shift register pseudorandom number algorithm. *J. ACM*, 20(3):456--468.

Markettos, A. T. & Moore, S. W. (2009). The frequency injection attack on ring-oscillator-based true random number generators. In Clavier, C. & Gaj, K., editores, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pp. 317--331. Springer.

Matsumoto, M. & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3--30.

Shamir, A. (1981). On the generation of cryptographically strong pseudo-random sequences. In Even, S. & Kariv, O., editores, *ICALP*, volume 115 of *Lecture Notes in Computer Science*, pp. 544--550. Springer.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell Syst. Technical Jrnl.*, 27:379--423, 623--656.

Vanstone, S.; van Oorschot, P. & Menezes, A. (1997). Handbook of applied cryptography. *Discrete Mathematics and its Applications, CRC Press.*

Vazirani, U. V. & Vazirani, V. V. (1984). Efficient and secure pseudo-random number generation (extended abstract). In *FOCS*, pp. 458--463. IEEE.

von Neumann, J. (1951). Various techniques used in connection with random digits. *J. Research Nat. Bur. Stand., Appl. Math. Series*, 12:36--38.