ESTUDIO Y ANÁLISIS DE LA SEGURIDAD DEL ALGORITMO HOTP COMO GENERADOR DE CONTRASEÑAS DE UN SOLO USO

SERGIO GARCÍA CALDERÓN

Proyecto de grado para optar al título de
Magíster en Ingeniería de Sistemas y Computación

Asesor
MILTON QUIROGA

UNIVERSIDAD DE LOS ANDES
DEPARTAMENTO DE INGENIERIA DE SISTEMAS Y COMPUTACION
MAGISTER EN INGENIERIA DE SISTEMAS Y COMPUTACION
BOGOTA D.C.
2005

TABLA DE CONTENIDO

LISTADO DE FIGURAS

LISTADO DE TABLAS

1. INTRODUCCIÓN

Hoy en día es importante asegurar la identidad de las personas que acceden a nuestros sistemas. La identificación y autenticación son la clave de la mayoría de los sistemas de control de acceso. Identificación es el acto mediante el cual, un usuario demuestra su identidad ante un sistema, usualmente en la forma de un identificador o nombre de usuario. Autenticación es la verificación de que la identidad que demuestra el usuario es válida; esto se logra por medio del envío una contraseña hacia el sistema. [4]

El uso de un nombre de usuario y una contraseña es la forma más común de identificación/autenticación. Desafortunadamente, el esquema de contraseñas es uno de los más débiles ya que es fácilmente violable haciendo uso de múltiples tipos de ataques.

Por ejemplo, las contraseñas son susceptibles a ataques de fuerza bruta cuando son definidas con poca complejidad o tamaño, también son vulnerables a ataques de intercepción cuando son intercambiadas a través de medios inseguros (no cifrados) o a ataques activos que buscan vulnerar aquellos sistemas donde estas se encuentran almacenadas.

Pero uno de los aspectos más críticos y que brinda la vulnerabilidad más alta en el uso de contraseñas es el factor humano. Los usuarios pueden decir o escribir sus contraseñas, haciéndolas susceptibles a que sean robadas o usadas por otros.

En un caso ideal, las contraseñas deberían ser usadas una única vez. A esta técnica se le conoce como “contraseñas de un solo uso”. Estas proveen máxima seguridad ya que a partir de la contraseña actual es estadísticamente improbable que se derive una nueva contraseña.

Realmente en el ambiente informático, la autenticación es la clave para otros procedimientos de seguridad. Por esta razón, se requiere dar un paso más en la seguridad y en la forma en como se autentican los usuarios, dado a que el uso de las contraseñas genera tantos problemas desde la administración así como la violación de las mismas ya que estas son fácilmente conjeturables y/o olvidables.

Para dar solución a esta problemática, aparece el doble-factor de autenticación como una alternativa al esquema violable de contraseñas. Este es un proceso de seguridad que confirma la identidad del usuario haciendo uso de dos factores distintivos. El doble-factor de autenticación no recae únicamente en "algo conocido" por el usuario (PIN o contraseña), sino que añade "algo que tiene" (Token, tarjeta inteligente, etc.). Este factor agregado, por lo general, es un dispositivo físico o alguna parte del cuerpo del usuario, como la huella digital. [2]

Pero el uso de doble-factor de autenticación aún se encuentra limitado tanto en escala como en alcance por diversas razones, por ejemplo, la poca interoperabilidad entre distribuidores de hardware y software ha sido un factor limitante para la dispersión de tecnología como el doble factor de autenticación. Particularmente, la ausencia de especificaciones "abiertas" ha llevado a soluciones donde tanto el hardware como el software se encuentran ligados a tecnología propietaria o comercial, resultando en soluciones de alto costo, pobre adopción y alta limitación en innovación. [5]

Como una solución a esta problemática, se presenta el algoritmo HOTP[1] (Hmac Based One-Time Password) propuesto por la comunidad OATH (Open Authentication) como una alternativa "abierta" para la generación de contraseñas de un solo uso (OTP[2]) que puede ser implementada por cualquier distribuidor de hardware o desarrollador de software para la creación de dispositivos de hardware interoperables con agentes de software.

[1] HOTP: Hmac base One-time password algorithm

El algoritmo es basado en eventos, motivo por el cual puede ser embebido en dispositivos de alto volumen como son tarjetas inteligentes (Java Smart Cards), tokens USB y SIM cards en celulares y GSM con el objetivo de garantizar un doble-factor de autenticación en donde se utilice "algo que se conoce" como un PIN y "algo que se tiene" el dispositivo con el generador de valores de HOTP. [5]

El algoritmo HOTP actualmente propuesto por la OATH se encuentra en estudio. Se encuentra publicado como borrador (Internet Draft) en la IETF[3] con fecha de vencimiento mayo de 2005. Al momento de escribir estas líneas, no existe un estudio formal sobre la seguridad y aleatoriedad del algoritmo.

La seguridad del algoritmo HOTP se encuentra relacionada a múltiples aspectos. Una de las fortalezas más grandes del algoritmo es que este hace uso de funciones de dispersión (hash del tipo HMAC-SHA1) para la generación de las contraseñas de un solo uso. Dentro de las propiedades de toda función de dispersión está definida que estas deben ser irreversibles dentro de un contexto específico, asegurando así que nadie podrá obtener el secreto a partir de una contraseña de un solo uso interceptada.

La seguridad del algoritmo también esta ligada al protocolo de autenticación con el cual se implemente. El protocolo propuesto implementado sobre una arquitectura de tarjetas inteligentes o SIM Cards, cumple todos los requerimientos de seguridad especificados en el diseño del algoritmo.

Mediante el uso de un conjunto de pruebas estadísticas se puede cuantificar la aleatoriedad de los valores generados por el algoritmo en comparación a otros valores catalogados como aleatorios como son: los resultados obtenidos de la descomposición de un isótopo radioactivo (cantidad física), raíz de dos, raíz de tres, Pi con un millón de dígitos, los

[2] OTP: One-time password
[3] IETF: Internet Engineering Task Force

valores generados por el RNG de Java y los valores generados por el autenticador de RSA Security SecurID.

1.1. OBJETIVOS

1.1.1. Objetivo General

Estudiar y analizar la seguridad del algoritmo HOTP[4] para la generación de contraseñas de un solo uso haciendo uso de pruebas estadísticas para verificar la aleatoriedad de los valores generados.

1.1.2. Objetivos Específicos

- Realizar un estudio, y análisis criptográfico – estadístico del algoritmo HOTP.
- Comparar estadísticamente la aleatoriedad del algoritmo HOTP con respecto a las propiedades matemáticas de Pi, raíz cuadrada de dos, raíz cuadrada de tres, los valores generador por una fuente nuclear (descomposición de un isótopo radioactivo) y los valores generados por el dispositivo de autenticación comercial de RSA Security haciendo uso de un conjunto de pruebas estadísticas estandarizadas por el NIST[5] para verificar aleatoriedad.
- Analizar diferentes tipos de alternativas para la implementación segura del algoritmo HOTP.
- Estudiar, analizar e implementar pruebas estadísticas estandarizadas por el NIST para la medición de aleatoriedad. Las pruebas incluidas son:

    - Prueba de Frecuencia (Frequency Monobit Test)
    - Prueba de Frecuencia en Bloques (Frequency within a Block)

[4] HOTP: Hmac base One-time password algorithm

- Prueba de Cadenas (Runs Test)
- Prueba de Cadenas en Bloques (Longest Run of Ones in a Block Test)
- Prueba de Complejidad Lienal (Linear Complexity Test)
- Prueba Serial (Serials Test)
- Prueba de Emparejamiento Usando Plantillas que se Sobreponen (Non-overlapping Template Matching Test)
- Prueba de Emparejamiento Usando Plantillas que se no se Sobreponen (Overlapping Template Matching Test)
- Prueba de Acercamiento a la Entropía (Approximate Entropy Test)
- Prueba de Sumas Acumulativas (Cumulative Sums (Cusums) Test)

1.2. JUSTIFICACIÓN

El uso de un nombre de usuario y una contraseña es la forma más común de identificación/autenticación hoy en día. Desafortunadamente, el esquema de contraseñas es uno de los más débiles ya que es fácilmente violable haciendo uso de múltiples tipos de ataques ya que estas son fácilmente conjeturables u olvidables.

Por esta razón, se requiere dar un paso más en la seguridad y en la forma en se autentican los usuarios, dado a que el uso de las contraseñas genera tantos problemas desde la administración así como de la violación de las mismas; El doble-factor de autenticación aparece como una alternativa al esquema violable de contraseñas. Este es un proceso de seguridad que confirma la identidad del usuario por medio de dos factores distintivos. [2]

Pero el uso de doble-factor de autenticación aún se encuentra limitado tanto en escala como en alcance por diversas razones como son la poca interoperabilidad que existe entre

[5] NIST: National Institute of Standards and Technology

diferentes distribuidores de soluciones comerciales o los elevados costos asociados a estos tipos de soluciones.

Una mejor solución, esl algoritmo HOTP (Hmac Based One-Time Password) propuesto por la comunidad OATH (Open Authentication), el cual se presenta como una alternativa novedosa y "abierta" para la generación de contraseñas de un solo uso (OTP) que puede ser implementada por cualquier distribuidor de hardware o desarrollador de software para la creación de dispositivos de hardware interoperables con agentes de software.

Este proyecto busca estudiar la seguridad de los valores o contraseñas de un solo uso generadas por el algoritmo basándose en un conjunto de 10 pruebas estadísticas que buscan comprobar la aleatoriedad con respecto a la expansión decimal de Pi con un millón de dígitos, raíz cuadrada de dos, raíz cuadrada de tres, los resultados obtenidos de la degradación de un isótopo radioactivo en una fuente nuclear, los valores obtenidos del generador de números aleatorios de Java y los valores entregados por el dispositivo de autenticación de RSA Security SecurID.

El documento se presenta en cuatro capítulos como un estudio y análisis de la seguridad del algoritmo evaluando el diseño seguro del mismo y las propiedades de aleatoriedad que presentan los valores generados por él.

El capítulo dos presenta el marco teórico del documento donde se estudian las raíces del algoritmo como son los algoritmos de generación de contraseñas de un solo uso (OTP) y el doble-factor de autenticación como un esquema a tener en cuenta para la implementación en ambientes seguros del algoritmo HOTP. Se presenta el funcionamiento del algoritmo HOTP, su diseño y requerimientos de seguridad. Para finalizar el capítulo se presenta la solución comercial SecurID de RSA la cual es tomada como punto de referencia al ser una solución comercial de alto uso en el mercado actual, para validar las propiedades de seguridad ofrecidas por el algoritmo HOTP como una solución alternativa.

El capítulo tres corresponde a un estudio de la seguridad del algoritmo HOTP al analizar las propiedades criptográficas de los componentes que lo conforman.

El capítulo cuatro, como complemento a la evaluación de la seguridad del algoritmo en el capítulo anterior, busca evaluar la aleatoriedad de los valores generados por el algoritmo. La evaluación de aleatoriedad juega un papel importante en la evaluación de la seguridad de algoritmos que hacen uso de generadores de números aleatorios; los generadores de números aleatorios son un componente importante en la construcción de llaves criptográficas y otros parámetros de seguridad.

2. MARCO TEÓRICO

El algoritmo HOTP (Hmac Based One-Time Password) propuesto por la comunidad OATH (Open Authentication) se presenta como una alternativa "abierta" a las soluciones comerciales existentes hoy en día como lo es SecurID de RSA para la generación de contraseñas de un solo uso (OTP).

En el capítulo a continuación se encuentra el marco teórico del proyecto el cual esta conformado por la descripción, funcionamiento y características del algoritmo HOTP, temas como OTP (contraseñas de un solo uso) y doble-factor de autenticación cuya explicación es necesaria para la comprensión del diseño del algoritmo HOTP y su posible implementación en ambientes seguros.

2.1. OTP: ONE-TIME PASSWORDS

Los algoritmos OTP (One-time passwords) son un sistema de autenticación que a través de la generación de contraseñas de un solo uso, proporcionan acceso a sistemas y aplicaciones que requieren de protección contra ataques pasivos basados en la captura y posterior reutilización de contraseñas. [2]

Uno de los ataques más comunes en sistemas en red es el de intercepción o *sniffing*. En este tipo de ataque, una entidad (persona, programa o sistema) obtiene información de autenticación al interceptar contraseñas en tránsito por la red y posteriormente la utiliza para obtener acceso al sistema. [3]

Los sistemas OTP hacen uso de un secreto para generar una secuencia de contraseñas, cada una de un solo uso (single use). Con esto se garantiza que el secreto nunca viaja por la red durante el proceso de autenticación, protegiendo así el sistema ante ataques pasivos.

Aunque los sistemas OTP brindan protección contra ataques pasivos al subsistema de autenticación, estos no garantizan la obtención de acceso a información privada ya que no brindan protección contra otros tipos de ataques como son ataques activos o ataques de ingeniería social. [2]

2.1.1. DESCRIPCIÓN DE OTP

En el esquema de One-Time Passwords (passwords de un solo uso), la misma contraseña nunca viaja por la red. En cada conexión se usa una contraseña distinta, que una vez utilizada para autenticarse, deja de ser válida. [1]

Para emplear el sistema OTP se debe definir: [4]

1. Secreto: Valor conocido únicamente por el cliente y por el servidor de autenticación (secreto compartido) que nunca viaja por la red.
2. Llave temporal: Valor temporal cuyo tiempo de vida se encuentra asociado a la duración de una sesión.
3. Sesión: Intervalo de tiempo en el cual tiene validez una llave temporal.

Adicionalmente, existen dos entidades involucradas en la operación del sistema OTP. La primera entidad, el generador de contraseñas de un solo uso, es el encargado de producir las llaves temporales a partir del secreto y de la información proporcionada por el servidor en forma de reto (challenge). La segunda entidad, el servidor, es el encargado de transmitir el reto al cliente y verificar la validez de la contraseña de un solo uso recibida.

Adicionalmente, el servidor debe proporcionar un mecanismo para el cambio del secreto en forma segura. [2]

El proceso es el siguiente:

1. El usuario se conecta al servidor (la máquina en la que quiere abrir una sesión).
2. El servidor envía al usuario un reto.
3. El usuario calcula la respuesta al reto (el OTP), y envía de vuelta al servidor un valor generado a partir de esta información el cual es único y será usado solo esta vez.
4. El servidor por su parte calcula también la respuesta, y la compara con la que ha recibido. Si ambas coinciden, se permite el acceso al servidor.

La función que se utiliza para calcular la respuesta es tal que resulta imposible calcular un OTP a partir de los anteriores. Esto significa que, aunque se intercepte alguno de los OTP, éste no se podrá utilizar para entrar en la cuenta del usuario, ni tampoco para averiguar los OTPs que se utilizarán a continuación. [1]

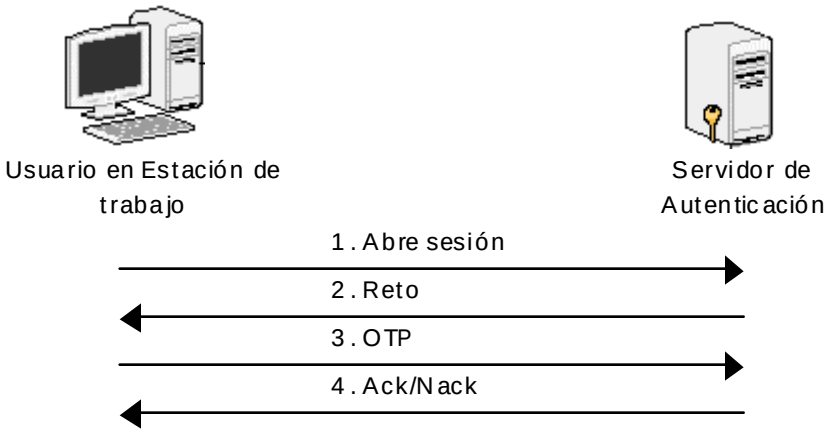


Figura 1. Proceso de autenticación OTP

2.1.2. FUNCIÓN SEGURA DE DISPERSIÓN (HASH)

La seguridad del sistema OTP se basa en la dificultad de reversibilidad de la función segura de dispersión o hash usada para la generación del valor OTP. La función debe ser manejable en un solo sentido, hacia delante (forward). Esto significa que un mismo valor de hash se puede obtener aplicando la misma función de dispersión al mismo secreto y al mismo valor inicial. Pero que a partir del valor generado es computacionalmente imposible obtener el secreto. A esta propiedad se le conoce como irreversible en sentido opuesto (backward irreversibility). [2]

Una función de dispersión (hash) es una operación que se realiza sobre un conjunto de datos de cualquier tamaño de tal forma que se obtiene como resultado otro conjunto de datos, de tamaño fijo e independiente del tamaño original que, además, tiene la propiedad de estar asociado inequivocamente a los datos iniciales, es decir, es imposible encontrar dos mensajes distintos que tengan un número de hash igual. [4]

Las funciones hash (o primitivas hash) pueden operar como: *MDC* (Modification Detection Codes) ó *MAC* (Massage Authentication Codes).

Los MDC sirven para resolver el problema de la integridad de la información, al mensaje se le aplica un MDC (una función hash) y se manda junto con el propio mensaje, al recibirlo el receptor aplica la función hash al mensaje y comprueba que sea igual al hash que se envió antes.

Es decir, se aplica un hash al mensaje M y se envía con el mensaje (M, h(M)), cuando se recibe se le aplica una vez más el hash (ya que M es público) obteniendo h'(M), si h(M)=h'(M), entonces se acepta que el mensaje se a transmitido sin alteración.

Los MAC sirven para autenticar el origen de los mensajes. Es decir, se combina el mensaje M con una clave privada K y se les aplica un hash h(M, K). Si al llegar a su destino h(M, K) se comprueba la integridad de la clave privada K, entonces se demuestra que el origen es solo el que tiene la misma clave K, probando así la autenticidad del origen del mensaje.

Las propiedades que deben de tener las primitivas de dispersión son:

1. Resistencia a la preimagen: Dada cualquier imagen, es computacionalmente imposible encontrar un mensaje x tal que h(x) = y. Otra forma como se conoce esta propiedad es que h sea de un solo sentido.
2. Resistencia a una segunda preimagen: Dado x, es computacionalmente imposible encontrar una x' tal que h(x)=h(x'). Otra forma de conocer esta propiedad es que h sea resistente a una colisión suave.
3. Resistencia a colisión: Significa que es computacionalmente imposible encontrar dos diferentes mensajes x, x' tal que h(x)=h(x'). Esta propiedad también se conoce como resistencia a colisión fuerte.

Existen varios tipos de funciones de dispersión. Entre las más conocidas se encuentran: MD4 y MD5 desarrollado por Ronald Rivest y SHA por el instituto NIST.

2.1.3. GENERACIÓN DE CONTRASEÑAS DE UN SOLO USO

El proceso de generación de contraseñas de un solo uso esta compuesto por tres pasos:

1. Iniciación: Fase donde se combinan todas las posibles entradas.
2. Procesamiento: Fase donde se aplica la función de dispersión múltiples veces.
3. Salida: Fase donde se convierte la salida de tamaño fijo generada por la función de dispersión en un valor legible por el usuario.

La fase de iniciación recibe como entrada un secreto compartido conocido por el cliente y por el servidor. Este valor debe ser mínimo de 10 caracteres de longitud para reducir el riesgo de posibles ataques de fuerza bruta. Durante la fase de iniciación, al secreto compartido se le concatena una semilla. La semilla es el valor enviado por el servidor como reto en texto claro. Este valor es el que permite a un cliente autenticarse ante el servidor.

Durante la fase de procesamiento, se producen un conjunto de contraseñas de un solo uso al aplicar la función de dispersión múltiples veces a la salida del paso inicial. El primer valor OTP a usar se genera al aplicar la función de dispersión un número "n" de veces sobre la salida de la fase de iniciación. El segundo valor OTP a usar se genera al aplicar la función de dispersión "n-1" veces sobre la salida de la fase de iniciación y así sucesivamente.

Para un atacante quien interceptó un valor OTP sería imposible generar el siguiente valor de contraseña requerido ya que para esto tendría que invertir la función de dispersión.

En la fase de salida, se toma la salida generada durante la fase de procesamiento. Esta salida siempre es de tamaño fijo y conocido. Por ejemplo: Para una función de dispersión MD5 la salida siempre será de 128 bits mientras que para una función de dispersión SHA-1 la salida tendrá un tamaño mínimo de 160 bits.

La verificación de la contraseña de un solo uso se lleva a cabo por el servidor. Este tiene almacenado el secreto compartido para cada uno de los clientes y el valor de semilla transmitido. Con esta información y un previo acuerdo de la función de dispersión a utilizar el servidor puede generar el mismo valor de OTP que el cliente para verificar correspondencia. [1]

2.2. HOTP: HMAC BASED ONE-TIME PASSWORD

A pesar del incremento en los niveles de amenaza y ataques, la gran mayoría de aplicativos y sistemas aún basan su seguridad en débiles esquemas de identificación y autenticación como política para control de acceso.

El método cada vez más violado de las contraseñas es el más usado, aún cuando ya se encuentran disponibles diferentes métodos como el de doble-factor de autenticación.

El doble factor de autenticación más que un método es un proceso de seguridad que confirma la identidad del usuario por medio de dos factores distintivos. Al requerir estos factores, los sistemas reducen el riesgo de fraudes y generan un ambiente de seguridad en las transacciones por diferentes medios (p.e. Internet). [3] [6]

Pero el uso de doble-factor de autenticación aún se encuentra limitado tanto en escala como en alcance. La poca interoperabilidad entre distribuidores de hardware y software ha sido un factor limitante para la dispersión de tecnología como el doble factor de autenticación. Particularmente, la ausencia de especificaciones "abiertas" ha llevado a soluciones donde tanto el hardware como el software se encuentran ligados a tecnología propietaria o comercial, resultando en soluciones de alto costo, de pobre adopción y limitadas en innovación.

Para que se propague el uso de doble factor de autenticación se debe embeber en dispositivos flexibles y sencillos que puedan funcionar con un alto número de aplicaciones en diferentes tipos de ambientes y arquitecturas. [5]

Contraseñas de un solo uso (OTP: One-time passwords) son la forma más sencilla y popular de un doble factor de autenticación para asegurar el control de acceso a redes y aplicaciones. [1]

El algoritmo HOTP (Hmac Based One-Time Password) propuesto por la comunidad OATH (Open Authentication) se presenta como un algoritmo abierto para la generación de contraseñas de un solo uso (OTP) que puede ser implementado por cualquier distribuidor de hardware o desarrollador de software para la creación de dispositivos de hardware interoperables con agentes de software.

El algoritmo es basado en eventos motivo por el cual puede ser embebido en dispositivos de alto volumen como son tarjetas inteligentes (Java Smart Cards), tokens USB y SIM cards GSM con el objetivo de garantizar un doble-factor de autenticación en donde se utilice "algo que se conoce" como un PIN y "algo que se tiene" el dispositivo con el generador de valores de HOTP. [5]

2.2.1. REQUERIMIENTOS DE DISEÑO DEL ALGORITMO

A continuación se presenta un conjunto de requerimientos que conllevaron al diseño del algoritmo:

1. El algoritmo debe basarse en secuencias o contadores, no tiempo: Uno de los objetivos es la implementación del algoritmo en dispositivos de alto volumen como Java Smart Cards, SIMs GSM y tokens USB.

2. El algoritmo debe ser económico de implementar en hardware al minimizar el uso de requerimientos computacionales, batería, número de botones y tamaño de la pantalla (LCD). El algoritmo debe ser implementable en todo tipo de dispositivo, desde aquellos que no soportan valores numéricos o no tienen pantalla hasta aquellos más sofisticados.

3. El tamaño del valor HOTP debe ser de longitud considerable y numérico: Dependiendo del dispositivo en el cual sea implementado el valor generado debe ser

sencillo y corto para que pueda ser leído y digitado fácilmente por un usuario a través de dispositivos con un número limitado de botones.

4. Debe existir una forma sencilla de re-sincronizar los contadores usados por el algoritmo.

5. El algoritmo debe hacer uso de un secreto compartido de no menos de 128 bits. (Recomendación: 160 bits).

Como puntos de importancia se hizo énfasis en dos temas. Facilidad de uso para el usuario final y que pueda ser implementado en soluciones de bajo costo y de pocos recursos computacionales como son tarjetas SIM y tarjetas inteligentes (Java Cards). [5]

2.2.2. DESCRIPCIÓN DEL ALGORITMO HOTP

Fundamentalmente el algoritmo se encuentra compuesto por tres pasos:

1. Generación de un HMAC-SHA1 a partir de un secreto compartido entre el generador (cliente) y el verificador (servidor) concatenado a un contador de intentos de autenticación (tanto exitosos como fallidos).
2. El truncado dinámico del valor HMAC-SHA1.
3. La conversión del valor obtenido del truncado dinámico a un valor numérico, corto de fácil legibilidad por parte del usuario.

A continuación se presenta la notación y la descripción de los fundamentos del algoritmo, la base funcional para calcular el valor HMAC-SHA-1 y el método de truncamiento para el cálculo del valor HOTP. [5]

Notación

- Una cadena se representa mediante una cadena binaria (secuencia de 1's y 0's).
- Si s es una cadena, |s| representa su longitud.
- Si n es un número, |n| representa su valor absoluto.
- Si s es una cadena, s[i] representa su i-ésimo bit. (La numeración se inicia s[0] s[1]… s[n-1] donde n = |s| es la longitud de s.)

Simbología

- **C** : (8 bytes). Valor contador. Valor que debe ser sincronizado entre el generador de HOTP (cliente) y el validador (Servidor).
- **K** : Secreto compartido entre cliente y servidor. Cada cliente debe tener un secreto compartido único.
- **T** : Umbral de número de autenticaciones fallidas.
- **s** : Parámetro de resincronización. El servidor tratará de verificar un autenticador entre "s" valores de contador consecutivos.
- **d** : Representa el número de dígitos en el valor HOTP resultante.

El algoritmo HOTP se basa en un contador incremental y una llave simétrica estática conocida únicamente por el generador y el servidor de validación. Para la generación del valor HOTP, se hace uso del algoritmo HMAC-SHA-1 definido en el RFC 2104. Debido a que el resultado del HMAC-SHA-1 es de 160 bits, se debe truncar este valor a algo que pueda ser digitado por un usuario.

$$\mathbf{HOTP\ (K,\ C) = Truncar(\ HMAC\text{-}SHA\text{-}1\ (K,\ C)\ )}$$

Donde: Truncar representa la función que convierte el valor HMAC-SHA-1 al valor HOTP.

Generación del Valor HOTP (3 pasos):

1. Generar valor HMAC-SHA-1.

```
Sea HS = HMAC-SHA-1 (K, C)    // HS cadena de 20 bytes
```

2. Generar una cadena de 4 bytes (Truncado dinámico)

```
Sea Sbits = DT(HS)  // DT retorna una cadena de 31 bits.
```

donde:

```
DT(Cadena) // Cadena (binaria) = Cadena[0] … Cadena[19]
    - Sea Offsetbits los 4 bits de bajo orden de Cadena[19]
    - Sea Offset = StToNum(Offsetbits)  // 0 <= Offset <=15
    - Sea P = Cadena[Offset] … Cadena[Offset + 3]
```

3. Calcular el valor HOTP.

- Sea Snum = StToNum (S) // Convierte S a un número en 0 … 2^{31} - 1
- Retornar D = Snum mod 10^{digito} // D en un número en el rango 0 … $10^{d - 1}$

La función de truncado esta conformada por los pasos 2 y 3 (Truncado dinámico y la reducción módulo 10^d).

El paso 2 busca extraer un código binario de 4 bytes del resultado de la función HMAC-SHA-1. El bit más significativo de P es enmascarado para evitar confusión entre cálculos computacionales de signo. El remover el signo evita cualquier ambigüedad.

El paso 3 busca a través de una reducción módulo 10^d retornar el valor HOTP de longitud deseable.

El valor "digito" (d) especifica el tamaño del valor HOTP deseado. Este valor puede ser 6, 7 o 8 dependiendo de los requerimientos de seguridad deseados. El valor 6 es el valor mínimo requerido de seguridad. Se recomienda trabajar siempre con una valor de 8. [5]

2.2.3. CARACTERÍSTICAS DE SEGURIDAD

Cualquier algoritmo de contraseñas de un solo uso es tan seguro como la aplicación y el *protocolo de autenticación que lo implemente***.**

El algoritmo HOTP debe ser implementado a través de un protocolo de autenticación (P) que cumpla las siguientes premisas: [5]

1. El protocolo P debe ser doble-factor. Por lo tanto debe incluir algo que se conoce como un código secreto, contraseña o PIN y algo que se tiene, como puede ser un token o una tarjeta inteligente.

2. El protocolo P no debe ser vulnerable a ataques de fuerza bruta. Se debe implementar algún esquema de bloqueo ante un número de intentos fallidos de autenticación.

3. El protocolo se debe implementar a través de medios seguros y probados para evitar riesgos asociados a la transmisión de datos sensibles a través de redes públicas.

Para la validación, tanto el cliente como el servidor calculan el valor de HOTP esperado. En caso de que el valor entregado por el cliente (Hardware o Software) no coincida con el valor de HOTP que espera el servidor se debe iniciar el proceso de re-sincronización.

En caso de que el proceso de re-sincronización falle se permiten dos intentos más de autenticación antes de que el servidor bloquee la cuenta del usuario.

La seguridad del algoritmo HOTP al igual que cualquier implementación de un algoritmo de generación de contraseñas de un solo uso se basa en la propiedad de las funciones de dispersión de irreversibilidad en donde es casi imposible obtener el secreto compartido a partir de aplicar la función de dispersión inversa sobre uno de los valores entregados por esta.

Con esto en mente se puede deducir que la mejor estrategia de ataque al algoritmo HOTP es hacer uso de un ataque de fuerza bruta en donde se enumeren y prueben todos los posibles valores.

La seguridad del algoritmo contra ataques de fuerza bruta se puede describir mediante la siguiente fórmula: [5]

$$P_{sec} = (s * v) / 10^{d}$$

Donde: P_{sec} = Probabilidad de éxito del atacante.

s = Tamaño de la ventana de re-sincronización.

v = Número de autenticaciones permitidas.

d = Número de dígitos en valor HOTP generado.

2.2.4. RE-SINCRONIZACIÓN

Existen dos entidades involucradas en la operación del algoritmo HOTP. La primera entidad, el generador de valores HOTP (cliente), y la segunda entidad, el verificador (servidor), el encargado de la validez del valor de HOTP generado por el cliente.

Tanto el cliente como el servidor implementan el algoritmo HOTP. El cliente para emitir el valor como un segundo factor de autenticación y el servidor para validarlo. Para que esto

sea posible, tanto el cliente como el servidor deben estar sincronizados, o sea que el valor de HOTP emitido por el cliente coincida con el valor de HOTP esperado por el servidor. [2]

Debido a que el valor de HOTP depende no solo del secreto compartido si no también de un contador de autenticaciones se puede presentar una desincronización entre el contador que tiene el cliente y el contador almacenado en el servidor.

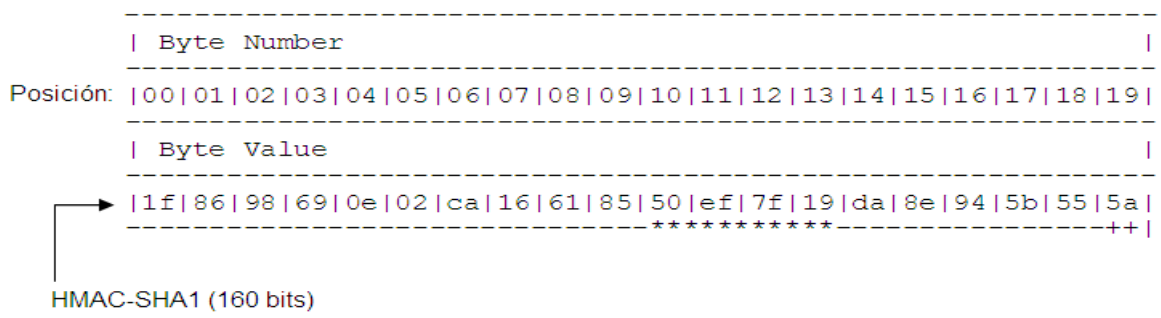
Esto se puede presentar cuando el cliente ha generado valores de HOTP en momentos en los cuales no hay conectividad con el servidor de autenticación. Como tanto el cliente como el servidor generan nuevos valores de HOTP cada vez que se hace un intento de autenticación (ya sea exitoso y fallido) es posible que el valor de contador del cliente difiera del almacenado en el servidor.

Para solucionar el problema de la falta de sincronismo. HOTP implementa un método de re-sincronización donde el servidor ante un intento fallido de autenticación general una ventana deslizante de "*n*" valores de HOTP y verifica cada uno de estos contra el valor emitido por el cliente. Si el valor emitido por el cliente se encuentra dentro de la ventana de valores de HOTP, el servidor modifica su contador de intentos por el número del contador con el cual se generó el valor de HOTP que correspondió al generado por el cliente. Luego procede a autorizar la autenticación del cliente.

En caso de que el valor emitido por el cliente no se encuentre en la ventana deslizante se incrementa el contador de autenticaciones fallidas en 1. Después de 3 intentos fallidos de autenticación se bloquea la cuenta del usuario. Esto con el objetivo de proteger el sistema ante ataques de fuerza bruta o de negación de servicio. [5]

2.2.5. EJEMPLO

El siguiente código ilustra el resultado HMAC-SHA1 de un secreto "x" dado un cierto valor de contador de autenticaciones: [5]



Paso 1: Generar valor HMAC-SHA1:

1f|86|98|69|0e|02|ca|16|61|85|50|ef|7f|19|da|8e|94|5b|55|5a

Paso 2: Truncado dinámico

Offsetbits = los 4 bits de bajo orden de Cadena[19]
Cadena[19] = 5a = 01011010 donde los 4 bits de bajo orden son: 1010
Offset = StToNum(1010) = 10 (0xa)
P = Cadena[Offset]... [Offset+3] = 0 x 50 ef 7f 19
0 x 50 ef 7f 19 = 0100001110111101111111100011001

Paso 3: Calcular el valor HOTP

Snum = StToNUm(S), S = 0100001110111101111111100011001 = 1357872921
HOTP = 1357872921 mod 10^6 = **872921** (para d = 6)

Ejemplo 1: Funcionamiento algoritmo HOTP[6]

[6] Ejemplo: Extraído de RFC: D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen. "HOTP: An HMAC-based One Time Password Algorithm". RFC – Internet Draft, Octubre de 2004. http://ietfreport.isoc.org/ids/draft-mraihi-oath-hmac-otp-01.txt

La probabilidad de que un atacante exitosamente adivine el valor de HOTP para el presente caso es de:

d = 6
s = 5 (Número máximo de reintentos de re-sincronización).
v = 3 (Número máximo de autenticaciones fallidas ante el servidor).

$$P_{sec} = (s * v) / 10^{d} = (5 * 3) / 10^{6} = 0,000015 = 0,0015\%$$

2.2.6. ESCENARIOS DE IMPLEMENTACIÓN

El algoritmo será implementado en un ambiente de celulares (simulador) y de tarjetas inteligentes donde el generador de valores de HOTP se encontrara embebido en la SIM o en el chip la tarjeta inteligente. El verificador se encontrará corriendo como un aplicativo en el servidor.

A continuación se describe el funcionamiento del protocolo implementado en un *celular*:

1. El usuario posee un celular cuya SIM card tiene instalado el algoritmo HOTP. En el menú del aplicativo digita su número de PIN (Valor conocido: primer factor de autenticación). Éste valor es verificado por el programa en la SIM Card.

2. La SIM Card procesa el valor de PIN enviado. En caso de que el PIN sea incorrecto se solicita al usuario ingresar el PIN nuevamente. Después de "n" intentos fallidos se bloquea y la SIM debe ser desbloqueda por el administrador del sistema. Si el valor es válido la SIM automáticamente inicia el proceso de generación del valor HOTP.

3. La SIM procesa y calcula el valor de HOTP. Este valor es desplegado por el display del celular al usuario. El usuario lee el valor y lo utiliza para autenticarse ante el sistema.

   Una vez la SIM emite un valor de HOTP incrementa su contador de número de autenticaciones para la generación del siguiente valor de HOTP.

4. El sistema valida las credenciales del usuario. El pin del usuario y el valor HOTP (doble-factor de autenticación).

   En caso de que el valor de HOTP no sea válido se inicia el proceso de re-sincronización. Si este proceso falla, y adicionalmente se cumple el umbral reintentos fallidos de conexión, se procede a bloquear la cuenta del usuario.

   En caso de que las credenciales coincidan o la re-sincronización sea exitosa se autoriza la autenticación del usuario al aplicativo.

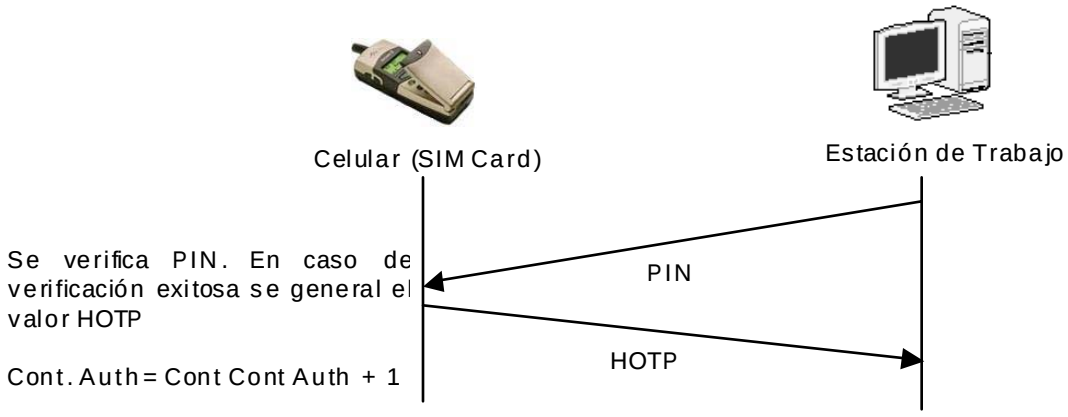


Figura 2. Proceso de autenticación HOTP - Escenario perfecto

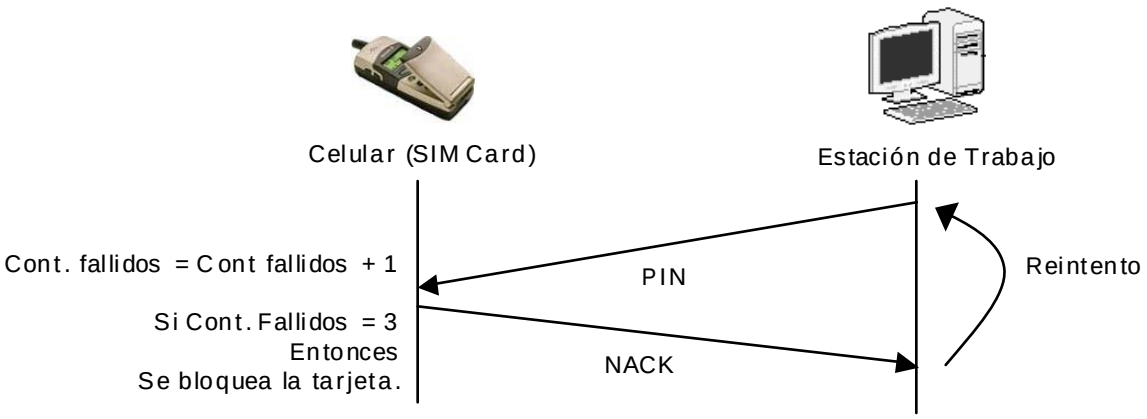


Figura 3. Proceso de autenticación HOTP – Esquema de PIN inválido

A continuación se describe el funcionamiento del protocolo implementado en un esquema de *tarjetas inteligentes*:

1. El usuario inserta su tarjeta inteligente y digita su número de PIN (Valor conocido: primer factor de autenticación) en la ventana de ingreso de PIN. Éste valor es enviado a la tarjeta inteligente.

2. La tarjeta inteligente procesa el valor de PIN enviado. En caso de que el PIN sea incorrecto se solicita al usuario ingresar el PIN nuevamente. Después de "n" intentos fallidos se bloquea la tarjeta y esta debe ser desbloqueda por el administrador del sistema. Si el valor es válido la tarjeta responde con un mensaje de aceptación del PIN.

3. Una vez la estación recibe la señal de aceptación de PIN por parte de la tarjeta, emite un comando a la tarjeta solicitando el valor HOTP (Valor generado: segundo factor de autenticación).

4. La tarjeta recibe la solicitud, la procesa y calcula el valor de HOTP. Este envía el valor a la estación de trabajo. El usuario lee el valor y lo utiliza para autenticarse ante el servidor. (Nombre de usuario valor de HOTP entregado por la tarjeta).

   Una vez la tarjeta emite un valor de HOTP incrementa su contador de número de autenticaciones para la generación del siguiente valor de HOTP.

5. El servidor valida las credenciales del usuario. El nombre de usuario y el valor HOTP.

   En caso de que el valor de HOTP no sea válido se inicia el proceso de re-sincronización. Si este proceso falla, y adicionalmente se cumple el umbral reintentos fallidos de conexión, se procede a bloquear la cuenta del usuario.

   En caso de que las credenciales coincidan o la re-sincronización sea exitosa se autoriza la autenticación del usuario al aplicativo.

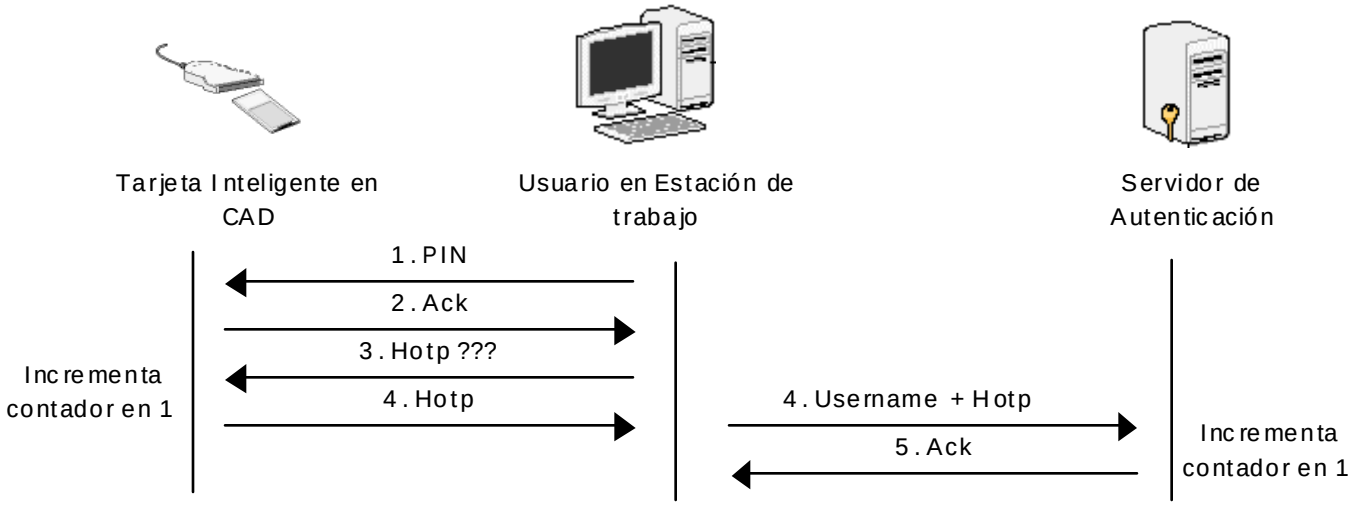


Figura 2. Proceso de autenticación HOTP - Escenario perfecto

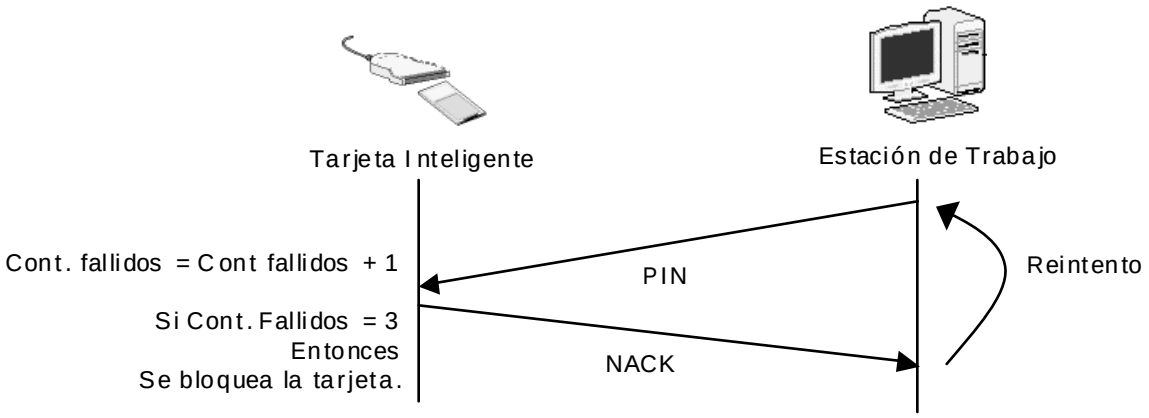


Figura 3. Proceso de autenticación HOTP – Esquema de PIN inválido

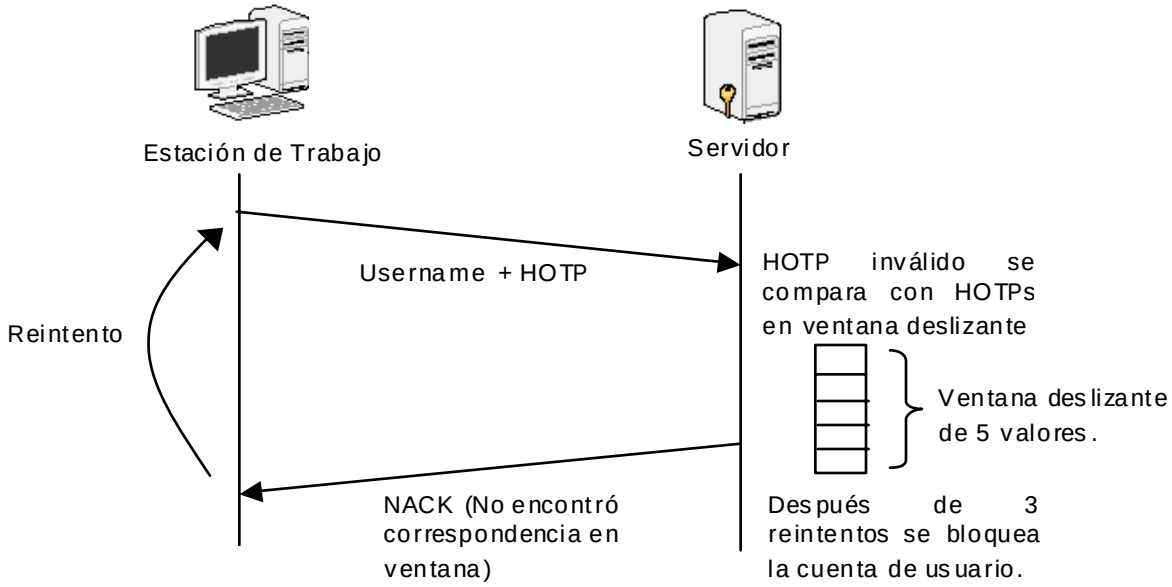


Figura 4. Proceso de autenticación HOTP – Esquema de HOTP inválido

2.3. RSA SECURID

RSA SecurID es una solución propietaria de la compañía RSA Security y por lo tanto su funcionamiento es más o menos el de una caja negra donde solo se conoce el resultado pero no el proceso[7]. [8]

- La gestión de identidades y accesos ayuda a optimizar los procesos del negocio y a construir relaciones en línea con clientes, socios y empleados
- El acceso seguro móvil y remoto ofrece diversas opciones de autenticación que permiten el acceso seguro a la red desde fuera del firewall.
- El acceso seguro en la empresa permite controlar el acceso seguro a la información vital del negocio dentro de la empresa siguiendo sencillos procedimientos de conexión, como por ejemplo SSO.
- Transacciones seguras - Una tecnología ínter plataforma que verifica el remitente y la vía de transmisión de todas las comunicaciones del negocio, y asegura que estos mensajes están protegidos.

A través de la solución SecurID, RSA ofrece doble-factor de autenticación basándose en algo que se conoce (PIN) y algo que se posee (valor proporcionado por un Autenticador).

Aun que el algoritmo de generación de contraseñas de un solo uso de RSA es privado. Como la arquitectura que compone la solución si es conocida. La arquitectura esta compuesta autenticadores, agentes de autenticación y el administrador de autenticación. [9]

- **Los autenticadores:** Dispositivos físicos portátiles (hardware) o pequeños programas (software) que generan claves dinámicas que funcionan como segundo

[7] RSA SecurID no se encuentra patentado dentro del departamento de patentes y marcas de los Estados Unidos indicando que su tecnología es completamente privada y secreta. (http://www.uspto.gov)

factor de autentificación del usuario, cada vez que este quiera acceder a un sistema de información.

Figura 5. Autenticador SecurID (Token)

- **Agentes de Autenticación:** Programas (software) que permiten el ingreso del passcode (PIN + valor generado por el autenticador) y envían la información al administrador de autenticación de RSA. Actualmente RSA security ha creado agentes para Windows, Internet Information Server, Apache, Linux/Unix.

- **Administrador de Autenticación:** Programa que se encarga de validar la autenticación del usuario a través del passcode emitido en un cierto instante de tiempo. Los autenticadores generan claves nuevas cada minuto. Estas claves son válidas durante el instante de tiempo antes de que se genere la siguiente clave.

3. ANÁLISIS DE SEGURIDAD DEL ALGORITMO HOTP

Al tratarse de una alternativa novedosa, como lo es el algoritmo HOTP, se debe estudiar la seguridad del algoritmo analizando su diseño, implementación y propiedades de aleatoriedad de las contraseñas generadas por el mismo.

La seguridad del algoritmo HOTP depende no solo del protocolo de autenticación sobre el cual sea implementado. Este depende de dos propiedades fundamentalmente las cuales serán discutidas en éste y el siguiente capítulo:

1. Las fortalezas asociadas a HMAC-SHA-1.
2. Las propiedades de aleatoriedad del generador de contraseñas de un solo uso del algoritmo.

3.1.1. HMAC-SHA1

Los códigos HMAC[8] se crearon para verificar la integridad de la información transmitida o almacenada en medios no seguros. La integridad se verifica mediante un mecanismo que permite comparar contra algo que se considera válido.

Para esto se estandarizó un procedimiento basado en una clave secreta usualmente llamado Código de Autenticación de Mensajes (MAC). El empleo típico de estos mecanismos es a través de dos partes, un remitente y un receptor, que comparten una clave secreta para validar la información transmitida entre ellas.

[8] HMAC: Hashing for Message Authentication Codes

El remitente calcula el código HMAC de los datos originales y envía los datos originales y el código HMAC como un solo mensaje. El receptor vuelve a calcular el código HMAC del mensaje recibido y comprueba si el código calculado coincide con el transmitido. Cualquier cambio realizado en los datos o en el código HMAC ocasionará un desajuste, ya que es necesario conocer la clave secreta para cambiar el mensaje y reproducir el código HMAC correcto. Por tanto, si los códigos coinciden, se autenticará el mensaje. [15]

HMAC propone el empleo de criptografía aplicada a funciones de dispersión (Hash). El RFC 2104, estandariza el empleo de HMAC con las funciones Hash definidas como MD5[9] y SHA-1[10]. El presente documento hace uso de la función de dispersión SHA-1 para la implementación del algoritmo HOTP. [15] [16]

La función de Hash (H) empleada en el HMAC se encargará de comprimir un texto de longitud finita por medio de iteraciones de una función de compresión básica sobre bloques de datos (64 Bytes), y una clave secreta (K); y por medio de ambas se obtendrá un resumen de longitud fija (L), que será de 20 Byte para SHA-1 (160 Bits). [15]

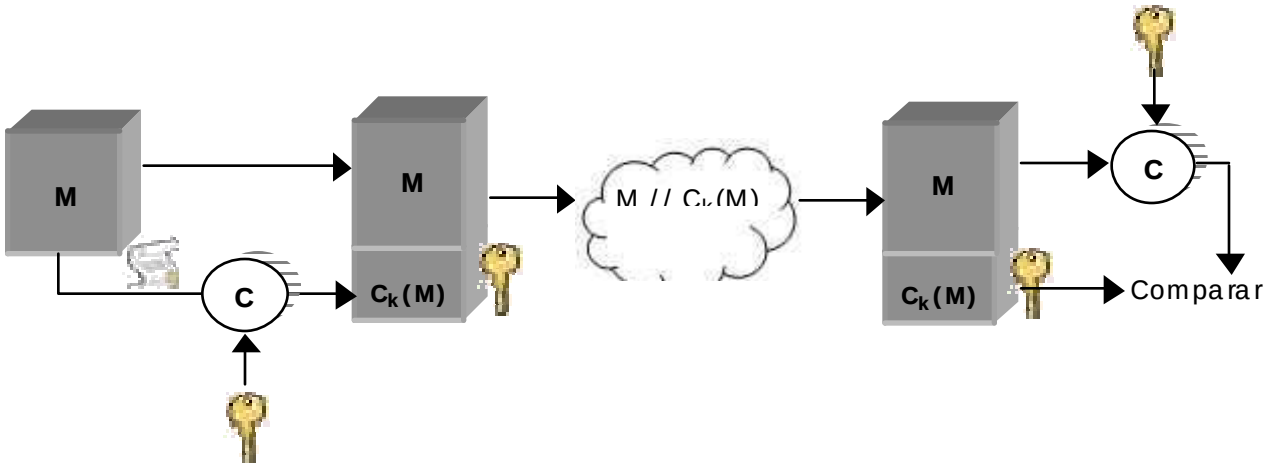


Figura 6. Código de Autenticación de Mensaje (MAC)

[9] MD5: Message Digest V. 5.0 (RFC 1321)

La seguridad de las funciones de Hash como un aporte a la seguridad de HMAC radica en su propiedad de "Una vía" (One-way), ya que no es posible a través del resumen de salida obtener el texto de entrada. También resultará computacionalmente imposible obtener un valor de salida igual a través de otro valor de entrada, como así tampoco desde un valor de salida ya calculado, obtener otro valor de entrada diferente al verdadero. [4]

HMAC hace uso de llaves de longitud fija para el cifrado y descifrado del mensaje. Se podría inferir que la seguridad de HMAC también se basa en las fortalezas y debilidades de las llaves empleadas en la ejecución del algoritmo.

La longitud de las llaves HMAC pueden ser de cualquier longitud. Sin embargo, las llaves no deben tener una longitud inferior al tamaño del valor generado por la función HMAC (L=16 para MD5, L=20 para SHA-1) ya que esto conlleva a un decremento en la función de seguridad del algoritmo. Las llaves de longitud mayor a "L" no incrementan el nivel de seguridad del algoritmo. Se recomiendan llaves de mayor longitud extensa cuando la aleatoriedad del algoritmo se considera débil. [15]

Un aporte a la seguridad del algoritmo se presenta debido a que la función HMAC-SHA1 usada en el primer paso de la generación de valores HOTP pertenece a la familia de funciones pseudoaleatorias (funciones aparentemente aleatorias por la complejidad de su salida, pero que en realidad no lo son) las cuales brindan un mayor grado de complejidad a la salida. [5]

Aún cuando esto no puede ser probado. El uso de las pruebas de estadísticas de aleatoriedad sobre los valores generados por HOTP se puede considerar como una prueba práctica válida de seguridad sobre las contraseñas generadas.

[10] SHA-1: Estándar Hash Algorithm V 1.0 (FIPS 180-1)

4. PRUEBAS ESTADÍSTICAS DE ALEATORIEDAD

Hoy en día las aplicaciones hacen uso de la criptografía para incrementar su nivel de seguridad. Los protocolos criptográficos actuales hacen uso de valores de entrada generados por generadores aleatorios o semi-aleatorios de números.

El presente capítulo describe un conjunto de pruebas estadísticas que pueden ser usadas para la medición del nivel de aleatoriedad de diferentes tipos de generadores de números aleatorios. Las pruebas buscan desviaciones o anomalías con respecto a condiciones verdaderamente aleatorias en las secuencias binarias evaluadas.

4.1. ALEATORIEDAD

Una secuencia de bits aleatorios se puede interpretar como el resultado de múltiples lanzamientos imparciales de una moneda en donde una cara se etiqueta con el valor de "0" y la otra con el valor de "1". Con cada lanzamiento la probabilidad de que se produzca un "0" o un "1" es de ½. Adicionalmente, cabe resaltar que no hay dependencia entre los lanzamientos ya que el un lanzamiento previo no afecta el resultado del siguiente lanzamiento.

La secuencia de "0"s y "1"s generada a partir del lanzamiento imparcial de una moneda se considera una secuencia verdaderamente aleatoria. Los valores de "0" y "1" se encuentran aleatoria y uniformemente distribuidos en la secuencia. Todos los elementos de la secuencia se generan independientemente los unos de los otros; el próximo valor de la secuencia no puede ser predecido sin importar cuantos lanzamientos se hayan realizado anteriormente.

Aunque para propósitos criptográficos la idea de utilizar el lanzamiento imparcial de monedas es algo no práctico, el resultado hipotético generado a partir de los lanzamientos de moneda produce una secuencia verdaderamente aleatoria, la cual puede ser utilizada como marco de referencia para la evaluación de aleatoriedad en generadores de valores aleatorios y semi-aleatorios. [16]

4.2. GENERADORES DE NÚMEROS ALEATORIOS

Existen dos tipos de generadores para la producción de secuencias aleatorias de números aleatorios[11] y números semi-aleatorios[12].

Los generadores de números aleatorios (RNGs) hacen uso de una fuente no determinística y una función de procesamiento para la generación de aleatoriedad. La función de procesamiento, también conocida como proceso de destilación se usa para superar cualquier debilidad que tengan los datos de entrada (fuente). Por lo general, la fuente no determinística ha sido generada por una cantidad física como pueden ser por ejemplo: los resultados obtenidos de una fuente nuclear, o la cantidad de ruido en un circuito eléctrico, entre otros; O procesos que cambian en el tiempo como por ejemplo: los movimientos del ratón del computador en un instante.

Los valores generados por los RNGs pueden ser usados directamente como valores aleatorios o como entrada a generadores de valores semi-aleatorios. Se debe garantizar que los valores generados por los RNGs sean verdaderamente aleatorios. Para ello existen un conjunto de pruebas que pueden ser utilizadas para evaluar los resultados producidos por

[11] **Generadores de Números Aleatorios:** Random Number Generators (RNGs)
[12] **Generadores de Números Semi-aleatorios:** Pseudorandom Number Generators (PRNGs)

los generadores y determinar si estos valores que aparentan aleatorios en realidad lo son, y si son o no predecibles.

Los generadores de números semi-aleatorios (PRNGs) hacen uso de múltiples entradas y generan múltiples números semi-aleatorios. Las entradas para estos generadores se conocen como semillas. Las semillas deben ser valores aleatorios no predecibles. Estas semillas por lo general son el resultado de valores producidos por generadores de números aleatorios (RNGs).

La salida generada por PRNGs son típicamente funciones determinísticas de la semilla. La verdadera aleatoriedad se encuentra en la semilla. La naturaleza determinística del proceso es el que conlleva a la semi-aleatoriedad ya que la secuencia es reproducible a partir de la semilla. Esto es deseable en sistemas en donde se desea confirmar la secuencia a partir de la reproducción de la misma (a partir de la semilla).

4.3. VERIFICACIÓN DE ALEATORIEDAD

Existen diferentes tipos de pruebas estadísticas que pueden ser aplicadas a una secuencia para compararla y evaluarla con respecto a una secuencia verdaderamente aleatoria.

La aleatoriedad es una propiedad estadística y por lo tanto, las propiedades de una secuencia aleatoria se pueden caracterizar y describir en términos de probabilidad.

El resultado obtenido de aplicar pruebas estadísticas a una secuencia verdaderamente aleatoria es algo ya conocido y puede ser descrito en términos probabilísticos. Existen un número infinito de pruebas estadísticas aplicables a secuencias en búsqueda de "patrones" que si son detectados indican la presencia de no aleatoriedad en la secuencia evaluada.

El aplicar un conjunto finito de pruebas sobre una secuencia no es una prueba "completa" de aleatoriedad. Se deben saber interpretar los resultados obtenidos de la prueba para evitar conclusiones erróneas acerca de un generador específico.

Una prueba estadística se formula para validar una *hipótesis nula* (H_0) específica. La hipótesis nula por lo general soporta la probabilidad de que la secuencia evaluada es aleatoria. Asociada a la hipótesis nula, se presenta la *hipótesis alternativa* (H_a) la cual soporta la probabilidad de que la secuencia evaluada "no" es aleatoria.

Para cada prueba aplicada, la conclusión se deriva de un valor estadístico seleccionado que se usa para determinar la aceptación o el rechazo de la hipótesis nula. Bajo la suposición de aleatoriedad, el valor estadístico posee una distribución de posibles valores.

Una distribución de probabilidad es una lista de todos los resultados posibles de un experimento y de la probabilidad asociada a cada resultado.

De la distribución, se determina un *valor crítico* (un valor grande que se acerque a la cola de la distribución) el cual es utilizado para determinar la aceptación o el rechazo de la hipótesis nula cuando es comparado con el valor estadístico. Si el valor estadístico excede el valor crítico, se rechaza la hipótesis nula. De otro modo, la hipótesis nula es aceptada.

Las pruebas estadísticas basadas en hipótesis son un tipo de procedimiento "conclusivo" con dos posibles resultados. La aceptación de H_0 (aleatoriedad) o la aceptación de H_a (no aleatorio).

Existen otros dos tipos de conclusiones cuya probabilidad de ocurrencia es casi nula. A estos dos tipos de conclusiones se les conoce como "Error tipo 1" y "Error tipo 2".

El error tipo "1" se presenta cuando una secuencia verdaderamente aleatoria aparenta ser no aleatoria. A la probabilidad de ocurrencia de este tipo de error se le conoce como *nivel de significancia* y se denota por α. Un valor común para α es de 0.01 ya que la probabilidad de que una secuencia verdaderamente aleatoria presente características de no aleatoriedad es muy remota.

Un error tipo "2" se presenta cuando una secuencia no aleatoria es determinada como aleatoria. La probabilidad de este tipo de error se denota por β. A diferencia de α, β puede asumir diferentes valores ya que existen infinitas formas en las que una secuencia sea vista como no aleatoria. Cada forma puede tener un valor diferente β.

Realidad	**Conclusión**	
	Aceptar H_0	**Aceptar H_a (Rechaza H_0)**
Sec. es aleatoria	No error	Error tipo 1
Sec. no es aleatoria	Error tipo 2	No error

Tabla 1. Errores en pruebas estadísticas basadas en hipótesis

Cada prueba se basa en un valor estadístico el cual es una función de los datos. Para un valor estadístico "S" y un valor crítico "t", la probabilidad de un error tipo 1 es de: $P(S > t \,||\, H_0$ es verdadero) y la probabilidad de un error tipo 2 es de: $P(S \leq t \,||\, H_0$ es falso).

Probabilidad de Error tipo 1: $P(S > t \,||\, H_0$ es verdadero)
Probabilidad de Error tipo 2: $P(S \leq t \,||\, H_0$ es falso)

El valor estadístico es usado para calcular un valor P el cual resume la fortaleza de la evidencia contra la hipótesis nula. Para las pruebas estadísticas, cada valor de P corresponde a la probabilidad de que un generador perfecto de números aleatorios produzca una secuencia menos aleatoria que la secuencia siendo evaluada. Si "P" es igual a 1

entonces la secuencia es completamente aleatoria. Si "P" es igual a 0 entonces la secuencia no es aleatoria.

Se debe seleccionar el nivel de significancia α para las pruebas. Si $P \geq \alpha$, se acepta la hipótesis nula. Si $P < \alpha$, se rechaza la hipótesis nula.

Si $P \geq \alpha$, se acepta H_0
Si $P < \alpha$, se rechaza H_0

Un α de 0.001 indica que se espera el rechazo de 1 secuencia en 1000 si la secuencia era aleatoria. Para $P > 0.001$, la secuencia se considera aleatoria con un nivel de confianza del 99.9%. Para un valor de $P < 0.001$, la secuencia se considera no aleatoria con un nivel de confianza del 99.9%.

Un α de 0.01 indica que se espera el rechazo de 1 secuencia en 100 si la secuencia era aleatoria. Para $P > 0.01$, la secuencia se considera aleatoria con un nivel de confianza del 99%. Para un valor de $P < 0.01$, la secuencia se considera no aleatoria con un nivel de confianza del 99%.

Con respecto a las secuencias binarias a ser evaluadas mediante las pruebas estadísticas, se hacen las siguientes suposiciones:

- **Uniformidad:** En cualquier punto durante la generación de valores aleatorios o semi-aleatorios el número de "0"s y "1"s debe ser más o menos el mismo. Se espera que el número de ceros en la secuencia evaluada sea de n/2 donde "n" es igual a la longitud de la secuencia.

- **Escalabilidad:** Cualquier prueba aplicada a la secuencia puede ser aplicada a subsecuencias extraídas aleatoriamente.

- **Consistencia:** El comportamiento del generador debe ser consistente a través de los valores de entrada. Se considera inadecuado realizar pruebas sobre un PRNG basándose en la salida de una sola semilla (fuente) o para un RNG basándose en la salida de solo una cantidad física.

Las pruebas a estudiar, implementar y posteriormente a utilizar en el análisis estadístico son:

1. Prueba de Frecuencia (Frequency Monobit Test)
2. Prueba de Frecuencia en Bloques (Frequency within a Block)
3. Prueba de Cadenas (Runs Test)
4. Prueba de Cadenas en Bloques (Longest Run of Ones in a Block Test)
5. Prueba de Complejidad Lienal (Linear Complexity Test)
6. Prueba Serial (Serials Test)
7. Prueba de Emparejamiento Usando Plantillas que se Sobreponen (Non-overlapping Template Matching Test)
8. Prueba de Emparejamiento Usando Plantillas que se no se Sobreponen (Overlapping Template Matching Test)
9. Prueba de Acercamiento a la Entropía (Approximate Entropy Test)
10. Prueba de Sumas Acumulativas (Cumulative Sums (Cusums) Test)

En el presente capítulo se puede visualizar el objetivo y funcionamiento de cada una de las pruebas a utilizar para la evaluación de aleatoriedad de los algoritmos.

4.3.1. FREQUENCY (MONOBIT) TEST

El propósito de esta prueba es el de validar que la proporción de ceros y unos en la secuencia evaluada sea más o menos la misma. La prueba busca que la fracción de unos en la secuencia sea cercana a ½.

Es de mencionar que esta es la primera prueba que se debe efectuar sobre una secuencia. La ejecución de las demás pruebas dependen del resultado de esta prueba.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

Valor estadístico y distribución:

Sobs: Valor absoluto de la suma de los Xi (donde Xi = 2ε - 1 = $\pm$ 1) en la secuencia dividida por la raíz cuadrada de la longitud de la secuencia.

La distribución de probabilidad para la prueba estadística es una media normal ($Z = \frac{Sobs}{\sqrt{2}}$ se distribuye normal. Por lo tanto |z| se distribuye media normal).

Si la secuencia es aleatoria, los "-1" y los "1" tenderán a cancelarse resultando en un valor estadístico de casi 0. Si hay demasiados "1"s o "0"s el valor del estadístico será superior a 0.

Descripción de la Prueba

1. Conversión a ± 1: Los 0s y 1s de la secuencia de entrada (ε) son transformados a -1 y +1 respectivamente para producir el valor $S_n = X_1 + X_2 + \ldots + X_n$ donde $X_i = 2\varepsilon_i - 1$.

2. Calcular el valor estadístico Sobs: $Sobs = \frac{|S_n|}{\sqrt{n}}$

3. Calcular el valor P: $P = erfc\left(\frac{Sobs}{\sqrt{2}}\right)$ donde erfc() es la función de error complementario.

Conclusión e Interpretación de Resultados

Si el valor calculado de $P < 0.01$ se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Si P fuese un valor pequeño < 0.01, los valores de $|S_n|$ y $|S_{obs}|$ serían grandes. Valores grandes positivos de S_n indican una gran proporción de "1"s mientras que valores grandes negativos de S_n indican una gran proporción de "0"s.

4.3.2. FREQUENCY WITHIN A BLOCK TEST

El foco de esta prueba es la proporción de ceros y unos en bloques de M-bits. La prueba busca determinar si la frecuencia de 1s en bloques de M-bits es aproximadamente M/2.

Cuando M = 1 la prueba se degenera y se debe aplicar la prueba 1. Frecuencia (Monobit).

Notación:

M: Tamaño de cada bloque.

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

Valor estadístico y distribución:

X^2obs: Medida de que tanto la proporción de 1s observada en el bloque de M bits se acerca a la proporción esperada (½).

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Particionar la secuencia en $N = \frac{n}{M}$ bloques. Se deben descartar los bits no utilizados.

2. Determinar la proporción de 1s en cada bloque de M-bits (π_i):

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)}(M + j)}{M}, 1 \le i \le N$$

3. Calcular el valor estadístico X^2: $X^2\ (obs) = 4M \sum_{i=1}^{N} (\pi_i - 1/2)^2$

4. Calcular el valor P = igamc(N/2, X^2(obs)/2). A igamc se le conoce como la función incompleta de Gamma para Q(a,x).

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Los valores pequeños de P (< 0.01), indicarían una gran desviación en la proporción de ceros y unos en por lo menos uno de los bloques.

4.3.3. RUNS TEST

El foco de esta prueba es el número de cadenas (runs) en cada secuencia.

Una cadena o run es una secuencia continua de bits idénticos. Una cadena de longitud k es una cadena compuesta por k bits idénticos. Una cadena se identifica ya que sus extremos (inicial y final) se encuentran delimitados por un bit de valor opuesto.

El propósito de la prueba es el de validar que el número de cadenas de ceros o unos de diferentes longitudes revelen patrones de cadenas esperados (similares a los de una secuencia verdaderamente aleatoria). Este tipo de prueba es utilizada comúnmente para determinar si el grado de oscilación entre ceros y unos es muy rápido o muy lento.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

Valor estadístico y distribución:

V_n(obs): Número total de cadenas (Número total de cadenas de ceros + Número total de cadenas de unos) en la secuencia de n bits.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Determinar la proporción de 1s en la secuencia de entrada (π): $\pi = \frac{\sum j\varepsilon_j}{n}$

2. Si la proporción de 1s no supera la prueba de Frecuencia (Frequency Test), la prueba no se lleva a cabo y se asigna un valor de 0 a P. Para este cálculo se hace uso de la siguiente función:

   Limite: $|\pi - 1/2| \geq t$ donde $t = \frac{2}{\sqrt{n}}$

3. Calcular el valor estadístico V_n(obs): $Vn(obs) = \sum_{k=1}^{n-1} r(k) + 1$ donde:

   r(k) = 0, si $\varepsilon_k = \varepsilon_{k+1}$

   r(k) = 1, si $\varepsilon_k \neq \varepsilon_{k+1}$

4. Calcular el valor P: $P = erfc\left(\frac{Vn(obs) - 2n\pi(1-\pi)}{2\sqrt{2n\pi(1-\pi)}}\right)$

Conclusión e Interpretación de Resultados

Si el valor calculado de $P < 0.01$ se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Un valor grande de Vn(obs) indica un cambio muy rápido entre ceros y unos en la secuencia (oscilación rápida). Un valor pequeño de Vn(Obs) indica un cambio muy lento entre ceros y unos (oscilación lenta).

4.3.4. LONGEST RUN OF ONES IN A BLOCK TEST

El foco de esta prueba es la cadena de unos más larga en bloques de "M" bits. La prueba busca determinar si la longitud de la cadena más larga de unos dentro de la secuencia analizada posee una longitud a similar a la encontrada en una secuencia verdaderamente aleatoria.

Notación:

M: Tamaño de cada bloque (M debe ser =8, = 128 o = 10^4).

M	K	N
8	3	16
128	5	49
10^4	6	75

Tabla 2. Valores de K y N para M =8, =128 o = 10^4

N: Número de bloques.

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

Valor estadístico y distribución:

X^2(obs): Medida de que tanto la cadena de unos más larga en bloques de M bits se acerca al valor esperado de cadenas largas en bloques de M-bits.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Particionar la secuencia en $N = \frac{n}{M}$ bloques.
2. Calcular el valor estadístico X^2(obs): $X^2(obs)_i = \sum_{i=o}^{k} \frac{(V_i - N\pi_i)^2}{N\pi_i}$
3. Calcular el valor P = igamc(k/2, X^2(obs)/2).

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Los valores grandes de P ($\geq$ 0.01), indicarían una gran ocurrencia de grupos de 1s.

4.3.5. NON-OVERLAPPING TEMPLATE MATCHING TEST

El foco de esta prueba es el número de ocurrencias de cadenas predefinidas. La prueba busca detectar generadores que producen múltiples ocurrencias de un patrón no periódico.

Este tipo de prueba hace uso de una ventana de m-bits usada para buscar un patrón específico de m-bits. Si la secuencia no se encuentra la ventana se desliza en 1 bits generando una nueva ventana de búsqueda. Si se encuentra el patrón, se resetea la ventana al bit siguiente de la cadena encontrada y se reanuda la búsqueda.

Notación:

m: Longitud en bits de cada plantilla.
n: Longitud de la cadena de bits.
ε: Secuencia de bits evaluada generada por RNGs o PRNGs.
$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$
B: Plantilla de m-bits a ser buscada.
M: Longitud en bits de cada subcadena de ε a ser analizada.
N: Número de bloques independientes.

Valor estadístico y distribución:

X^2(obs): Medida de que tanto el número observado de ocurrencias de un determinado patrón se asemeja al valor esperado de patrones encontrados.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Dividir la secuencia en N bloques independientes de longitud M.

2. Sea W_j (j=1, …, N) el número de veces que B (el patrón) se encuentra en el bloque j.

3. Bajo una hipótesis de aleatoriedad se debe calcular la media teórica (μ) y la varianza (σ^2).

$$\mu = (M - m + 1) / 2^m \qquad \sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right)$$

4. Calcular X^2(obs): $X^2(obs) = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$

5. Calcular el valor P: $P = igamc(\frac{N}{2}, \frac{X^2(obs)}{2})$

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Un valor de pequeño de P indicaría la existencia de patrones irregulares de la secuencia buscada.

4.3.6. OVERLAPPING TEMPLATE MATCHING TEST

El foco de esta prueba es el número de ocurrencias de cadenas predefinidas. La prueba busca detectar generadores que producen múltiples ocurrencias de un patrón no periódico.

Este tipo de prueba hace uso de una ventana de m-bits usada para buscar un patrón específico de m-bits. Si la secuencia no se encuentra la ventana se desliza en 1 bits generando una nueva ventana de búsqueda. A diferencia de la prueba anterior, si se encuentra el patrón, la ventana se desliza al siguiente bit antes de reanudar la búsqueda.

Notación:

- m: Longitud en bits de cada plantilla.
- n: Longitud de la cadena de bits.
- ε: Secuencia de bits evaluada generada por RNGs o PRNGs.
  $\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$
- B: Plantilla de m-bits a ser buscada.
- M: Longitud en bits de cada subcadena de ε a ser analizada.
- N: Número de bloques independientes.

Valor estadístico y distribución:

- X^2(obs): Medida de que tanto el número observado de ocurrencias de un determinado patrón se asemeja al valor esperado de patrones encontrados.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Dividir la secuencia en N bloques independientes de longitud M.

2. Calcular el número de ocurrencias de B en cada uno de los N bloques.

3. Calcular los valores λ y η los cuales serán utilizados para calcular las probabilidades estadísticas π.

$$\lambda = (M - m + 1) / 2^m \qquad \eta = \lambda / 2$$

4. Calcular X^2(obs): $X^2(obs) = \sum_{j=1}^{N} \frac{(V_i - N\prod_i)^2}{\sigma^2}$

5. Calcular el valor P: $P = igamc(\frac{N}{2}, \frac{X^2(obs)}{2})$

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Un valor de pequeño de P indicaría la existencia de patrones irregulares de la secuencia buscada.

4.3.7. LINEAR COMPLEXITY TEST

El foco de esta prueba es la longitud del registro lineal de retroalimentación.

La prueba busca determinar si una secuencia es lo suficientemente compleja para ser considerada como aleatoria. Secuencias verdaderamente aleatorias se caracterizan por hacer uso de entradas o registros lineales de retroalimentación (LFSR) extensos.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

M: Longitud en bits de cada bloque.

K: Grados de libertad (K=6).

Valor estadístico y distribución:

X^2(obs): Medida de que tanto el número observado de ocurrencias de una longitud fija de LFSR se asemeja al valor esperado generado por una secuencia considerada como aleatoria.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Dividir la secuencia en N bloques independientes. n=M*N

2. Haciendo uso del algoritmo Berlekamp-Massey determinar la complejidad lineal de L_i para cada uno de los N bloques. L_i es la longitud del registro de retroalimentación más corto que genera todos los bits en el bloque i. Dentro de una secuencia L_i, la combinación de algunos bits, cuando son sumados mod 2, producen el siguiente bit en la secuencia (bit $L_i + 1$).

3. Bajo un supuesto de aleatoriedad, calcular μ:

$$\mu = \frac{M}{2} + \frac{(9 + (-1)^{M+1})}{36} + \frac{(M/3 + 2/9)}{2^M}$$

4. Para cada subcadena calcular T_i: $T_i = (-1)^M \bullet (L_i - \mu) + 2/9$

5. Calcular X^2(obs): $X^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\prod_i)^2}{N\prod_i}$

6. Calcular *P:* $P = igamc\left(\frac{K}{2}, \frac{X^2(obs)}{2}\right)$

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

4.3.8. SERIAL TEST

El foco de esta prueba es la frecuencia de todos los patrones de "m" bits que se sobreponen en la secuencia. La prueba busca determinar si la secuencia presente características de uniformidad validando que el número de ocurrencias de los 2^m patrones que se sobreponen es aproximadamente el número esperado en comparación a lo producido por una secuencia aleatoria.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

m: Longitud en bits de cada bloque.

Valor estadístico y distribución:

$\nabla\psi^2_m(obs)$ y $\nabla^2\psi^2_m(obs)$: Medida de que tanto el número observado de frecuencias de patrones de m-bits se asemeja al número esperado de patrones de m-bits.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Generar una secuencia aumentada ε' extendiendo la secuencia al anexar los primeros m-1 bits al final de la secuencia.

2. Determinar la frecuencia de todos los posibles bloques de m-bits, de m-1 bits y de m-2 bits.

   Sea $v_{i,\ldots im}$ la frecuencia de los patrones de m bits $i_1,\ldots i_m$.
   Sea $v_{i,\ldots im-1}$ la frecuencia de los patrones de m-1 bits $i_1,\ldots i_{m-1}$.
   Sea $v_{i,\ldots im-2}$ la frecuencia de los patrones de m-2 bits $i_1,\ldots i_{m-2}$.

3. Calcular ψ^2_m: $\psi^2_m = \frac{2^M}{n} \sum_{i_1 \ldots i_m} \left(V_{i_1,\ldots,i_m} - n/2^M \right)^2$

4. Calcular ψ^2_{m-1}: $\psi^2_{m-1} = \frac{2^{M-1}}{n} \sum_{i_1 \ldots i_{m-1}} \left(V_{i_1,\ldots,i_{m-1}} - n/2^{M-1} \right)^2$

5. Calcular ψ^2_{m-2}: $\psi^2_{m-2} = \frac{2^{M-2}}{n} \sum_{i_1 \ldots i_{m-2}} \left(V_{i_1,\ldots,i_{m-2}} - n/2^{M-2} \right)^2$

6. Calcular: $\nabla \psi^2_m = \psi^2_m - \psi^2_{m-1}$ y $\nabla^2 \psi^2_m = \psi^2_m - 2\psi^2_{m-1} + \psi^2_{m-2}$

7. Calcular *P*: $P_1 = igamc\left(2^{m-2}, \nabla \psi^2_m\right)$ y $P_1 = igamc\left(2^{m-3}, \nabla \psi^2_m\right)$

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

4.3.9. APPROXIMATE ENTROPY TEST

El foco de esta prueba es la frecuencia de todos los patrones de "m" bits que se sobreponen en la secuencia. La prueba busca comparar la frecuencia de dos bloques consecutivos que se sobreponen contra el valor esperado generado por una secuencia considerada como aleatoria.

Cuando se habla de entropía, se habla de incertidumbre. Entre mayor sea el grado de incertidumbre menos es la probabilidad de que un ataque sea exitóso.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

m: Longitud en bits de cada bloque.

Valor estadístico y distribución:

X^2(obs): Medida de que tanto el valor observado de entropía se asemeja al valor esperado generado por una secuencia considerada como aleatoria.

La distribución de referencia para la prueba estadística es de una chi-cuadrado (X^2).

Descripción de la Prueba

1. Generar una secuencia aumentada, extendiendo la secuencia al anexar los primeros m-1 bits al final de la secuencia

2. Realizar una cuenta de los n bloques que se sobreponen. La cuenta de todos los posibles valores de m bits (m+1 bit) se representan mediante $C^i_{m.}$

3. Calcular C^i_m: $C^i{}_m = \frac{\#i}{n}$ para cada valor de i.

4. Calcular φ^m: $\varphi^m = \sum_{i=0}^{2^m-1} \Pi_i \log \Pi_i$

5. Repetir los pasos (1) – (4), reemplazando m por m+1.

6. Calcular X^2(obs): $X^2 = 2n[\log 2 - ApEn(m)]$

7. Calcular *P*: $P = igamc\left(2^{m-1}, \frac{X^2}{2}\right)$

Conclusión e Interpretación de Resultados

Si el valor calculado de P < 0.01 se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

Los valores pequeños de ApEn(m) implican alta regularidad. Grandes valores implican alta irregularidad.

4.3.10. CUMULATIVE SUMS TEST

El propósito de esta prueba es el de determinar si la suma acumulativa de secuencias parciales existentes en la secuencia analizada es muy grande o muy pequeña con respecto a la esperada para una secuencia aleatoria. Para una secuencia verdaderamente aleatoria, la suma acumulativa debe ser cercana a 0.

Notación:

n: Longitud de la cadena de bits.

ε: Secuencia de bits evaluada generada por RNGs o PRNGs.

$\varepsilon = \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$

mode: Mode = 0. Se aplica la prueba en sentido hacia adelante.

Mode = 1. Se aplica la prueba en sentido opuesto (reversa).

Valor estadístico y distribución:

z: Valor que corresponde al recorrido más largo desde el origen en la suma acumulativa de la secuencia.

La distribución de probabilidad para la prueba estadística es una normal.

Descripción de la Prueba

1. Normalizar la secuencia mediante la conversión a ± 1: Los 0s y 1s de la secuencia de entrada (ε) son transformados a -1 y +1, respectivamente para producir $S_n = X_1 + X_2 + \ldots + X_n$ donde $X_i = 2\varepsilon_i - 1$.

2. Calcular la sumatoria parcial S_i de subsecuencias largas sucesivas. Para mode = 0 se inicia con X_1. Para mode = 1 se inicia con X_n.

Si mode = 0	Si mode = 1
$S_1 = X_1$	$S_1 = X_n$
$S_2 = X_1 + X_2$	$S_2 = X_n + X_{n-1}$
.	.
.	.
$S_k = X_1 + X_2 + \ldots + X_k$	$S_k = X_n + X_{n-1} + \ldots + X_{n-k+1}$
..	.
$Sn = X_1 + X_2 + \ldots + Xk + \ldots X_n$	.
	$Sn = X_n + X_{n-1} + \ldots + X_{n-k+1} + \ldots X_1$

3. Calcular el valor estadístico $z=\max_{1<k<n}|S_k|$

4. Calcular el valor P:

$$P = -1\sum_{k=\left(\frac{-n}{z}+1\right)/4}^{\left(\frac{n}{z}-1\right)/4}\left[\phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)-\phi\left(\frac{(4k-1)z}{\sqrt{n}}\right)\right]+\sum_{k=\left(\frac{-n}{z}-3\right)/4}^{\left(\frac{n}{z}-1\right)/4}\left[\phi\left(\frac{(4k+3)z}{\sqrt{n}}\right)-\phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)\right]$$

Conclusión e Interpretación de Resultados

Si el valor calculado de $P < 0.01$ se concluye que la secuencia no es aleatoria. De otro modo se concluye que la secuencia es aleatoria.

En mode = 0, cuando el estadístico presenta valores altos es que hay un número elevado de ceros o unos al inicio de la secuencia.
En mode = 1, cuando el estadístico presenta valores altos es que hay un número elevado de ceros o unos al final de la secuencia.

4.4. RESULTADOS DE LAS PRUEBAS

4.4.1. ACEPTACIÓN DE LA HIPÓTESIS NULA

Mediante el uso del conjunto de pruebas estadísticas ejecutadas se cuantifica la aleatoriedad del algoritmo HOTP en comparación a otros valores aleatorios como son: una cantidad física, raíz de dos, raíz de tres, e, Pi con un millón de dígitos, los valores generados por el generador de números aleatorios de Java y los valores generados por el autenticador de RSA Security.

Los resultados obtenidos fueron los siguientes:

Pruebas	e	Pi	Raiz de 2	Raiz de 3	Cantidad Fisica	Hotp (n=1000)	RSA (n=1000)	Java RNG
Frequency	0.92608329	0.6144219	0.8177513	0.5461573	0.28097797	0.38641444	0.87257749	0.74029984
Block Frequency (M=10)	0.23198071	0.3139302	0.2265836	0.5632272	0.73068207	0.76815739	0.91122580	0.85425363
Runs	0.56191688	0.4192684	0.3134272	0.2611232	0.20028332	0.77445941	0.52182378	0.99322337
Longest Runs of Ones	0.71894532	0.0243896	0.0121165	0.4467261	0.75522656	0.63252082	0.28437747	0.97156475
Serials	0.42099826	0.1059711	0.9599614	0.3819387	0.72987160	0.59105293	0.12989039	0.63621955
Linear Complexity	0	0	0	0	0.85134482	0.40178210	0.15940460	0
Non-Overlapping	0.07879013	0.1657574	0.5694611	0.5322352	0.99925226	0.96903365	0.04137294	0.99998339
Overlapping	0.11043116	0.2968995	0.7919616	0.0827162	0	0.29571010	0.16709561	0
Aproximate Entropy	0.42650563	0.6070678	0.8899906	0.4352703	0.46909161	0.96394592	0.09856496	0.17143473
Cumulative Sums	0	0	0	0	0.37700989	0.64955264	0.81303614	0.98108335

Tabla 3. Resultados obtenidos de aplicar las pruebas estadísticas sobre las secuencias seleccionadas.

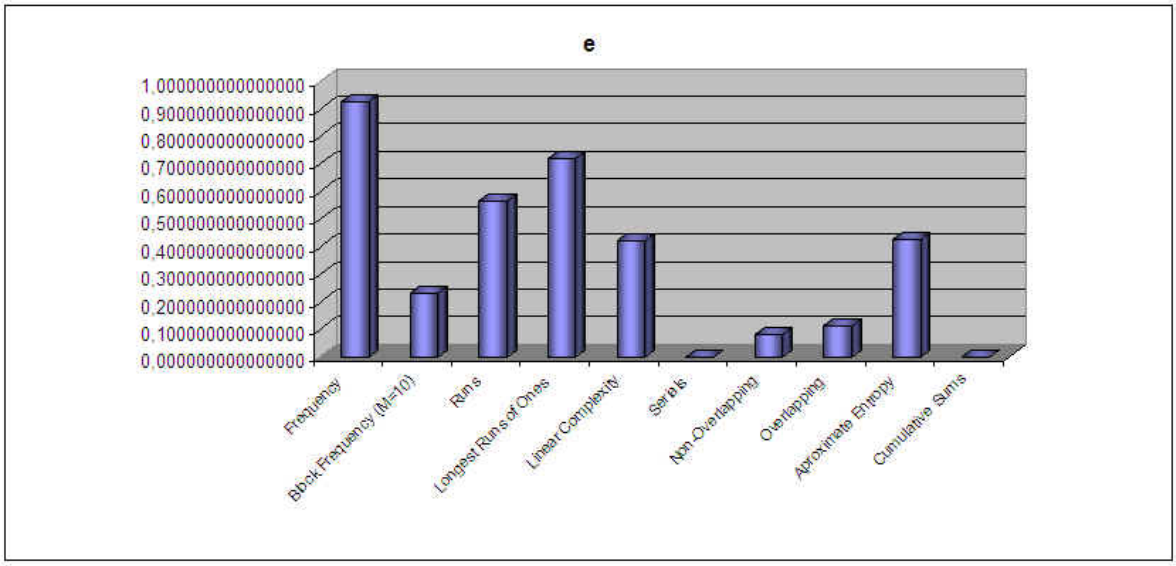


Figura 7. Resultados de las pruebas sobre "e"

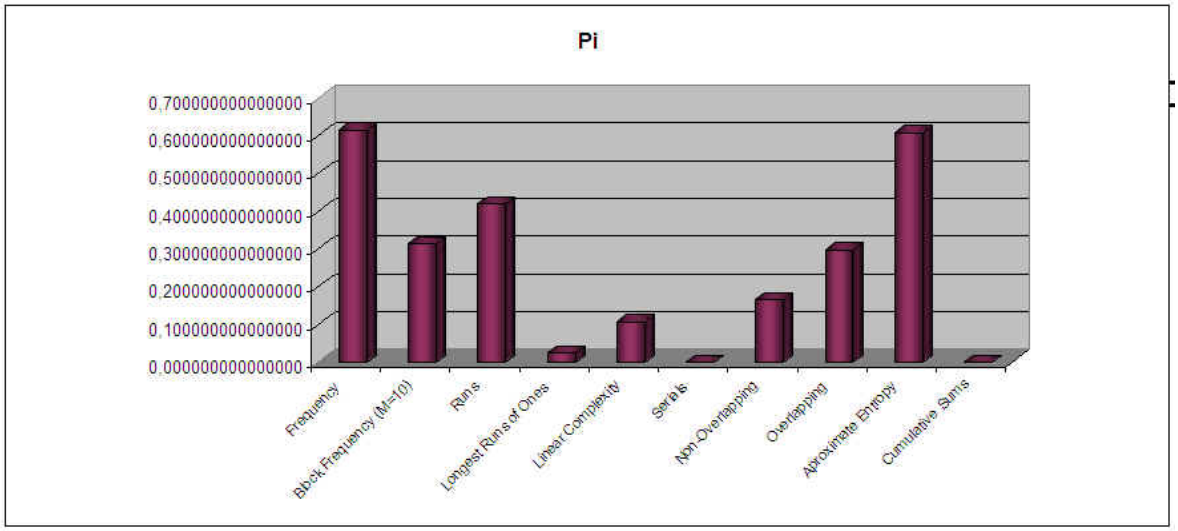


Figura 8. Resultados de las pruebas sobre "Pi"

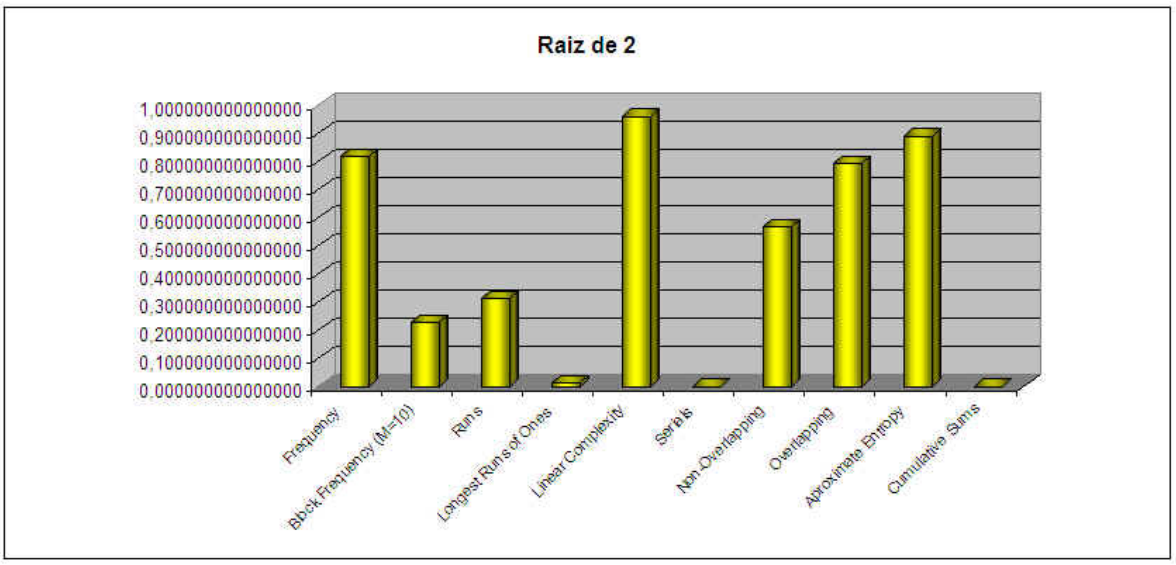


Figura 9. Resultados de las pruebas sobre "Raíz de 2"

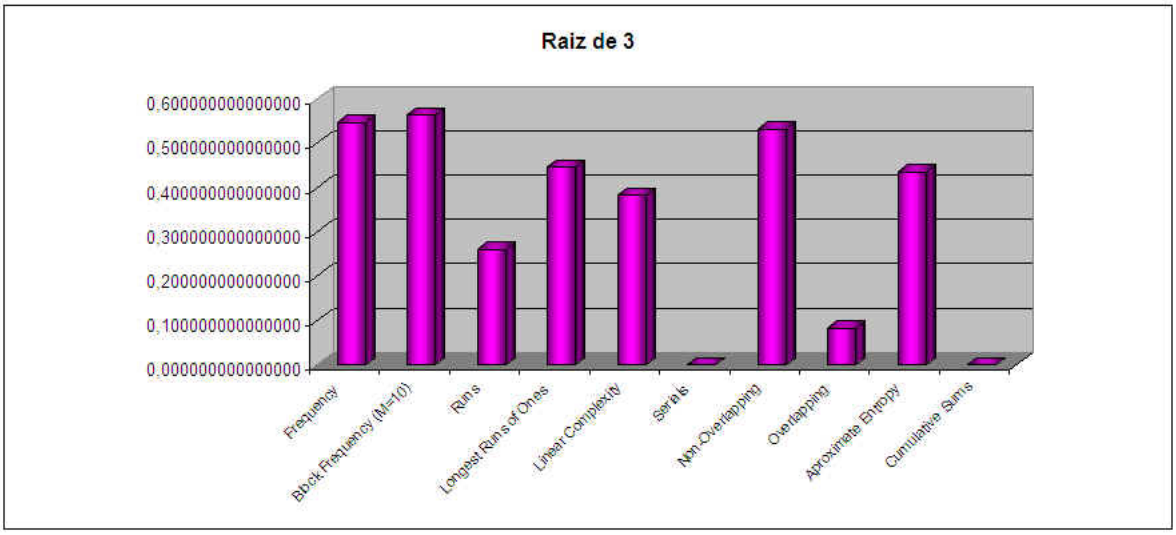


Figura 10. Resultados de las pruebas sobre "Raíz de 3"

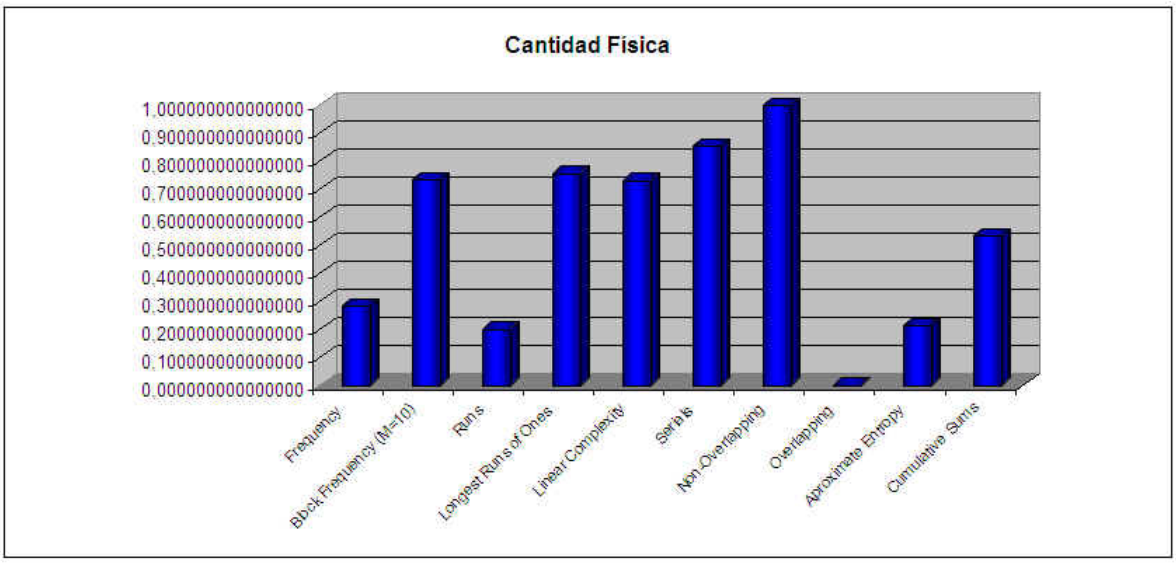


Figura 11. Resultados de las pruebas sobre "Cantidad Física"

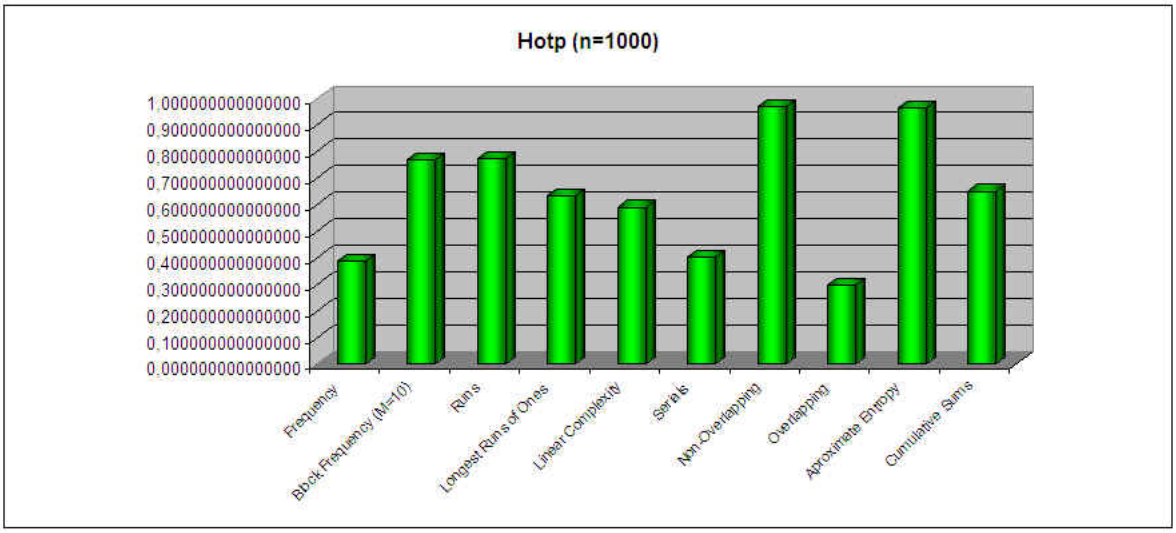


Figura 12. Resultados de las pruebas sobre "Hotp"

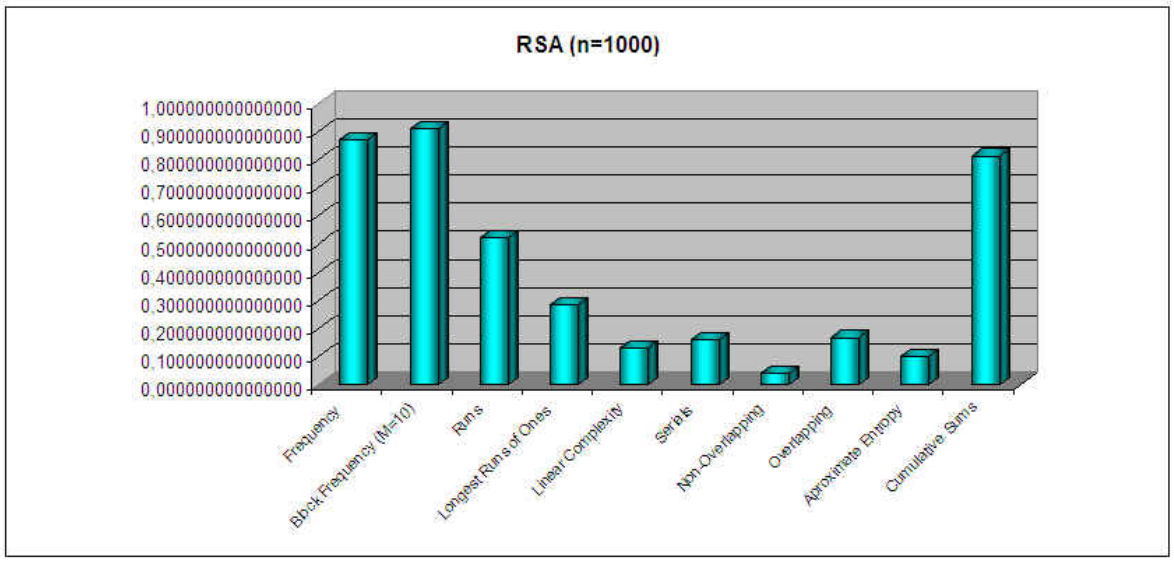


Figura 13. Resultados de las pruebas sobre "RSA"

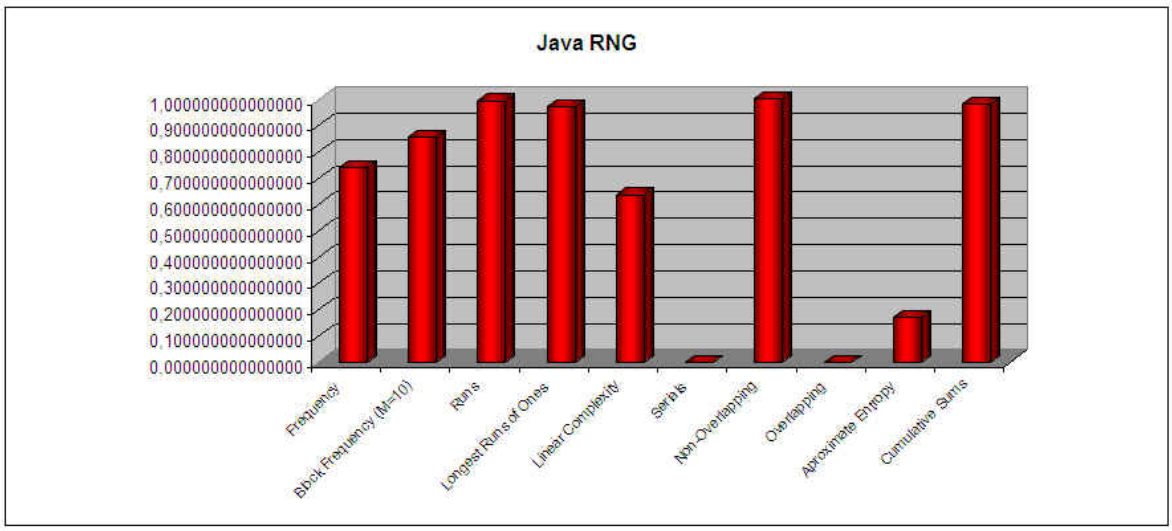


Figura 14. Resultados de las pruebas sobre "Java RNG"

Las pruebas estadísticas aplicadas a las secuencias se usaron para llevar a cabo una comparación con respecto a una secuencia verdaderamente aleatoria.

Para las pruebas se seleccionó un valor crítico de 0.01 para determinar la aceptación o el rechazo de la hipótesis nula (aleatoriedad de la secuencia) cuando es comparado con el valor estadístico.

Como se mencionó anteriormente, un valor crítico (α) de 0.01 indica que se espera a lo sumo un rechazo de 1 secuencia en 100 si la secuencia es aleatoria.

Como se puede observar en la tabla y en las gráficas de resultados obtenidos, todas las secuencias presentaron valores probabilísticos superiores al 0.01. Por lo tanto, se puede asegurar con un 99% de confiabilidad que las secuencias presentan características de aleatoriedad.

Las pruebas aplicadas buscaban desviaciones o anomalías con respecto a condiciones verdaderamente aleatorias en las secuencias binarias evaluadas. Las secuencias evaluadas presentaron un mínimo de desviaciones; Para todas las pruebas aplicadas se aceptó la hipótesis nula de aceptación de aleatoriedad.

4.4.2. COMPARACIÓN DE LOS RESULTADOS OBTENIDOS

Un segundo objetivo de la medición de aleatoriedad de las secuencias consistía en comparar la aleatoriedad entre las diferentes secuencias, y en especial entre la secuencia generada por Hotp con respecto a la generada por la solución comercial RSA.

Para comparar el grado de aleatoriedad entre las diferentes secuencias se ordenaron y enumeraron las secuencias en orden descendente por el valor probabilístico de aleatoriedad obtenido para cada una de las pruebas.

Los resultados obtenidos del ordenamiento fueron los siguientes:

#	Algoritmos	Frequency
1	e	0.9260832906758789
2	RSA	0.8725774993410996
3	Raiz de 2	0.8177513809816284
4	Java	0.7402998418448125
5	Pi	0.6144219294989157
6	Raiz de 3	0.5461573080730796
7	Hotp	0.3864144489873138
8	Cantidad Física	0.2809779762663137

#	Algoritmos	Block Frequency (M=10)
1	RSA	0.911225808524154
2	Java	0.8542536354925319
3	Hotp	0.7681573990810643
4	Cantidad Física	0.7306820799708964
5	Raiz de 3	0.5632272876232571
6	Pi	0.3139302308920807
7	e	0.2319807191643673
8	Raiz de 2	0.2265836144586060 3

#	Algoritmos	Runs
1	Java	0.9932233785337405
2	Hotp	0.7744594148079573
3	e	0.5619168850302545
4	RSA	0.5218237864888895
5	Pi	0.41926842044315493
6	Raiz de 2	0.3134272425127038
7	Raiz de 3	0.2611232602838417
8	Cantidad Física	0.2002833239987769

#	Algoritmos	Longest Runs of Ones
1	Java	0.971564754423215
2	Cantidad Física	0.7552265660799791
3	e	0.7189453298987654
4	Hotp	0.6325208253456209
5	Raiz de 3	0.4467261349887360 3
6	RSA	0.2843774780133880 3
7	Pi	0.0243896985338965 85
8	Raiz de 2	0.012116599153050 77

#	Algoritmos	Serials
1	Raiz de 2	0.9599614077390768
2	Cantidad Física	0.7298716090519525
3	Java	0.6362195515889799
4	Hotp	0.5910529330316987
5	e	0.4209982603054207 6
6	Raiz de 3	0.3819387521004175
7	RSA	0.1298903983867254 8
8	Pi	0.1059711989321480 6

#	Algoritmos	Linear Complexity
1	Cantidad Física	0.8513448253456387
2	Hotp	0.4017821076702344
3	RSA	0.1594046027727946
4	e	0
4	Java	0
4	Pi	0
4	Raiz de 2	0
4	Raiz de 3	0

#	Algoritmos	Non-Overlapping
1	Java	0.9999833932213484
2	Cantidad Física	0.9992522603315008
3	Hotp	0.9690336578556106
4	Raiz de 2	0.569461171562019
5	Raiz de 3	0.5322352840045714
6	Pi	0.1657574574552243
7	e	0.0787901326766633 8
8	RSA	0.0413729415688849 1

#	Algoritmos	Overlapping
1	Raiz de 2	0.7919616459734532
2	Pi	0.2968995915562530 6
3	Hotp	0.2957101035079951
4	RSA	0.1670956101298890 8
5	e	0.1104311643205683 5
6	Raiz de 3	0.0827162075316406 5
7	Cantidad Física	0
7	Java	0

#	Algoritmos	Approximate Entropy
1	Hotp	0.963945927735498
2	Raiz de 2	0.8899906452893527
3	Pi	0.6070678542105872
4	Cantidad Física	0.4690916162166196 6
5	Raiz de 3	0.4352703953123929
6	e	0.4265056379896054 5
7	Java	0.1714347348324791 7
8	RSA	0.0985649695102954 5

#	Algoritmos	Cumulative Sums
1	Java	0.9810833514746271
2	RSA	0.8130361417590151
3	Hotp	0.6495526489454839
4	Cantidad Física	0.3770098932769885 3
5	e	0
5	Pi	0
5	Raiz de 2	0
5	Raiz de 3	0

Tabla 4. Ponderación de las secuencias por cada prueba aplicada

Estos resultados se pueden visualizar mejor mediante el uso de los siguientes gráficos:

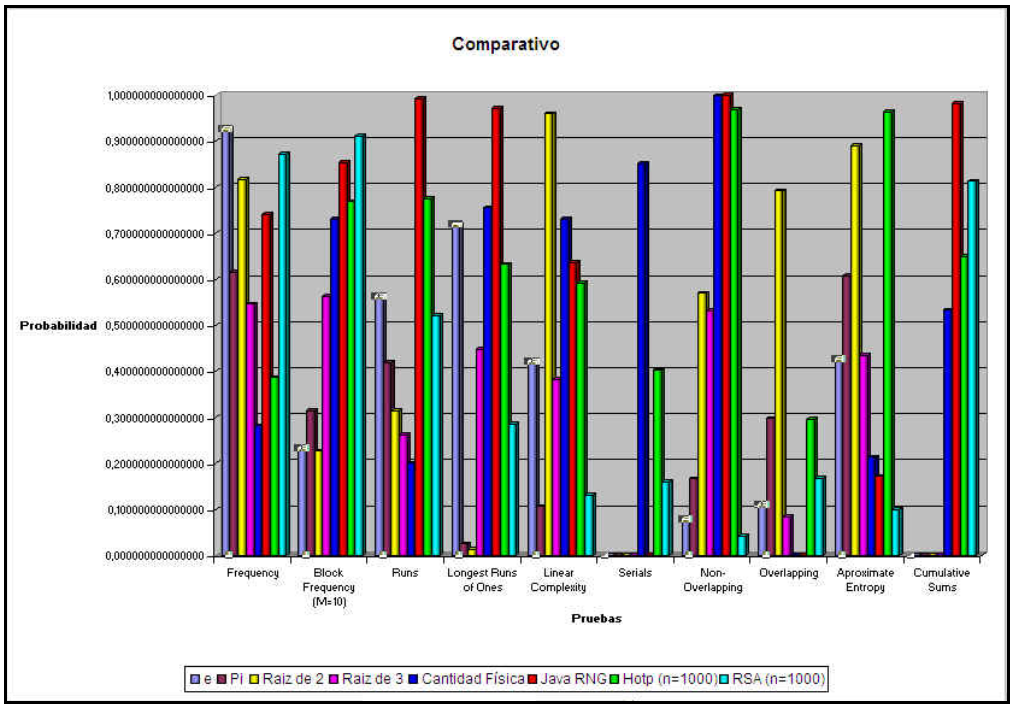


Figura 15. Diagrama de Gantt – Comparativo Probabilidad obtenida de aplicar las pruebas a cada tipo de secuencia.

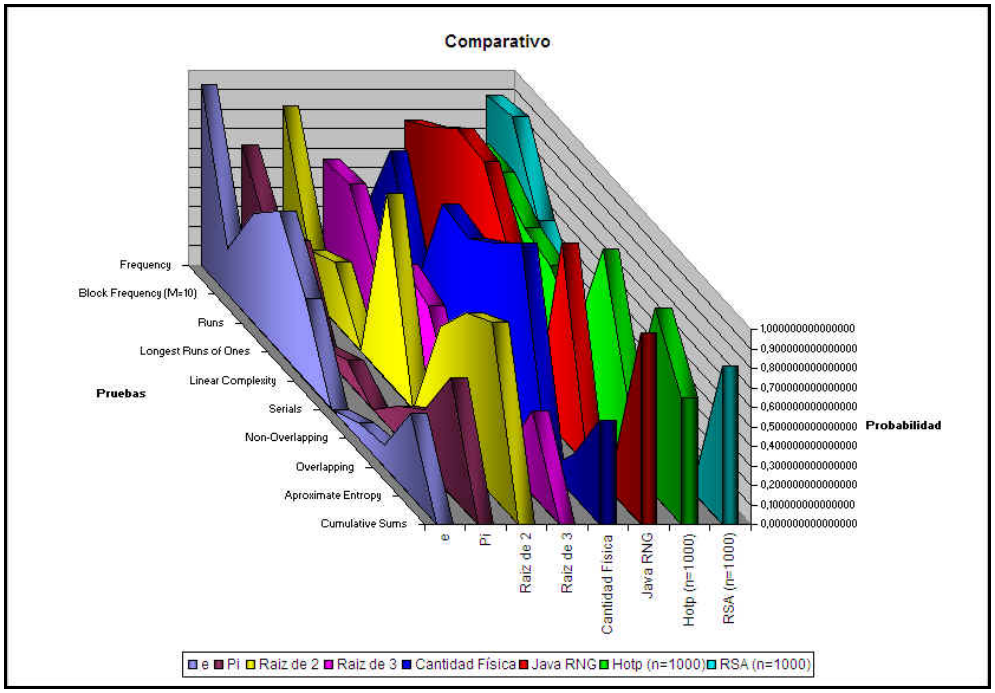


Figura 16. Diagrama de Áreas – Comparativo Probabilidad obtenida de aplicar las pruebas a cada tipo de secuencia.

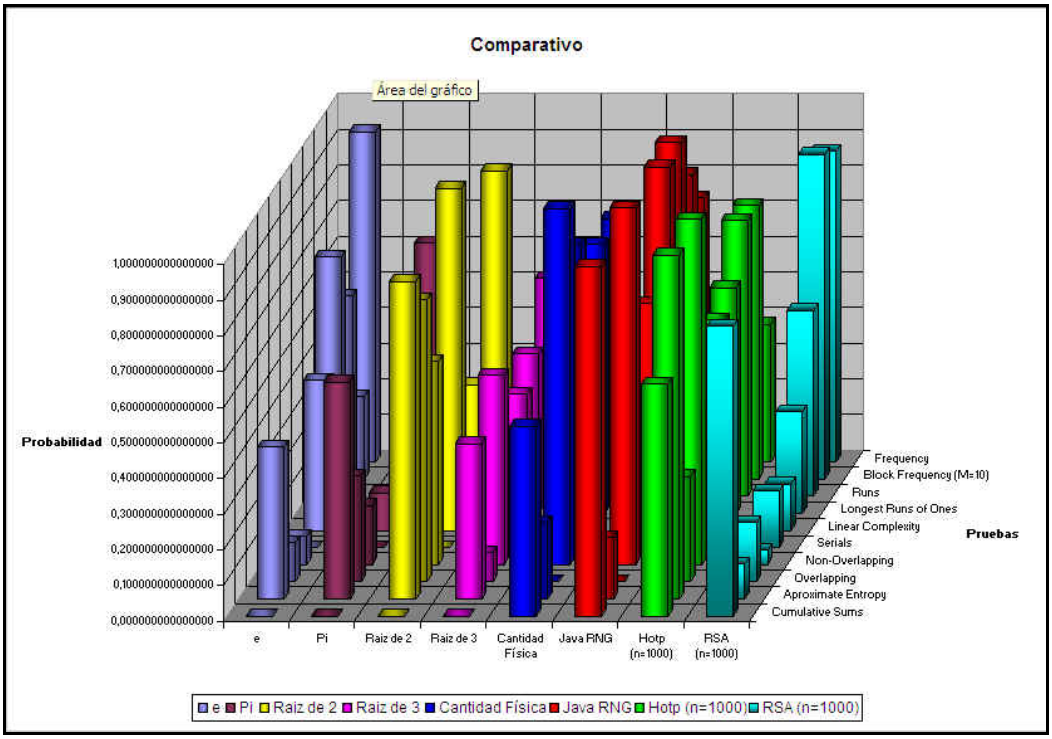


Figura 17. Diagrama de Barras – Comparativo Probabilidad obtenida de aplicar las pruebas a cada tipo de secuencia.

De acuerdo con la ponderación se obtuvieron los siguientes resultados:

Ponderación	
Secuencia	**Total**
Java	16
Hotp	25
Cantidad Física	27
e	30
RSA	31
Raiz de 2	34
Raiz de 3	38
Pi	41
Total general	***242***

Tabla 5. Resultados ponderación total.

Las secuencias con menor valor de ponderación presentaron menores desviaciones o anomalías con respecto a condiciones verdaderamente aleatorias en la mayoría de las pruebas estadísticas sobre las cuales fueron aplicadas.

Java, presenta en la mayoría de pruebas superiores características de aleatoriedad (16) con respecto a las demás secuencias. Hotp (25) se presenta como la segunda clase de secuencia de mayores características de aleatoriedad. A diferencia, RSA (31) dentro del grupo de secuencias evaluadas ocupa el quinto lugar presentando mayores desviaciones con respecto a condiciones verdaderamente aleatorias con respecto a las secuencias generadas por Java, Hotp, la cantidad física y e.

Revisando los resultados obtenidos se puede visualizar que el algoritmo HOTP presenta mayores propiedades de aleatoriedad con respecto a los otros valores seleccionados para la comparación (exceptuando la secuencia generada por el RNG de Java). Los demás valores en comparación incluyendo la solución comercial RSA presentan altos niveles de aleatoriedad en algunas pruebas, pero menores características de aleatoriedad en otras.

La evaluación de aleatoriedad juega un papel importante en la evaluación de la seguridad de algoritmos que hacen uso de generadores de números aleatorios. Los generadores de números aleatorios son un componente importante en la construcción de llaves criptográficas y otros parámetros de seguridad.

Teniendo esto en cuenta, se puede inferir a partir de los resultados obtenidos, que el generador de números aleatorios de HOTP produce valores que pueden ser utilizados por algoritmos criptográficos con mayor confianza que aquellos generados por otros tipos de algoritmos ya que presentan mayores propiedades de aleatoriedad; Lo cual es una característica deseada en criptografía ya que dificulta la labor de los criptoanalistas en el rompimiento de algoritmos criptográficos al reducir la probabilidad de deducir o predecir

los valores aleatorios generados que generalmente son utilizados usados como entrada en algoritmos criptográficos.

Igualmente, se puede concluir que los valores generados por el RNG de Java constituyen una secuencia con propiedades de aleatoriedad que puede ser usada en algoritmos criptográficos de alto nivel de confiabilidad.

5. CONCLUSIONES

En los capítulos anteriores se han expuesto una gran diversidad de métodos y estudios para verificar la seguridad proporcionada por el algoritmo HOTP como generador de números aleatorios.

La comunidad OATH publicó como Internet-Draft en octubre del 2004 – mayo de 2005 el algoritmo HOTP como una alternativa novedosa y abierta para generación de contraseñas de un solo uso.

A diferencia de las soluciones comerciales como RSA SecurID, la solución abierta HOTP no basa su seguridad al funcionar como una caja negra en donde solo se conoce la salida pero no el funcionamiento interno.

Éste algoritmo basa se seguridad principalmente en las propiedades de seguridad ya conocidas y evaluadas de HMAC-SHA1. Propiedades como son la resistencia a las preimagenes y a las colisiones las cuales aseguran que a partir de valores generados por HOTP es difícil obtener el secreto compartido para la generación de nuevos valores (contraseñas).

Adicionalmente, estudios recientes han concluido que a la fecha en la que se escribieron estás letras, la función de dispersión más segura y usada en el mundo es SHA1 con respecto a otras conocidas como son SHA0 y MD5. A futuro podrían implementarse mejoras a la seguridad del algoritmo HOTP al hacer uso de otros tipos de funciones de dispersión (Hash) que brinden mayores niveles de seguridad (p.e. SHA-256).

Otro tipo de mejora que puede efectuarse sobre el algoritmo consiste en dificultar la irreversibilidad de la función de dispersión (Hash) al implementar el algoritmo generando valores (contraseñas) de mayor número de dígitos (p.e. 9 dígitos). Actualmente el algoritmo

soporta la generación de valores de 9 dígitos. A la hora de implementase solo tendría que modificarse el parámetro de entrada "*d*" de 6 dígitos a 7, 8 o 9 sin tener que modificar el código del aplicativo.

Cabe mencionar que el estudio y las pruebas aplicadas a los valores generados por el algoritmo HOTP correspondían a secuencias de 6 dígitos; Esto con el objetivo de tener una mejor comparación con respecto a la solución comercial SecurID de RSA ya que ésta también genera secuencias de 6 dígitos decimales.

La evaluación de la seguridad del algoritmo no se limitó a las propiedades de diseño del mismo. La función principal del algoritmo HOTP es la generación de valores o contraseñas de un solo uso. Al tratarse de un generador de números se debe evaluar la aleatoriedad de éstos valores generados.

La evaluación de aleatoriedad juega un papel importante en la evaluación de la seguridad de algoritmos que hacen uso de generadores de números aleatorios. Los generadores de números aleatorios son un componente importante en la construcción de llaves criptográficas y otros parámetros de seguridad.

Para la evaluación de aleatoriedad se estudiaron e implementaron 10 pruebas estandarizadas por el NIST para la medición de aleatoriedad en secuencias. Las pruebas aplicadas buscaban desviaciones o anomalías con respecto a condiciones verdaderamente aleatorias en las secuencias binarias evaluadas.

Mediante el uso de las pruebas estadísticas se cuantificó la aleatoriedad de los valores generados por el algoritmo (1000 valores) en comparación a otros valores catalogados como aleatorios como son: los resultados obtenidos de la descomposición de un isótopo radioactivo (cantidad física), raíz de dos, raíz de tres, Pi con un millón de dígitos, los

valores generados por el RNG de Java y los valores generados (1000 valores) por el autenticador de RSA Security SecurID.

Las secuencias evaluadas presentaron un mínimo de desviaciones; Para todas las pruebas aplicadas se aceptó la hipótesis nula de aceptación de aleatoriedad.

Revisando los resultados obtenidos se puede visualizar que el algoritmo HOTP presentó mayores propiedades de aleatoriedad con respecto a los otros valores seleccionados para la comparación (exceptuando la secuencia generada por el RNG de Java). Los demás valores en comparación incluyendo la solución comercial RSA presentan altos niveles de aleatoriedad en algunas pruebas, pero menores características de aleatoriedad en otras.

El resultado de las pruebas estadísticas conlleva a concluir que la solución abierta HOTP presenta mayores características de seguridad que la solución comercial de RSA SecurID al generar valores o contraseñas de un solo uso con mayores propiedades de aleatoriedad garantizando así las propiedades de no colisión y pre-imagen heredadas de la función de dispersión que lo implementa.

Al desconocerse el funcionamiento interno de la solución de RAS se dificulta concluir que la solución abierta HOTP posee un diseño más o menos seguro que la solución comercial de RSA. Este mismo desconocimiento del funcionamiento interno de SecurID puede tomarse como una debilidad de la solución ya que si algún día el algoritmo para la generación de valores de RSA es publicado toda la solución podría convertirse en insegura de la noche a la mañana.

El algoritmo HOTP para la generación de contraseñas de un solo uso, al tratarse de una solución abierta puede ser implementado por cualquier distribuidor de hardware o desarrollador de software.

Aun cuando el algoritmo puede ser implementado en cualquier tipo de dispositivo o agente de software y éste posee un diseño seguro (actualmente) y los valores que genera presentan características de aleatoriedad, se recomienda la implementación del mismo en un ambiente seguro.

Las tarjetas inteligentes y las SIM cards se consideran medios seguros ya que el código fuente del algoritmo y los valores secretos como la clave del usuario/administrador y el secreto compartido se encuentran grabados dentro del chip y para acceder a ellos solo se puede hacer mediante el llamado a funciones controladas que permitan la extracción o modificación de éstos parámetros. Técnicas de criptoanálisis o ingeniería reversa resultan difíciles de aplicar ya que en primera instancia deben superar las barreras físicas que brindan éstos tipos de dispositivos.

Para garantizar que ataques de fuerza bruta resulten inefectivos contra éstos tipos de aplicaciones en tarjetas SIM o Java, se recomienda hacer uso de las mejores prácticas de desarrollo ya que se deben implementar mecanismos que garanticen el bloqueo de las tarjetas o del programa cuando se emitan repetidas secuencias de comandos inválidos.

A la fecha en la que se inició el proyecto, no existían soluciones comerciales que implementen este tipo de algoritmos abiertos en una arquitectura de tarjetas inteligentes, o SIM cards justificando un futuro proyecto de desarrollo de estos tipos de algoritmos sobre estas arquitecturas.

El documento hace referencia a múltiples escenarios de implementación segura del algoritmo los cuales pueden ser usados como puntos de referencia para diferentes tipos de soluciones comerciales basadas en HOTP.

Aún cuando no se encuentra dentro del alcance de éste proyecto la implementación segura del algoritmo HOTP, en el anexo del presente documento se puede visualizar el código

fuente para la implementación del algoritmo en tarjetas SIM o Javacards. El código descrito únicamente implementa el algoritmo del lado del cliente como un ejemplo de generadores de valores de HOTP en arquitecturas seguras.

6. REFERENCIAS

[1] R. Rivest. "RFC 1321: The MD5 Message-Digest Algorithm", Network Working Group, Abril 1992, http://www.faqs.org/rfcs/rfc1321.html (2005-01-28)

[2] N. Haller, C. Metz, P. Nesser, M. Straw. "RFC 2289: A One-Time Password System", Network Working Group, Febrero 1998, http://www.ietf.org/rfc/rfc2289.txt (2005-02-02)

[3] Bruce Schneier, David Banisar, "The Electronic Privacy Papers" 2004.

[4] Bruce Schneier, "Applied Cryptography" Segunda Edición, 2001.

[5] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, O. Ranen. "HOTP: An HMAC-based One Time Password Algorithm". RFC – Internet Draft, Octubre de 2004. http://ietfreport.isoc.org/ids/draft-mraihi-oath-hmac-otp-01.txt (2005-01-23)

[6] OATH (Open Authentication membership) Web Site. http://www.openauthentication.org/pr_04_10_26_1.asp (2005-01-10)

[7] "Aladdin etoken", 2004, http://www.ealaddin.com/eToken (2005-01-22)

[8] RSA Security Inc., "RSA Security Web Page", 2005, http://www.rsasecurity.com (2005-02-03)

[9] RSA Security Inc., "RSA SecureID Authentication", 2005, http://www.rsasecurity.com/products/securid/datasheets/SID_DS_0804_lowres.pdf (2005-02-03)

[10] S. Crocker, J. Schiller, "Randomness Recommendations for Security", Network Working Group, Diciembre 1994. http://www.ietf.org/rfc/rfc1750.txt (2005-04-13)

[11] Andrew Rukhin. Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo, "A statictical test suite for random and pseudorandom number generators for cryptographic applications", NIST Institute, Mayo 15 de 2001, http://csrc.nist.gov/rng/SP800-22b.pdf (2005-01-30)

[12] "United States Patent and Trademark Office", 2005, http://www.uspto.gov (2005-02-16)

[13] H. Krawczyk, M. Bellare, R. Canetti , "RFC 2104 - HMAC: Keyed-Hashing for Message Authentication", Febrero 1997, http://www.faqs.org/rfcs/rfc2104.html (2005-02-17)

[14] D. Eastlake, P. Jones, "RFC 3174 - US Secure Hash Algorithm 1 (SHA1)", Septiembre de 2001, http://www.faqs.org/rfcs/rfc3174.html (2005-02-17)

[15] NIST (National Institute of Standards and Technology), "Random Number Generation and Testing", diciembre 2004, http://csrc.nist.gov/rng (2005-02-28)

ANEXO 1 – CÓDIGO FUENTE (JAVACARD) PARA IMPLEMENTACIÓN DE HOTP SOBRE CELULARES

Objetivo

Programa que busca ilustrar el funcionamiento básico de un esquema de autenticación haciendo uso del algoritmo HOTP para generación de contraseñas de un solo uso en un ambiente de celulares.

Código Fuente

El código fuente presentado a continuación fue compilado haciendo uso del JDK de Java 1.4.2_04.

Para el desarrollo sobre el ambiente de celulares (midlet) se uso el J2ME Gíreles Toolkit y el los plug-ins Eclipseme 0.94.

```
/*
 * Created on 26/03/2005
 *
 * CLASE HOTPMIDLET V.1.0
 * Clase donde se implementa el midlet de HOTP para su ejecución en el J2ME Wireless
 * toolkit.
 */

/**
 * @author Sergio Garcia - 200217365
 *
 * Universidad de los Andes
 * Proyecto de Grado - Magister en Ing. de Sistemas
 *
 * Asesor: Milton Quiroga
 */


import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.bouncycastle.crypto.digests.SHA1Digest;
import org.bouncycastle.crypto.macs.HMac;
import org.bouncycastle.crypto.params.KeyParameter;


public class hotpMidlet extends MIDlet implements CommandListener
{
```

```
//DATOS MIEMBRO
HMac hmac = new HMac(new SHA1Digest());
byte[] resBuf = new byte[hmac.getMacSize()];
int counter = 0;



private Command exitCommand;
private Command aceptarCommand;
private Display display;
private Form screen;
public TextField clave;

public hotpMidlet()
{
  // Objeto display del midlet.
  display = Display.getDisplay(this);

  // Comando salir
  exitCommand = new Command("Salir", Command.EXIT,2);

  // Comado aceptar
  aceptarCommand = new Command("OK", Command.OK,2);

  // Creacion de formulario principal
  screen = new Form("Menu");

  // Constructor del cuadro de texto
  clave = new TextField( "Digite su Clave" , "" ,50 , TextField.ANY );
  screen.append(clave);

  // Adicionar comandos y clase que los manejara
  screen.addCommand(exitCommand);
  screen.addCommand(aceptarCommand);
  screen.setCommandListener(this);
}

/** STARTAPP()
 * Inicia el aplicativo (midlet)
***/
public void startApp() throws MIDletStateChangeException
{
  // Seleccionamos la pantalla a mostrar
  display.setCurrent(screen);
}

/** PAUSEAPP()
 * Pausa el aplicativo -> "Incoming Call"
***/
public void pauseApp()
{
}

/** DESTROYAPP()
 * Elimina el aplicativo (Cerrar)
***/
public void destroyApp(boolean incondicional)
{
}

/** COMMANDACTION()
 * Ejecución de comandos
***/
public void commandAction(Command c, Displayable s)
{

      // Salir
  if (c == exitCommand)
  {
    destroyApp(false);
    notifyDestroyed();
  }
  else //if (c == ok)
  {
      funcionHash(clave.getString());
```

```
    String cadena = hotp();
         StringItem resultado = new StringItem( "Clave HOTP" , cadena);
         screen.append(resultado);
         display.setCurrent(screen);
        counter++;

    }
  }

  /** FUNCION_HASH()
   * Funcion de BouncyCastle para generacion de HMAC-SHA1
  ***/
   public void funcionHash(String clave)
  {

    byte[] mensaje = new byte[8];
    mensaje[7]=(byte)this.counter;;
    hmac.init(new KeyParameter(clave.getBytes()));
    hmac.update(mensaje, 0, mensaje.length);
    hmac.doFinal(resBuf,0);
  }


   /** HOTP()
    * Funcion que calcula valor de HOTP
   ***/

  public String hotp()
  {
    byte offset = resBuf[19];
    // And con 00001111 = 15 para obtener los 4 bits menos significativos
    byte nible  = (byte) (offset & 15);
    int posicion = (int) nible;

    byte resultado[] = new byte[4];

    for (int i = 0 ; i < 4 ; i++)
      resultado[i] = resBuf[posicion + i];

    //And con 01111111 para eliminar el primer bit
    resultado[0]=(byte)(resultado[0] & 127);

    BigInteger resultadoFinal = new BigInteger(resultado);
    BigInteger modulo = new BigInteger("1000000");

    resultadoFinal = resultadoFinal.mod(modulo);
    System.out.println(resultadoFinal.toString());
    return resultadoFinal.toString();
  }

}
```

ANEXO 2 – CÓDIGO FUENTE (JAVACARD) PARA IMPLEMENTACIÓN DE HOTP SOBRE TARJETAS INTELIGENTES

X Nota: *Únicamente se implementó el algoritmo en la tarjeta inteligente ya que el objetivo del proyecto es el generar valores de HOTP haciendo uso de una arquitectura segura como lo es el de las tarjetas inteligentes.*

Objetivo

Programa que busca ilustrar el funcionamiento básico de un esquema de autenticación haciendo uso del algoritmo HOTP para generación de contraseñas de un solo uso en un ambiente de tarjetas inteligentes.

Funcionamiento

AID del Paquete - jchotp	
Valor	Longitud
0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1	10 bytes

AID del Applet - JChotp	
Valor	Longitud
0xa0:0x0:0x0:0x0:0x0:0x62:0x0:0x0:0x0:0xC:0x5:0x1	10 bytes

La siguiente tabla presenta el listado de métodos definidos en la clase JChotp:

Resumén de Métodos	
Public void	**deselect()** Invocado por el JCRE para informar que el Applet que se encuentra activo será desactivado ya sea por retiro de la tarjeta del lector o por un llamado a otro applet.

Private void	**CMD_AUTHSC(APDU apdu)** Método propietario donde se lleva a cabo la generación del valor HOTP una vez validado el PIN del usuario.
Private void	**CMD_GET_DATA(APDU apdu)** Método usado para la extracción de información de la tarjeta inteligente. En la versión actual del programa solo permite recuperar el número serial (scid) de la tarjeta.
Private void	**CMD_PUT_DATA(APDU apdu)** Método usado para la inserción de parámetros a la tarjeta. Este método se debe invocar la primera vez que se use la tarjeta para modificar los parámetros por defecto. Los parámetros que se pueden modificar son: • PIN del usuario • PIN del administrador • Llave para autenticación • Tiempo de expiración de la tarjeta (dado en número máximo de autenticaciones permitidas). • Número serial de la tarjeta (SCID).
Private void	**CMD_SELECT(APDU apdu)** Método invocado por el JCRE para informar de la selección y activación del applet.
Private void	**CMD_VERIFY(APDU apdu)** Método usado para verificar la valides del PIN de usuario o administrador ingresado.
Public static void	**Install(byte[] buffer, short offset, byte length)** El JCRE invoca este método estático para crear una instancia del applet.
Public void	**Process(APDU apdu)** Método invocado por el JCRE para procesar un APDU entrante. Este método hace el llamado a los diferentes métodos implementados en el applet dependiendo del tipo de comando recibido en la APDU.
Public Void	**ReceiveAPDUbody(APDU apdu)** Método que valida recepción de datos en el cuerpo de la APDU entrante.

Protected	**JChotp(byte[] buffer, short offset, byte length)** Constructor de la clase JChotp donde se inicializan todos los objetos que serán utilizados a lo largo de la vida del applet. Los objetos iniciados en la clase install() se inician una sola vez a lo largo de la vida del applet.

Toda comunicación entre el CAD (Card Aceptante Device: Lector de Tarjetas) y la tarjeta inteligente se lleva a cabo mediante el envío de APDUs de comandos y la recepción de APDUs de respuestas.

La tarjeta inteligente actúa como un medio pasivo; solo realiza algún tipo de acción cuando es solicitada por el CAD mediante una APDU de comando.

A continuación se describen las APDUs de comando y respuesta usadas en la comunicación con el applet:

SELECT						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xa4	0x04	0x00	0x0a	0xa0 0x0 0x0 0x0 0x62 0x0 0x0 0xc 0x5 0x1	N/a

VERIFY (userpin)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0x20	0x00	0x01	0x02	< 2 bytes con PIN del usuario> - 0x00 0x00	N/a

VERIFY (adminpin)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0x20	0x00	0x01	0x02	< 3 bytes con PIN del admin> - 0x22 0x22 0x22	N/a

PUT DATA (userpin)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xda	0x00	0x51	0x02	< 2 bytes con PIN del usuario> - 0x00 0x00	N/a

PUT DATA (adminpin)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**

0x00	0xda	0x00	0x52	0x03	< 3 bytes con PIN del admin> - 0x00 0x00 0x00	N/a

PUT DATA (Authkey)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xda	0x00	0x53	0x08	< 20 bytes con llave HMAC-SHA1> - 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0	N/a

PUT DATA (# máx. de autenticaciones)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xda	0x00	0x54	0x02	< 2 bytes con máx. AUTH > - 0x00 0x00	N/a
PUT DATA (scid)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xda	0x00	0x55	0x04	< 4 bytes con serial> - 0x0 0x0 0x0 0x0	N/a

GET DATA (scid)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x00	0xca	0x00	0x55	0x00		0x04

GET DATA (scid)			
APDU de Respuesta			
Datos	**SW1**	**SW2**	**Descripción**
<4 bytes con scid> - 0x00 0x00 0x00 0x00	90	00	Procesamiento Exitoso

AUTHSC (autenticar tarjeta)						
APDU de Comandos						
CLA	**INS**	**P1**	**P2**	**Lc**	**Datos**	**Le**
0x80	0x08	0x00	0x00			4

AUTHSC (autenticar tarjeta)			
APDU de Respuesta			
Datos	**SW1**	**SW2**	**Descripción**
<4 bytes conformado por el valor HOTP en hexádecimal>	90	00	Procesamiento Exitoso

Las respuestas emitidas por las tarjetas poseen un código llamado “Status Word” el cual indica el resultado del procesamiento. Existen diversos números de status words para indicar lo que esta sucediendo internamente durante el procesamiento de comandos.

A continuación se presenta un listado de los Status Words que pueden ser emitidos como respuestas ante comandos:

Listado de Status Words posibles	
Status Word	Descripción
90 00	**SW_NO_Error** Procesamiento exitoso
6E 00	**SW_CLA_NOT_SUPPORTED** Valor de CLA inválido.
6D 00	**SW_INS_NOT_SUPPORTED** Valor de INS inválido.
69 86	**SW_COMMAND_NOT_ALLOWED** Comando no soportado
69 85	**SW_CONDITIONS_NOT_SATISFIED** Procesamiento inválido – no se cumplen las condiciones de funcionamiento del applet.
69 84	**SW_DATA_INVALID** Dato inválido.
6A 84	**SW_FILE_FULL** No hay suficiente espacio de memoria en el archivo.
6A 82	**SW_FILE_NOT_FOUND** Archivo no encontrado.
6A 86	**SW_INCORRECT_P1P2** Parámetros P1 o P2 incorrectos.
69 82	**SW_SECURITY_STATUS_NOT_SATISFIED** No se cumplen con las condiciones de seguridad.
6A 80	**SW_WRONG_DATA** Datos inválidos.
67 00	**SW_WRONG_LENGTH** Longitud de datos inválida.
6B 00	SW_WRONG_P1P2 Parámetros P1 o P2 incorrectos.

63 C#	**SW_PIN_FAILED** PIN inválido. # indica el número de intentos restantes antes de que el PIN se bloquee.
69 99	**SW_APPLET_SELECTED_FAILED** Error durante la selección del Arlet.
6F 00	**SW_UNKNOWN** Error en el funcionamiento del Applet; Causa desconocida.

Código Fuente

El código fuente presentado a continuación fue compilado haciendo uso del JDK de Java 1.3.1_12 ya que este no presenta problemas de incompatibilidad con el framework de Javacard.

Para el desarrollo sobre el ambiente de tarjetas inteligentes se uso el kit de desarrollo de Java (JCDK 2.2.1) el cual incluye el framework para Javacard 2.2.1.

```
//-------------------------------------------------------------------------------------------------------
//
//ALGORITMO HOTP - for Javacard
//
//Desarrollado por: SERGIO GARCIA - serg-gar@uniandes.edu.co, sergegar@hotmail.com
//                                  200217365
//                                  Universidad de los Andes
//
//El software desarrollado se basa en los ejemplos de programacion sobre
//java cards de Wolfgang Rank (Wolfgang.Rank@de.gi-de.com) y Ziqun Chen (zhiqun.chen@sun.com)
//
//-------------------------------------------------------------------------------------------------------
//
//Archivo: JChotp.java (Version 1.0)
//Paquete: jchotp
//
//Package AID (jchotp): 0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1  - A000000062030 10C0601
//Applet AID (JChotp) : 0xa0:0x0:0x0:0x0:0x62:0x0:0x0:0xC:0x5:0x1  - A000000062000 10C0501
//
//-------------------------------------------------------------------------------------------------------
//Programa que busca la identificacion de un usuario mediante un PIN y su posterior autenticacion
//mediante la generaciones de valores de HOTP como contraseñas de un solo uso.
//
//1.)   SMART CARD                                                                    CAD
//
//                                        <--------------------    APDU VERIFY USER o ADMIN PIN
//
//        Sucess o fail  ---------------------->   Si: SUCESS => ir a 2.)
//
//
//SCID: Nro. Serial de la Tarjeta. (Valor público) --> Puede ser consultado a la tarjeta.
```

```
//AUTHKEY: Llave (Secreto - Valor HMAC-SHA1) conocido solo por la tarjeta y su emisor.
//
//2.)    SMART CARD                                                                    CAD
//
//                                          <--------------------    x = HMAC-SHA1
//
//                y = HOTP      ---------------------->
//
//-------------------------------------------------------------------------------------------------------

package jchotp;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.JCSystem;
import javacard.framework.OwnerPIN;
import javacard.framework.Util;
import javacard.security.Signature;

public class JChotp extends Applet {

  // Clases (CLA) e Instrucciones (INS)
  final static byte      PROP_CLASS  = (byte) 0x80;     // Clase para APDUs de comandos propietarios
(00: JCRE)
  final static byte      INS_SELECT  = (byte) 0xA4;     // ISO/IEC 7816-4 comando SELECT Applet
  final static byte      INS_VERIFY  = (byte) 0x20;     // ISO/IEC 7816-4 comando VERIFY
  final static byte      INS_PUTDATA = (byte) 0xDA;     // ISO/IEC 7816-4 comando PUT DATA
  final static byte      INS_GETDATA = (byte) 0xCA;     // ISO/IEC 7816-4 comando GET DATA
  final static byte      INS_AUTHSC  = (byte) 0x08;     // Comando propietario para Autenticar Smart
Card

  // Tipos especificos de codigos de errores (Excepciones)
  final static short     SW_PIN_FAILED =     (short) 0x63C0;  // Error: Verificacion de PIN fallida (El
ultimo nibble especifica # de intentos restantes)
  final static short     SW_DATA_NOT_FOUND = (short) 0x6A88;  // Error: No se encontro el objeto

  // Parametros para comando de Autenticacion (AUTHSC)
  final static byte      LEN_CAPDU_AUTHSC = (byte)   8;   // Longitud de datos en comando APDU de AUTHSC
  final static byte      LEN_RAPDU_AUTHSC = (byte)  30;   // Longitud de datos en respuesta APDU de
AUTHSC
  final static short     INDEX_AUTHKEY    = (short)  0;   // Offset de inicio para Llave de
autenticacion de Smart Card en respuesta AUTHSC
  final static short     INDEX_OFFSET     = (short) 20;   // Offset de desplazamiento para Llave de
autenticacion de Smart Card en respuesta AUTHSC

  // Elementos de Datos
  static short           authcntr;                        // Contador de # max. de autenticaciones (2
bytes)
  final static short     SIZE_AUTHCNTR  = (short)  2;     // Tamaño de contador de autenticaciones
  private byte[]        scid;                            // Identificador de la Smart Card(S/N)
  final static short     SIZE_SCID      = (short)  4;     // Tamaño de identificador de Smart Card
  private byte[]         workarray;                       // Arreglo de trabajo
  final static short     SIZE_WORKARRAY = (short) 30;     // Tamano del arreglo de trabajo
  private byte[]         authkey;                         // Llave de autenticacion
  final static short     SIZE_AUTHKEY   = (short) 20;     // Tamaño de llave de autenticacion (Llave de
8 bytes - DES key)
  private Signature     sig;                                                  // Firma digital (MAC:
Message Authentication Code) de 8 Bytes

  // Etiquetas de elementos a los que se les pueden modificar valores (PUT DATA)
  final static short     TAG_USERPIN  = (short) 0x51;   // Etiqueta para USERPIN
  final static short     TAG_ADMINPIN = (short) 0x52;   // Etiqueta para ADMINPIN
  final static short     TAG_AUTHKEY  = (short) 0x53;   // Etiqueta para AUTHKEY
  final static short     TAG_AUTHCNTR = (short) 0x54;   // Etiqueta para AUTHCNTR
  final static short     TAG_SCID     = (short) 0x55;   // Etiqueta para SCID

  // Constantes y Variables para administracion de ADMINPIN
  private byte[] DEFAULT_ADMINPIN=
  {(byte)0x22,
   (byte)0x22,
   (byte)0x22 };                                                    // Valor x defecto de
ADMINPIN
  final static byte      ADMINPIN_SIZE         = (byte) 3;   // Tamaño del ADMINPIN
```

```
  final static byte      DEFAULT_ADMINPIN_MAXEC = (byte) 2;   // # max de Errores de digitacion de
ADMINPIN
  final static byte      ADMINPIN_ID            = (byte) 2;   // Identificador de PIN identifier, PIN 2
= ADMINPIN
  private OwnerPIN       adminpin;                            // Objeto PIN

  // Constantes y Variables para administracion de USERPIN
  private byte[] DEFAULT_USERPIN=
  {(byte)0x00,
   (byte)0x00 };                                                          // Valor x defecto de USERPIN
  final static byte      USERPIN_SIZE          = (byte) 2;    // Tamaño del USERPIN
  final static byte      DEFAULT_USERPIN_MAXEC = (byte) 3;    // # max de Errores de digitacion de
USERPIN
  final static byte      USERPIN_ID            = (byte) 1;    // Identificador de PIN identifier, PIN 2
= USERPIN
  private OwnerPIN       userpin;                             // Objeto PIN

  // Constantes y Variables para administracion de la llave
  private byte[] DEFAULT_AUTHKEY=
  {(byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00,
   (byte)0x00 };                                                            // Valor de llave de
autenticacion por defecto
  final static byte     AUTHKEY_SIZE          = (byte) 20;    // Tamaño de llave de autenticacion

  /** ***************************************************************************************
        INSTALL - Instalacion del applet
  ********************************************************************************** **/
  public static void install(byte[] buffer, short offset, byte length) {
        // Crea instancia ZKPid applet
        new JChotp(buffer, offset, length);
  }

  /** ***************************************************************************************
  Constructor - Registro del Applet
  ********************************************************************************** **/
  protected JChotp(byte[] buffer, short offset, byte length) {

        // Objeto para Identificacion de Smart Card
        scid = new byte[SIZE_SCID];

        //      Objeto para almacenamiento de llave de autenticacion (HMAC-SHA1)
        authkey = new byte[SIZE_AUTHKEY];

        // Creacion de multiples objetos en RAM por medio de un arreglo de bytes
        workarray = JCSystem.makeTransientByteArray(SIZE_WORKARRAY, JCSystem.CLEAR_ON_DESELECT);

        // Copia de valor por defecto de llave de autenticacion a authkey
        Util.arrayCopy(DEFAULT_AUTHKEY, (short) (0), authkey, INDEX_AUTHKEY, SIZE_AUTHKEY);

        // Creacion de objeto para ADMINPIN y valores por defecto
        adminpin = new OwnerPIN(DEFAULT_ADMINPIN_MAXEC, ADMINPIN_SIZE);
        adminpin.update(DEFAULT_ADMINPIN, (short)(0), ADMINPIN_SIZE);

        // Creacion de objeto para USERPIN y valores por defecto
        userpin = new OwnerPIN(DEFAULT_USERPIN_MAXEC, USERPIN_SIZE);
        userpin.update(DEFAULT_USERPIN, (short)(0), USERPIN_SIZE);

        // Registro de aplicativo ante JCRE
```

```
        register();
    }


    /** *************************************************************************************
            PROCESS - Procesamiento de Comandos - Menu
    ************************************************************************************* **/
    public void process(APDU apdu) {
        byte[] cmd_apdu = apdu.getBuffer();

        // Elimina variables globales usadas
        Util.arrayFillNonAtomic(workarray, (short) 0, SIZE_WORKARRAY, (byte) 0);

        if (cmd_apdu[ISO7816.OFFSET_CLA] == ISO7816.CLA_ISO7816) {
          //CLASE ISO/IEC 7816 - "00"
          switch(cmd_apdu[ISO7816.OFFSET_INS]) {
                case INS_SELECT:
                  // SELECT
                  cmdSELECT(apdu);
                  break;
                case INS_VERIFY:
                  // VERIFY
                  cmdVERIFY(apdu);
                  break;
                case INS_PUTDATA:
                  // PUT DATA
                  cmdPUTDATA(apdu);
                  break;
                case INS_GETDATA:
                  // GET DATA
                  cmdGETDATA(apdu);
                  break;
                default :
                  // Intruccion NO soportada
                  ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
          }
        }
        else if (cmd_apdu[ISO7816.OFFSET_CLA] == PROP_CLASS) {
          //CLASE PROPIETARIA - "80"
          switch(cmd_apdu[ISO7816.OFFSET_INS]) {
                case INS_AUTHSC:
                  // AUTHSC
                  cmdAUTHSC(apdu);
                  break;
                default :
                  // Intruccion NO soportada
                  ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
          }
        }
        else {
          // Intruccion NO soportada
          ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
        }
    }

    /** *************************************************************************************
            DESELECT - Reset del Applet
    ************************************************************************************* **/
    public void deselect() {

        // Reset de los PIN
        adminpin.reset();
        userpin.reset();
    }


    /** *************************************************************************************
        SELECT - Seleccionar Applet
    ************************************************************************************* **/
    private void cmdSELECT(APDU apdu) {
        byte[] cmd_apdu = apdu.getBuffer();

        //Verifica condiciones en encabezado SELECT
        // Verifica P1='04'
        if (cmd_apdu[ISO7816.OFFSET_P1] != 0x04) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
```

```
        }
        // Verifica que P2='00'
        if (cmd_apdu[ISO7816.OFFSET_P2] != 0x00) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        short lc = (short)(cmd_apdu[ISO7816.OFFSET_LC] & 0x00FF);
        receiveAPDUBody(apdu);

        // No se encontro el Applet seleccionado
        if (JCSystem.getAID().equals(cmd_apdu, ISO7816.OFFSET_CDATA, (byte) lc) == false) {
          ISOException.throwIt(ISO7816.SW_APPLET_SELECT_FAILED);
        }

        // Applet Seleccionado existosamente
        ISOException.throwIt(ISO7816.SW_NO_ERROR);   //Finalizo Correctamente
}


/** ************************************************************************************
        VERIFY - Verifica (identifica) usuario tipo normal (USERPIN) o administrador (ADMINPIN)
************************************************************************************ **/
private void cmdVERIFY(APDU apdu) {

        byte[] cmd_apdu = apdu.getBuffer();

        // Validaciones en Encabezado
        // Verifica P1='00'
        if (cmd_apdu[ISO7816.OFFSET_P1] != 0) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        short lc = (short)(cmd_apdu[ISO7816.OFFSET_LC] & 0x00FF);
        receiveAPDUBody(apdu);

        if (cmd_apdu[ISO7816.OFFSET_P2] == USERPIN_ID) {  // USERPIN - Sesion de trabajo
          if (lc != USERPIN_SIZE) {
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
          }
          if (userpin.check(cmd_apdu, ISO7816.OFFSET_CDATA, USERPIN_SIZE) == false) {
                // PIN invalido
                short tries = userpin.getTriesRemaining();
                ISOException.throwIt( (short) (SW_PIN_FAILED + tries));
          }
        }
        else if (cmd_apdu[ISO7816.OFFSET_P2] == ADMINPIN_ID) {    // ADMINPIN - Administracion
          if (lc != ADMINPIN_SIZE) {
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
          }
          if (adminpin.check(cmd_apdu, ISO7816.OFFSET_CDATA, ADMINPIN_SIZE) == false) {
                // PIN invalido
                short tries = adminpin.getTriesRemaining();
                ISOException.throwIt( (short) (SW_PIN_FAILED + tries));
          }
        }
        else { // PIN no valido (No identifica un usuario)
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        // Verificacion de PIN exitosa
        ISOException.throwIt(ISO7816.SW_NO_ERROR);   //Finalizo Correctamente
}


/** ************************************************************************************
        PUT DATA - Actualiza Contadores y Variables
************************************************************************************ **/
private void cmdPUTDATA(APDU apdu) {
        byte[] cmd_apdu = apdu.getBuffer();

        // Validaciones en Encabezado
        // Verifica P1=0
        if (cmd_apdu[ISO7816.OFFSET_P1] != 0) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        // Verifica que P2 contenga etiqueta permitida
        short tag = (short) (cmd_apdu[ISO7816.OFFSET_P2] & (short) 0x00FF);
        if ((tag < (short) 0x0040) || (tag > (short) 0x00FE)) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
```

```
        }
        short lc = (short)(cmd_apdu[ISO7816.OFFSET_LC] & (short) 0x00FF);
        receiveAPDUBody(apdu);

        if (adminpin.isValidated() == false) {
          ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }

        switch(tag) {
          case TAG_USERPIN:
                // Cambio de USERPIN
                if (lc != USERPIN_SIZE) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                Util.arrayCopy(cmd_apdu, (short)((ISO7816.OFFSET_CDATA) & 0x00FF), workarray, (short) 0,
lc);
                userpin.update(workarray, (short) 0, USERPIN_SIZE);
                break;
          case TAG_ADMINPIN:
                // Cambio de ADMINPIN
                if (lc != ADMINPIN_SIZE) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                Util.arrayCopy(cmd_apdu, (short)((ISO7816.OFFSET_CDATA) & 0x00FF), workarray, (short) 0,
lc);
                adminpin.update(workarray, (short) 0, ADMINPIN_SIZE);
                break;
          case TAG_AUTHKEY:
                // Cambio de AUTHKEY
                if (lc != SIZE_AUTHKEY) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                Util.arrayCopy(cmd_apdu, (short)((ISO7816.OFFSET_CDATA) & 0x00FF), authkey, (short) 0,
lc);
                break;
          case TAG_AUTHCNTR:
                // Cambio de AUTHCNTR
                if (lc != SIZE_AUTHCNTR) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                Util.arrayCopy(cmd_apdu, (short)((ISO7816.OFFSET_CDATA) & 0x00FF), workarray, (short) 0,
lc);
                if (Util.getShort(workarray, (short)0) > 0x7FFF)
ISOException.throwIt(ISO7816.SW_WRONG_DATA);
                authcntr = Util.getShort(workarray, (short)0);
                break;
          case TAG_SCID:
                // Cambio de SCID
                if (lc != SIZE_SCID) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                Util.arrayCopy(cmd_apdu, (short)((ISO7816.OFFSET_CDATA) & 0x00FF), scid, (short) 0, lc);
                break;
        default :
          // Comando no soportado
          ISOException.throwIt(SW_DATA_NOT_FOUND);
        }
        // Datos actualizados correctamente
        ISOException.throwIt(ISO7816.SW_NO_ERROR);    //Finalizo Correctamente
  }


  /** ***************************************************************************
        GET DATA - Extrae valores de parametros y variables
  ***************************************************************************** **/
  private void cmdGETDATA(APDU apdu) {
        byte[] cmd_apdu = apdu.getBuffer();

        // Validaciones en Encabezado
        // Verifica P1=0
        if (cmd_apdu[ISO7816.OFFSET_P1] != 0) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }
        // Verifica P2 en rango valido para etiquetas (0x40 - 0xFE)
        short tag = (short) (cmd_apdu[ISO7816.OFFSET_P2] & (short) 0x00FF);
        if ((tag < 0x40) || (tag > 0xFE)) {
          ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        }

        switch(tag) {       // Se pueden adicionar etiquetas
          case TAG_SCID:
                // Recuperar Identificador de Smart Card (SCID)
                short le = apdu.setOutgoing();
                if (le != SIZE_SCID) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

                apdu.setOutgoingLength((byte)4);
```

```
                Util.arrayCopy(scid, (short) (0), cmd_apdu, (short) 0, SIZE_SCID);

                apdu.sendBytes((short)0, (short)SIZE_SCID); //Finalizo Correctamente

                break;
        default :
          // Etiqueta NO encontrada
          ISOException.throwIt(SW_DATA_NOT_FOUND);
        }
  }


  /** ************************************************************************************
        AUTHSC - Programa propiestario para autenticacion de Smart Card
  ************************************************************************************ **/
  private void cmdAUTHSC(APDU apdu) {

        byte[] cmd_apdu = apdu.getBuffer();

        // Validaciones en Encabezado
        // Verifica que P1=0
        if (cmd_apdu[ISO7816.OFFSET_P1] != 0) ISOException.throwIt(ISO7816.SW_WRONG_P1P2);
        // Verifica que P2=0
        if (cmd_apdu[ISO7816.OFFSET_P2] != 0) ISOException.throwIt(ISO7816.SW_WRONG_P1P2);

        // Valida que no se exceda del nro. max de autenticaciones permtidas
        if (authcntr < 0) ISOException.throwIt(SW_PIN_FAILED);

        //      Valida verificacion de PIN antes de autenticar la SC
         if ((adminpin.isValidated() == false) && (userpin.isValidated() == false)) {
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);    // No se ha
identificado con PIN
         }

        // Datos para generacion del valor HOTP
        Util.arrayCopy(authkey, INDEX_AUTHKEY, workarray, (short) (0), SIZE_AUTHKEY);

        workarray[INDEX_OFFSET] = (byte)(workarray[19] & 0xf);

        workarray[21] = (byte)(workarray[workarray[INDEX_OFFSET]] & 0x7f);
        workarray[22] = (byte)(workarray[workarray[INDEX_OFFSET]+1] & 0xff);
        workarray[23] = (byte)(workarray[workarray[INDEX_OFFSET]+2] & 0xff);
        workarray[24] = (byte)(workarray[workarray[INDEX_OFFSET]+3] & 0xff);

        workarray[25] = (byte) (((short)workarray[21]) << 24);
        workarray[26] = (byte) (((short)workarray[22]) << 16);
        workarray[27] = (byte) (((short)workarray[23]) << 8);
        workarray[28] = (byte) (((short)workarray[24]));

        workarray[29] = (byte) (((short)workarray[21]) << 24
                | ((short)workarray[22]) << 16
                | ((short)workarray[23]) << 8
                | ((short)workarray[24]));

        authcntr = (short) (authcntr - (short) 1);  // Actualiza # restantes de autenticaciones posibles

        // Envio de Respuesta (Response)
        apdu.setOutgoing();
        apdu.setOutgoingLength(LEN_RAPDU_AUTHSC);
        apdu.sendBytesLong(workarray, (short) 0, LEN_RAPDU_AUTHSC);
  }


  /** ************************************************************************************
        RECEIVE APDU - Comando para recepcion de los datos de las APDUs
  ************************************************************************************ **/
  public void receiveAPDUBody(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        short lc = (short)(buffer[ISO7816.OFFSET_LC] & 0x00FF);

        if (lc != apdu.setIncomingAndReceive()) {
          ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
  }
}
```

ANEXO 3 – CÓDIGO FUENTE PARA IMPLEMENTACIÓN DE PRUEBAS ESTADÍSTICAS DE ALEATORIEDAD

Programa que implementa (Clean-room implementation) 10 pruebas para la medición de aleatoriedad estandarizadas por el NIST.

```
/*
 * Created on 26/03/2005
 *
 * CLASE RANDOMNESSTEST1 V.1.0
 * Clase donde se implementa la lectura de los archivos (binarios, hexadecimales o generados)
 * con los datos de entrada. Se almacenan estos datos en una secuencia sobre la cual
 * se aplican las pruebas estadísticas de aleatoriedad definidas por el NIST.
 *
 * Clase que implementa los métodos:
 *       - Frequency
 *       - Block Frequency
 *       -- Runs
 *       -- Longest Run of Ones
 *  --  Linear Complexity
 *  --  Serial Test
 *       - Aproximate Entropy
 *       - Cumulative Sums
 *       -- NonOverlapping Template Matching
 *       -- Overlapping Template Matching
 *
 *  Se pueden aplicar las pruebas sobre archivos binbarios. También se pueden correr las pruebas
 *  sobre un archivo hexadecimal generado por el RNG de Java.
 */

/**
 * @author Sergio Garcia - 200217365
 *
 * Universidad de los Andes
 * Proyecto de Grado - Magister en Ing. de Sistemas
 *
 * Asesor: Milton Quiroga
 */

package RandomTests;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Random;
import util.*;


public class RandomnessTests1 {
        StringBuffer e;
        static double alfa;

/*** CONSTRUCTOR
 * @Method RandomnessTest1()
 * Constructor de la Clase: Asigna la secuencia llena a la nueva instancia de la clase.
 * @param  StringBuffer sec: Secuencia que contiene los datos del archivo binario.
***/
public RandomnessTests1(StringBuffer sec){
        e = sec;         // Secuencia
        alfa = 0.01;     // Nivel de significancia
}

 /** FROMBINTEXTFILE()
  * @Method fromBinTextFile()
```

```
  * Funcion que retorna una nueva instancia de la clase RandomnessTest con la cadena de bits cargada a
partir de los datos de entrada de un archivo binario.
  * @param  String f: Archivo de texto con caracteres 0, 1 (Otros tipos de caracteres son descartados).
  * @return RandomnessTest: Nueva instancia de la clase RandomnessTest con secuencia binaria cargada.
 ***/
public static RandomnessTests1 fromBinTextFile(String f) throws Exception {

        BufferedReader br = new BufferedReader(new FileReader(f)); // Buffer de entrada
        StringBuffer out = new StringBuffer("");                                    // Buffer de salida
        String cad = "";
        // Cadena

        cad = br.readLine();                                        // Lee la primera línea del archivo
        while(cad != null){
          out.append(utilidades.toBin(cad, 1));   // Adjunta a la secuencia solo 0's y 1's
          cad = br.readLine();                                      // Lectura linea x linea
        }
        return new RandomnessTests1(out);                 // Nueva instancia con secuencia binaria
}


 /** FROMHEXTEXTFILE()
  * @Method fromHexTextFile()
  * Funcion que retorna una nueva instancia de la clase RandomnessTests1 con la cadena de bits cargada a
partir de los datos de entrada de un archivo hexadecimal.
  * @param   String f: Archivo de texto con caracteres Hex (0-F) (Otros tipos de caracteres son
descartados).
  * @return RandomnessTests1: Nueva instancia de la clase RandomnessTests1 con secuencia binaria cargada.
 ***/
public static RandomnessTests1 fromHexTextFile(String f)throws Exception {

        StringBuffer out = new StringBuffer("");
        BufferedReader br = new BufferedReader(new FileReader(f));
        String cad = "";

        cad = br.readLine();
        while(cad != null){
          out.append(utilidades.HextoBin(new StringBuffer(cad)));
          cad = br.readLine();
        }
        return new RandomnessTests1(out);                 // Nueva instancia con secuencia binaria
}

/** GENERARANDOMHEXA()
 * @Method generarRandomHexa()
 * Funcion que genera digitos hexadecimales aleatoriamente haciendo uso de la clase java.util.Random.
 * @param  int tam: Numero de caracteres a generar. (2048)
 * @return RandomnessTests1: Nueva instancia de la clase RandomnessTests1 con secuencia binaria cargada.
***/
public static RandomnessTests1 generarRandomHexa(int size) throws Exception {

    // Crea archivo HexaRandom.txt con salida de generador.
         File ArchivoHexa = new File("HexaRandom.txt");
         FileWriter RandomHexa = new FileWriter(ArchivoHexa);

         StringBuffer out = new StringBuffer("");
         StringBuffer out2 = new StringBuffer("");
         String cad = "";
         Random a = new Random();

         while (out.length() < size){
                 out.append(Integer.toHexString(a.nextInt()).toUpperCase());
                 RandomHexa.write(Integer.toHexString(a.nextInt()).toUpperCase());
         }

         // Convierte de Hexa a Bin para procesar prueba estadística
         if (out.length() > size){
                 out.delete(size, out.length());
         }
                 out2.append(utilidades.HextoBin(out));

         RandomHexa.close();
         return new RandomnessTests1(out2);        // Crea instancia con secuencia generada por RNG de
Java

}
```

```
/** SETALFA()
 * @Method setAlfa()
 * Funcion que establece valor de alfa (nivel de significancia)
 * @param  double a: Valor de alfa (por defecto: 0.01)
***/
public void setAlfa(double a){
        alfa = a; // por defecto 0.01
}

/** SETE()
  * @Method setE()
  * Funcion que establece la secuencia.
  * @param  StringBuffer a: Secuencia.
 ***/
public void setE(StringBuffer a){
        e = a;
}

/** SIZE()
  * @Method size()
  * Funcion que retorna el tamaño de la secuencia de bits.
  * @return  int e: Tamaño de la secuencia en bits.
 ***/
public int size(){
        return e.length();
}

/** ********************************************************
                  PRUEBAS ESTADÍSTICAS DE ALEATORIEDAD
********************************************************* **/

/** FREQUENCY()
 * @Method frequency()
 * Funcion que implementa la prueba de Frequency (Monobit) Test
 * El foco de esta prueba es la proporción de ceros y unos en la secuencia.
  * @return p-value.
***/
public double frequency() throws NoSuchMethodException {

        double p_value = -1;      // Probabilidad
        double s;
        int sum = 0;                      // Acumulador

        // Recorre la secuencia sumando 1 cuando se encuentra un 1 y restando un 1 cuando se encuentra
un 0
        for (int i = 0; i < e.length(); i++){
          char c = e.charAt(i);
          if (c =='1')
                sum++;   // Contador de 1's
          else
                sum--;   // Contador de 0's
        }

        s = Math.abs(sum)/Math.sqrt(e.length());           // Estadístico Sobs
        p_value = SpecialFunction.erfc(s/Math.sqrt(2));    // Probabilidad

        return p_value;
  }

 /** BLOCKFREQUENCY()
  * @Method blockFrequency()
  * Funcion que implementa la prueba de Frequency within a Block Test
  * El foco de esta prueba es la proporción de ceros y unos en bloques de M-bits.
  * @return p-value.
 ***/
public double blockFrequency(int M) throws NoSuchMethodException, Exception {

        double p_value = -1;
        double x = 0;

        int N = (int)(e.length()/M);      // Nro. de Bloques
        double pi[] = new double[N];      // Celda x Bloque

        // Calcula la proporcion de 1's en cada bloque
        for(int i = 0; i < N; i++){
```

```
          pi[i] = calcular_pi(M, i);
        }

        // Calcula el Estadistico X2(obs)
        for(int i = 0; i < N; i++){
          x = x + Math.pow((pi[i]-(1/2.0)), 2);
        }
        x = 4 * M * x;                                                                    //
Estadístico X2(obs)

        p_value = SpecialFunction.igamc((double)N/2,x/2);  // Probabilidad

        return p_value;
  }


/** APROXIMATEENTROPY()
 * @approximateEntropy()
 * Funcion que implementa la prueba de Approximate Entropy Test
 * El foco de esta prueba es la frecuencia de todos los patrones de “m” bits que se sobreponen en la
secuencia.
 * @param m: Longitud del bloque en bits.
 * @return p-value.
***/
public double approximateEntropy(int m) throws NoSuchMethodException {

        double p_value = -1;

        //ApEn(m)
        double q1 = calcular_fi(m);
        double q2 = calcular_fi(m+1);

        // Calculo del Estadistico X2(obs)
        double x = 2*e.length()*(Math.log(2)-(q1-q2));

        // Calculo de la Probabilidad
        p_value = SpecialFunction.igamc((Math.pow(2,m-1)),(x/2));
        return p_value;
}


/** CUMULATIVESUMS()
 * @cumulativeSums()
 * Funcion que implementa la prueba de Cumulative Sums (Cusums) Test
 * El propósito de esta prueba es el de determinar si la suma acumulativa de secuencias parciales
existentes en la secuencia analizada es muy grande o muy pequeña con respecto a la esperada para una
secuencia aleatoria.
 * @param mode: 0: Foward, 1: Reverse
 * @return p-value.
***/
public double cumulativeSums(boolean mode) throws NoSuchMethodException {

        double z;
        int tam = e.length();
        int x[] = new int[tam];
        int s[] = new int[tam];

        // Inicializa el arreglo en ceros
        for (int i = 0; i < tam; i++){
          s[i] = 0;
        }

        // Conversión de 0's a -1 y de 1 a +1.
        for (int i = 0; i < tam; i++){
          char c = e.charAt(i);
          if (c =='1')
                x[i] = 1;
          else
                x[i] = -1;
        }

        // Calculo de Sumas Parciales
        if (!mode){
        // Mode = 0
         for (int i = 0 ;i < tam; i++){
                for (int j = 0 ; j <= i; j++){
```

```
                s[i] = s[i] + x[j];
              }
          }
        }else{
        // Mode = 1
          for (int i = tam-1; i >= 0; i--){
                 for (int j = tam-1; j >= i; j--){
                   s[i] = s[i] + x[j];
                 }
          }
        }

        // Calculo del estadístico Z
        z = Math.abs(s[0]);
        for (int i = 1; i < tam; i++){
          if(Math.abs(s[i])>z){
                 z = Math.abs(s[i]);
         }
        }

        // Calculo de la Probabilidad
        double index_inf = ((((tam*-1)/z) + 1)/4.0);
        double index_sup = (((tam/z*1.0) - 1)/4.0);
        int k = (int)index_inf;
        double t1 = 0, t2 = 0;
        double sum1 = 0, sum2 = 0;

        while (k <=index_sup){
           t1                                                                                                    =
SpecialFunction.normal(((4*k+1)*z)/Math.sqrt(tam));//nd.getCDF(((4*k+1)*z)/Math.sqrt(tam));
           t2          =          SpecialFunction.normal(((4*k-1)*z)/Math.sqrt(tam));//nd.getCDF(((4*k-
1)*z)/Math.sqrt(tam));
           sum1 = sum1 + (t1-t2);
           k = k+1;
        }
        index_inf = ((((tam*-1)/z) - 3)/4.0);
        k = (int)index_inf;

         while (k<=index_sup){
           t1                                                                                                    =
SpecialFunction.normal(((4*k+3)*z)/Math.sqrt(tam));//nd.getCDF(((4*k+3)*z)/Math.sqrt(tam));
           t2                                                                                                    =
SpecialFunction.normal(((4*k+1)*z)/Math.sqrt(tam));//nd.getCDF(((4*k+1)*z)/Math.sqrt(tam));
           sum2 = sum2 + (t1-t2);
           k = k+1;
        }

        return (1-(sum1-sum2));   // Probabilidad
}


  /** ****************************************************
                                FUNCIONES ADICIONALES
  ***************************************************** **/

  /** CALCULAR_PI()
   * @Method calcular_pi()
   * Funcion que determina la proporcion de 1s en cada bloque.
   * @param  int M: Tamano de cada bloque, int index: Nro. de bloque
   * @return Proporcion de 1s en el bloque
  ***/
 private double calcular_pi(int M, int index) throws Exception{
        double out = 0;
        for (int j = 0; j < M; j++){
          out = out + utilidades.CharToInt(e.charAt((index * M)+ j));
        }
        return (out/M);  // Proporcion de 1s en cada bloque
 }

 /** CALCULAR_R()
   * @Method calcular_r()
   * Funcion que calcula r(k).
   * @param  int index: Posicion en la secuencia.
   * @return 0 o 1
  ***/
 private int calcular_r(int index){
```

```
        if (e.charAt(index) == e.charAt(index+1))
          return 0;               // r(k) = 0, si e(k) = e(k+1)
        else
          return 1;               // r(k) = 1, si e(k) <> e(k+1)
 }

 /** CALCULAR_FI()
   * @Method calcular_fi()
   * Funcion que ApEn(m)
   * @param  int m: Longitud en bits de cada bloque.
   * @return 0 o 1
  ***/
 private double calcular_fi(int m){

        int tam_e = e.length();                          // Longitud de la secuencia
        e.append(e.substring(0, m-1));            // Extiende la secuencia

        int tam = (int) Math.pow(2, m), conv;
        StringBuffer cad[] = new StringBuffer[tam];
        for (int i = 0 ;i < tam; i++)
          cad[i] = new StringBuffer("");

        conv = tam/2;
        while (conv > 0){
          for (int j = 0; j < tam/(2*conv); j++){
                 for (int i = 0; i < conv; i++){
                   cad [i+(conv*j*2)].append("0");
                   cad [(i+conv)+(conv*j*2)].append("1");
                 }
        }
           conv=conv/2;
        }

        double c[] = new double[tam];
        for (int j = 0; j < tam; j++){
          c[j] = 0;
        }

        int k = 0;
        while (k < tam_e){
          String temp = e.substring(k, k + m);
          boolean sw = true;
          for (int j = 0; j < tam && sw; j++){
                 if (temp.equals(cad[j].toString()) ){
                           c[j]++;
                           sw = false;
                 }
          }
          k++;
        }

        double res = 0.0;
        for (int i = 0; i < tam; i++){
          double t;
          c[i] = c[i]/tam_e;
          if (c[i] > 0)
           t = (c[i]*Math.log(c[i]));
          else
                 t = 0;
          res = res + t;
        }

        e.delete(e.length()- m + 1, e.length());
        return res;
 }


/** ****************************************************
                                MODULO PRINCIPAL Y CONCLUSIONES
**************************************************** **/

/** CONCLUSION **/
public static String respuesta (double p_value){
        // TRUE: La secuencia es aleatoria
        // FALSE: La secuencia no es aleatoria
        String conclusion = null;
```

```
        if (alfa < p_value){
                conclusion = "\n -> La secuencia es aleatoria.";
        } else {
                conclusion = "\n -> La secuencia NO es aleatoria.";
        }

        return conclusion;
}

/** MODULO PRINCIPAL **/
public static void main(String[] args) {

        // Parametros de Entrada
        BufferedReader buf = new BufferedReader(new InputStreamReader(System.in));
        String opc = null;
        int opcion = 0;

        // Parámetros de entrada para las pruebas
         int setM = 10;                  // Nro. de bloques en Prueba de Block Frequency
     boolean setMode = false;     // Mode para Cusums Test
     int setRNGsize = 2048;              // Nro. de valores Hexa generados por RNG de Java

         // Diferentes tipos de archivos sobre los cuales efectuar las pruebas

System.out.println("**********************************************************************************
***");
         System.out.println("*\t\t\tPRUEBAS DE ALEATORIEDAD                                                *");

System.out.println("**********************************************************************************
***");
         System.out.println("*\t1.   Correr   pruebas   de   aleatoriedad   sobre   archivo   binario
*");
         System.out.println("*\t2. Correr pruebas sobre de aleatoriedad sobre archivo generado con RNG
de Java *");
         System.out.println("*\t3.                                                                    Salir
*");

System.out.println("**********************************************************************************
***");
         System.out.print("\tOpcion: ");

         System.out.flush( );
          try
           {
                 opc = buf.readLine();

                 try{
                         opcion = Integer.parseInt(opc);
                 }catch(NumberFormatException Nfe){
                         System.out.println("Error, formato invalido: " + Nfe);
                 }

                 System.out.println("\n");
                 switch (opcion){
                         case 1:
                                  String sFile = new String("data.rsa.txt"); // NOMBRE DEL ARCHIVO:
Puede ser tambien: "data.e", "data.pi", "data.sqrt2", "data.sqrt3", "data.rsa.txt", "data.hotp.txt"
                                  File mFile = new File(sFile);

                                  System.out.println("Archivo Binario: " + sFile +  " \n");

                                 try{

                                 RandomnessTests1 r1 = RandomnessTests1.fromBinTextFile(sFile);
                                 RandomnessTests2 r2 = RandomnessTests2.fromBinTextFile(sFile);
                                 RandomnessTests3 r3 = RandomnessTests3.fromBinTextFile(sFile);

                                 System.out.println("*** FREQUENCY TEST ***\t");
                                 double res = r1.frequency();
                                 System.out.print("Frequency: " + res + "  \t ");
                                 System.out.println(RandomnessTests1.respuesta(res));
                                 System.out.println("");

                                 System.out.println("*** BLOCK FREQUENCY TEST ***\t");
                                 res = r1.blockFrequency(setM);
```

```
                                System.out.print("BlockFrecuency: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** RUNS TEST ***\t");
                                res = r2.runs();
                                System.out.print("Runs: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** LONGEST RUNS OF ONES TEST ***\t");
                                res = r2.longestRunOfOnes();
                                System.out.print("Longest Run of Ones: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** LINEAR COMPLEXITY TEST ***\t");
                                res = r3.linearComplexity(setM);
                                System.out.print("Linear Complexity: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** SERIAL TEST ***\t");
                                res = r3.serial(setM);
                                //System.out.print("Serial: " + res + "  \t ");
                                //System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** NON OVERLAPPING MATCHING TEST ***\t");
                                res = r2.nonOverlappingTemplateMatching("000000001");
                                System.out.print("Non Overlapping Template Matching: " + res + "   \t
");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** OVERLAPPING MATCHING TEST ***\t");
                                res = r2.overlappingTemplateMatching("111111111");
                                System.out.print("Overlapping Template Matching: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** APPROXIMATE ENTROPY TEST ***\t");
                                System.out.println("Aplicando la prueba Approximate Entropy, puede
tardar varios minutos .....");
                                res = r1.approximateEntropy(5);
                                System.out.print("Approximate Entropy: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** CUMMULATIVE SUMS TEST ***\t");
                                System.out.println("Aplicando la prueba Cusums, puede tardar varios
minutos .....");
                                res = r1.cumulativeSums(setMode);
                                System.out.print("Cumulative Sums (Cusums): " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("***");

                                }catch (Exception a){
                                        a.printStackTrace();
                                }
                                break;

                        case 2:
                                try{

                                System.out.println("Archivo generado con RNG de Java: HexaRandom.txt
\n");
                                RandomnessTests1 r1 = RandomnessTests1.generarRandomHexa(setRNGsize);
                                sFile   =   new   String("hexarandom.txt");      //data.random.txt   o
hexarandom.txt
                                mFile = new File(sFile);
                                RandomnessTests2 r2 = RandomnessTests2.fromBinTextFile(sFile);
                                RandomnessTests3 r3 = RandomnessTests3.fromBinTextFile(sFile);
```

```
                                double res = r1.frequency();
                                System.out.print("Frequency: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                res = r1.blockFrequency(setM);
                                System.out.print("BlockFrecuency: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                res = r2.runs();
                                System.out.print("Runs: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                res = r2.longestRunOfOnes();
                                System.out.print("Longest Run of Ones: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** SERIAL TEST ***\t");
                                res = r3.serial(setM);
                                //System.out.print("Serial: " + res + "  \t ");
                                //System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("*** LINEAR COMPLEXITY TEST ***\t");
                                res = r3.linearComplexity(setM);
                                System.out.print("Linear Complexity: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                res = r2.nonOverlappingTemplateMatching("000000001");
                                System.out.print("Non Overlapping Template Matching: " + res + "   \t
");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                res = r2.overlappingTemplateMatching("111111111");
                                System.out.print("Overlapping Template Matching: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("Aplicando la prueba Approximate Entropy, puede
tardar varios minutos .....");
                                res = r1.approximateEntropy(5);
                                System.out.print("Approximate Entropy: " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("Aplicando la prueba Cusums, puede tardar varios
minutos .....");
                                res = r1.cumulativeSums(setMode);
                                System.out.print("Cumulative Sums (Cusums): " + res + "  \t ");
                                System.out.println(RandomnessTests1.respuesta(res));
                                System.out.println("");

                                System.out.println("***");

                                }catch (Exception a){
                                        a.printStackTrace();
                                }

                                break;

                        case 3: System.exit(0);break;
                                 default:System.out.println("Opcion seleccionada invalida.");break;
                }

        }
        catch( IOException e )
               {
                 System.err.println("Error:" + e);
               }
        }
 }
```

```
/*
 * Created on 26/03/2005
 *
 * CLASE RANDOMNESSTEST2 V.1.0
 * Clase donde se implementa la lectura de los archivos (binarios, hexadecimales o generados)
 * con los datos de entrada. Se almacenan estos datos en una secuencia sobre la cual
 * se aplican las pruebas estadísticas de aleatoriedad definidas por el NIST.
 *
 * Clase que implementa los métodos:
 *       -- Frequency
 *       -- Block Frequency
 *       - Runs
 *       - Longest Run of Ones
 *  -- Linear Complexity
 *  -- Serial Test
 *       -- Aproximate Entropy
 *       -- Cumulative Sums
 *       - NonOverlapping Template Matching
 *       - Overlapping Template Matching
 *
 *  Se pueden aplicar las pruebas sobre archivos binbarios. También se pueden correr las pruebas
 *  sobre un archivo hexadecimal generado por el RNG de Java.
 */

/**
 * @author Sergio Garcia - 200217365
 *
 * Universidad de los Andes
 * Proyecto de Grado - Magister en Ing. de Sistemas
 *
 * Asesor: Milton Quiroga
 */

package RandomTests;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Vector;
import util.*;

public class RandomnessTests2 {
         static StringBuffer e;

         /*** CONSTRUCTOR
          * @Method RandomnessTests2()
          * Constructor de la Clase: Asigna la secuencia llena a la nueva instancia de la clase.
          * @param  StringBuffer sec: Secuencia que contiene los datos del archivo binario.
          ***/
          public RandomnessTests2(StringBuffer sec) {
                        e = sec;         // Secuencia
        }

        /** FROMBINTEXTFILE()
           * @Method fromBinTextFile()
           * Funcion que retorna una nueva instancia de la clase RandomnessTest con la cadena de bits
cargada a partir de los datos de entrada de un archivo binario.
           * @param   String f: Archivo de texto con caracteres 0, 1 (Otros tipos de caracteres son
descartados).
           * @return RandomnessTest: Nueva instancia de la clase RandomnessTest con secuencia binaria
cargada.
          ***/
         public static RandomnessTests2 fromBinTextFile(String f) throws Exception {

                BufferedReader br = new BufferedReader(new FileReader(f)); // Buffer de entrada
                StringBuffer out = new StringBuffer("");                                  // Buffer de
salida
                String cad = "";
                // Cadena

                cad = br.readLine();                                          // Lee  la primera  línea del
archivo
                while(cad != null){
                  out.append(utilidades.toBin(cad, 1));   // Adjunta a la secuencia solo 0's y 1's
                  cad = br.readLine();                                        // Lectura linea x linea
```

```
                }
                return new RandomnessTests2(out);                // Nueva instancia con secuencia
binaria
        }

  /** SIZE()
   * @Method size()
   * Funcion que retorna el tamaño de la secuencia de bits.
   * @return  int: Tamaño de la secuencia en bits.
  ***/
  public int size() {
        return e.length();
  }

/** ******************************************************
  PRUEBAS ESTADÍSTICAS DE ALEATORIEDAD
****************************************************** **/

  /** RUNS()
   * @Method runs()
   * Funcion que implementa la prueba de RUNS Test
   * El foco de esta prueba es el número de cadenas (runs) en cada secuencia.
   * @return p-value.
  ***/
  public double runs() throws Exception {

        int n = this.size(); // Tamano de la secuencia
        int ones = 0;
        double pi = 0;
        double Xobs = 0;
        double pvalue = 0;

        // Determina proporcion de 1s en la secuencia
        double tao = 2 / Math.sqrt(n);
        int Vnobs = 0;
        for (int i = 0; i < n; i++) {
          if (e.charAt(i) == '1')
                ones += 1;
        }

        // Calculo del limite para verificar si pasa prueba de frecuencia
        pi = (double) ones / n;
        if (pi - 0.5 >= tao) {
          pvalue = 0;
        }
        else {
          for (int i = 1; i < n; i++) {
                if (e.charAt(i) == e.charAt(i - 1))
                  continue;
                else
                  Vnobs += 1;
          }

          // Calcular Estadístico V(n) y la probabilidad
          Vnobs++;
          pvalue = SpecialFunction.erfc(Math.abs(Vnobs - 2 * n * pi * (1 - pi)) /
                                                                        (2 * (Math.sqrt(2 * n)) * pi
* (1 - pi)));

        }
        return pvalue;
  }


 /** LONGEST_RUN_OF_ONES()
   * @Method longestrunofones()
   * Funcion que implementa la prueba de LONGEST RUN OF ONES Test
   * El foco de esta prueba es el número de cadenas (runs) en cada subsecuencia.
   * @return p-value.
  ***/
  public double longestRunOfOnes() throws Exception {

        int n = this.size(); // Tamano de la secuencia

        int[] v = new int[7];
        int[] limV = new int[7];
```

```
double[] pi = new double[7];
int M = 0;
int K = 0;
int N = 0;
int runOnes = 1;
int maxRunOnes = 0;
double Xobs = 0;
int x = 0;
double pvalue;

if (n >= 128 && n < 6272)
  M = 8;
else if (n >= 6272 && n < 750000)
  M = 128;
else if (n >= 750000)
  M = 10000;
else {
  System.out.println("Error: La secuencia debe contener al menos 128 dígitos.");
  return -1;
}

// Valores parametrizados
if (M == 8) {    // M = 8
  K = 3;
  limV[0] = 1;
  limV[1] = 2;
  limV[2] = 3;
  limV[3] = 4;
  pi[0] = 0.2148;
  pi[1] = 0.3672;
  pi[2] = 0.2305;
  pi[3] = 0.1875;
}
else if (M == 128) {      // M = 128
  K = 5;
  limV[0] = 4;
  limV[1] = 5;
  limV[2] = 6;
  limV[3] = 7;
  limV[4] = 8;
  limV[5] = 9;
  pi[0] = 0.1174;
  pi[1] = 0.2430;
  pi[2] = 0.2493;
  pi[3] = 0.1752;
  pi[4] = 0.1027;
  pi[5] = 0.1124;
}
else {   // M = 10000
  K = 6;
  limV[0] = 10;
  limV[1] = 11;
  limV[2] = 12;
  limV[3] = 13;
  limV[4] = 14;
  limV[5] = 15;
  limV[6] = 16;
  pi[0] = 0.0882;
  pi[1] = 0.2092;
  pi[2] = 0.2483;
  pi[3] = 0.1933;
  pi[4] = 0.1208;
  pi[5] = 0.0675;
  pi[6] = 0.0727;
}

N = (int) n / M;

for (int i = 1; i < (n - (n % M)); i++) {
  if (e.charAt(i) == '1' && e.charAt(i) == e.charAt(i - 1))
        runOnes += 1;
  else {
        if (maxRunOnes < runOnes)
          maxRunOnes = runOnes;
        runOnes = 1;
  }
```

```
          if ( (i + 1) % M == 0) {
                i++;
                if (maxRunOnes < runOnes)
                  maxRunOnes = runOnes;

                runOnes = 1;
                for (int j = 0; j <= 7; j++) {
                  if (j == K) {
                        v[j] += 1;
                        maxRunOnes = 0;
                        break;
                  }
                  else if (maxRunOnes <= limV[j]) {
                        v[j] += 1;
                        maxRunOnes = 0;
                        break;
                  }
                }
          }
        }

        // Calculo del estadistico
        for (int i = 0; i <= K; i++) {
          Xobs += Math.pow(v[i] - N * pi[i], 2) / (N * pi[i]);
        }

        // Calculo de la Probabilidad
        pvalue = SpecialFunction.igamc( ( (double) K / 2), ( (double) Xobs / 2));

        return pvalue;
 }

/** NON_OVERLAPPING_TEMPLATE_MATCHING()
  * @Method nonoverlappingtemplatematching()
  * Funcion que implementa la prueba de NON OVERLAPPING TEMPLATE MATCHING Test
  * El foco de esta prueba es el número de coincidencias con patrones (plantillas) determinados.
  * @return p-value.
 ***/
 public double nonOverlappingTemplateMatching(String len) throws Exception {
        Vector template = new Vector();
        int m = len.length();

        /***  Para utilizar las plantillas predeterminadas, quitar comentarios a la plantilla a buscar
           if (m == 2) {
                  template.add("01");
                  template.add("10");
                }
                else if (m == 3) {
                  template.add("001");
                  template.add("011");
                  template.add("100");
                  template.add("110");
                }
                else if (m == 4) {
                  template.add("0001");
                  template.add("0011");
                  template.add("0111");
                  template.add("1000");
                  template.add("1100");
                  template.add("1110");
                }
                else if (m == 5) {
                  template.add("00001");
                  template.add("00011");
                  template.add("00101");
                  template.add("01011");
                  template.add("00111");
                  template.add("01111");
                  template.add("11100");
                  template.add("11010");
                  template.add("10100");
                  template.add("11000");
                  template.add("10000");
                  template.add("11110");
                }
                else if (m == 6) {
```

```
            template.add("000001");
            template.add("000011");
            template.add("000101");
            template.add("000111");
            template.add("001011");
            template.add("001101");
            template.add("001111");
            template.add("010011");
            template.add("010111");
            template.add("011111");
            template.add("100000");
            template.add("101000");
            template.add("101100");
            template.add("110000");
            template.add("110010");
            template.add("110100");
            template.add("111000");
            template.add("111010");
            template.add("111100");
            template.add("111110");
          }
          else if (m == 7) {
            template.add("0000001");
            template.add("0000011");
            template.add("0000101");
            template.add("0000111");
            template.add("0001001");
            template.add("0001011");
            template.add("0001101");
            template.add("0001111");
            template.add("0010011");
            template.add("0010101");
            template.add("0010111");
            template.add("0011011");
            template.add("0011101");
            template.add("0011111");
            template.add("0100011");
            template.add("0100111");
            template.add("0101011");
            template.add("0101111");
            template.add("0110111");
            template.add("0111111");
            template.add("1000000");
            template.add("1001000");
            template.add("1010000");
            template.add("1010100");
            template.add("1011000");
            template.add("1011100");
            template.add("1100000");
            template.add("1100010");
            template.add("1100100");
            template.add("1101000");
            template.add("1101010");
            template.add("1101100");
            template.add("1110000");
            template.add("1110010");
            template.add("1110100");
            template.add("1110110");
            template.add("1111000");
            template.add("1111010");
            template.add("1111100");
            template.add("1111110");
          }
*/

int n = this.size();
int N = 8;
int M = (int) n / N;
int match = 0;

/***  Quitar comentarios para usar plantilla predeterminada
       Vector tmpTemplate = (Vector) template.clone();
       Iterator iTmplt = tmpTemplate.iterator();
*/

String copy = e.toString();
```

```
        int offset = 0;
        int index = 0;
        int lastPos = 0;
        int[] w = new int[N];
        double pvalue = 0;

        for (int i = 0; i < N; i++)
          w[i] = 0;

        /***  Quitar comentarios para usar plantilla predeterminada
                  for (int i = 0; i < (n-(n%M)); i++) {
                        tmplt.indexOf()
                        while (iTmplt.hasNext()) {
                                tmplt = (String) iTmplt.next();
                                copy = sec.toString();
        */

        for (int i = 0; i < N; i++)
          w[i] = 0;

        lastPos = 0;
        index = 0;

        while (index != -1) {
          index = copy.indexOf(len);
          if (index != -1) {
                 if ( ( (index + lastPos) % M) <= M - m) {
                   int pos = (int) ( (index + lastPos) / M);
                   w[pos] += 1;
                   copy = copy.substring(index + m);
                   lastPos += index + m;
                 }
                 else {
                   copy = copy.substring(index + (M - (index % M)));
                   lastPos += index + (M - (index % M));
                 }
          }
        }

        double media = (M - m + 1) / Math.pow(2, m);
        double varianza = M * (1 / Math.pow(2, m) - (2 * m - 1) / (Math.pow(2, 2 * m)));
        double Xobs = 0;

        for (int j = 0; j < N; j++) {
          Xobs += Math.pow(w[j] - media, 2) / varianza;
        }

        pvalue = SpecialFunction.igamc( (double) N / 2, Xobs / 2);

        //*** Quitar en caso de usar las plantillas predeterminadas }
        return pvalue;
 }

/** OVERLAPPING_TEMPLATE_MATCHING()
  * @Method nonoverlappingtemplatematching()
  * Funcion que implementa la prueba de OVERLAPPING TEMPLATE MATCHING Test
  * El foco de esta prueba es el número de coincidencias con patrones (plantillas) determinados.
  * @return p-value.
 ***/
 public double overlappingTemplateMatching(String len) throws Exception {
        int m = len.length();
        Vector template = new Vector();

        /***  Para utilizar las plantillas pre-establecidas, quitar comentarios a la plantilla a buscar
         if (m == 2) {
          template.add("11");
          template.add("10");
          template.add("01");
         }
         else if (m == 3) {
          template.add("001");
          template.add("011");
          template.add("100");
          template.add("110");
         }
         else if (m == 4) {
```

```
 template.add("0001");
 template.add("0011");
 template.add("0111");
 template.add("1000");
 template.add("1100");
 template.add("1110");
}
else if (m == 5) {
 template.add("00001");
 template.add("00011");
 template.add("00101");
 template.add("01011");
 template.add("00111");
 template.add("01111");
 template.add("11100");
 template.add("11010");
 template.add("10100");
 template.add("11000");
 template.add("10000");
 template.add("11110");
}
else if (m == 6) {
 template.add("000001");
 template.add("000011");
 template.add("000101");
 template.add("000111");
 template.add("001011");
 template.add("001101");
 template.add("001111");
 template.add("010011");
 template.add("010111");
 template.add("011111");
 template.add("100000");
 template.add("101000");
 template.add("101100");
 template.add("110000");
 template.add("110010");
 template.add("110100");
 template.add("111000");
 template.add("111010");
 template.add("111100");
 template.add("111110");
}
else if (m == 7) {
 template.add("0000001");
 template.add("0000011");
 template.add("0000101");
 template.add("0000111");
 template.add("0001001");
 template.add("0001011");
 template.add("0001101");
 template.add("0001111");
 template.add("0010011");
 template.add("0010101");
 template.add("0010111");
 template.add("0011011");
 template.add("0011101");
 template.add("0011111");
 template.add("0100011");
 template.add("0100111");
 template.add("0101011");
 template.add("0101111");
 template.add("0110111");
 template.add("0111111");
 template.add("1000000");
 template.add("1001000");
 template.add("1010000");
 template.add("1010100");
 template.add("1011000");
 template.add("1011100");
 template.add("1100000");
 template.add("1100010");
 template.add("1100100");
 template.add("1101000");
 template.add("1101100");
 template.add("1101100");
```

```
          template.add("1110000");
          template.add("1110010");
          template.add("1110100");
          template.add("1110110");
          template.add("1111000");
          template.add("1111010");
          template.add("1111100");
          template.add("1111110");
         }
         */

        int n = this.size();
        //if (n < 1000000)
        //    System.out.println("Error:  \n  La  prueba  requiere  de  al  menos  10^6  digitos  en  la
secuencia.");
        //else
        //  n = 1000000;

        int lenM = 1032;
        int N = (int) n / lenM;
        double landa = (lenM - m + 1) / Math.pow(2, m);
        double Xobs = 0;
        int index = 0;
        int lastPos = 0;
        int[] w = new int[N + 1];
        int[] v = new int[6];
        double[] pi = new double[6];
        double pvalue = 0;

        String copy;

        /***  Quitar comentarios para utilizar plantillas predeterminadas
          Iterator iTmplt = template.iterator();
         while (iTmplt.hasNext()) {
                 tmplt = (String) iTmplt.next();
         */

        for (int i = 0; i < N; i++)
          w[i] = 0;
        copy = e.substring(0, n - 1);
        lastPos = 0;
        index = 0;
        while (index != -1) {
          index = copy.indexOf(len);
          if (index != -1) {
                 if ( ( (index + lastPos) % lenM) <= lenM - m) {
                   int pos = (int) ( (index + lastPos) / lenM);
                   w[pos] += 1;
                   copy = copy.substring(index + 1);
                   lastPos += index + 1;
                 }
                 else {
                   if ( (index + (lenM - ( (index + lastPos) % lenM))) < n) {
                          copy = copy.substring(index + (lenM - ( (index + lastPos) % lenM)));
                          lastPos += index + (lenM - ( (index + lastPos) % lenM));
                   }
                 }
          }
        }

        for (int i = 0; i < N; i++) {
          if (w[i] >= 5)
                 v[5] += 1;
          else
                 v[w[i]] += 1;
        }

        double comb = 1;
        double facSuma = 0;
        int K = 5;
        pi[0] = 0.367879;
        pi[1] = 0.183940;
        pi[2] = 0.137955;
        pi[3] = 0.099634;
        pi[4] = 0.069935;
        pi[5] = 0.140657;
```

```
        for (int j = 0; j <= K; j++) {
          if (j == 0)
                facSuma = 1;
          else
                facSuma = 0;
          for (int l = 1; l <= j; l++) {
                comb = SpecialFunction.fac(j - 1) /
                        (SpecialFunction.fac(l - 1) * SpecialFunction.fac(j - l));
                facSuma += (comb * Math.pow(landa / 2, l) / SpecialFunction.fac(l));
          }

          // Calculo del estadístico
          /***  Quitar comentarios para utilizar plantillas predeterminadas
          pi[j] = (Math.pow(Math.E, -landa/2) / Math.pow(2, j)) * facSuma; */
          Xobs += Math.pow(v[j] - N * pi[j], 2) / (N * pi[j]);
        }

        // Calculo de la probabilidad
        pvalue = SpecialFunction.igamc( (double) K / 2, Xobs / 2);
        /***  Quitar comentarios para utilizar plantillas predeterminadas
        } */

        return pvalue;
  }
}

/*
 * Created on 26/03/2005
 *
 * CLASE RANDOMNESSTEST3 V.1.0
 * Clase donde se implementa la lectura de los archivos (binarios, hexadecimales o generados)
 * con los datos de entrada. Se almacenan estos datos en una secuencia sobre la cual
 * se aplican las pruebas estadísticas de aleatoriedad definidas por el NIST.
 *
 * Clase que implementa los métodos:
 *       -- Frequency
 *       -- Block Frequency
 *       -- Runs
 *       -- Longest Run of Ones
 *  - Linear Complexity
 *  - Serial Test
 *       -- Aproximate Entropy
 *       -- Cumulative Sums
 *       -- NonOverlapping Template Matching
 *       -- Overlapping Template Matching
 *
 *  Se pueden aplicar las pruebas sobre archivos binbarios. También se pueden correr las pruebas
 *  sobre un archivo hexadecimal generado por el RNG de Java.
 */

/**
 * @author Sergio Garcia - 200217365
 *
 * Universidad de los Andes
 * Proyecto de Grado - Magister en Ing. de Sistemas
 *
 * Asesor: Milton Quiroga
 */

package RandomTests;

import java.io.BufferedReader;
import java.io.FileReader;
import util.SpecialFunction;
import util.utilidades;

public class RandomnessTests3 {
         static StringBuffer e;

        /*** CONSTRUCTOR
          * @Method RandomnessTests3()
          * Constructor de la Clase: Asigna la secuencia llena a la nueva instancia de la clase.
          * @param  StringBuffer sec: Secuencia que contiene los datos del archivo binario.
        ***/
        public RandomnessTests3(StringBuffer sec) {
```

```
                        e = sec;         // Secuencia
        }

        /** FROMBINTEXTFILE()
         * @Method fromBinTextFile()
         * Funcion que retorna una nueva instancia de la clase RandomnessTest con la cadena de bits
cargada a partir de los datos de entrada de un archivo binario.
         * @param   String f: Archivo de texto con caracteres 0, 1 (Otros tipos de caracteres son
descartados).
         * @return RandomnessTest: Nueva instancia de la clase RandomnessTest con secuencia binaria
cargada.
        ***/
        public static RandomnessTests3 fromBinTextFile(String f) throws Exception {

                BufferedReader br = new BufferedReader(new FileReader(f)); // Buffer de entrada
                StringBuffer out = new StringBuffer("");                                    // Buffer de
salida
                String cad = "";
                // Cadena

                cad = br.readLine();                                          // Lee  la primera  línea del
archivo
                while(cad != null){
                  out.append(utilidades.toBin(cad, 1));   // Adjunta a la secuencia solo 0's y 1's
                  cad = br.readLine();                                        // Lectura linea x linea
                }
                return new RandomnessTests3(out);         // Nueva instancia con secuencia binaria
         }

        /** SIZE()
         * @Method size()
         * Funcion que retorna el tamaño de la secuencia de bits.
         * @return  int: Tamaño de la secuencia en bits.
        ***/
        public int size() {
                return e.length();
        }

        /** ******************************************************
          PRUEBAS ESTADÍSTICAS DE ALEATORIEDAD
        ****************************************************** **/

        /** LINEAR COMPLEXITY()
         * @Method linearComplexity()
         * Funcion que implementa la prueba de LINEAR COMPLEXITY Test
         * El foco de esta prueba es el tamaño del registro de retroalimentacion en cada secuencia.
         * @return p-value.
        ***/
        public double linearComplexity(int M) throws Exception {

                // Particionar la secuencia
                int n = this.size(); // Tamano de la secuencia
                int n1 = (int)(n%M);       // Se descartan los bits sobrantes
                n = n-n1;
                int N = (int)(n/M);        // Nro. de Bloques
                int i, ii, j, d;  // Contadores
                int L, m, N_, state, parity, sign;

                int K = 6; // Grados de libertad
                char[] assignment = new char[35];

                double[] nu = new double[7]; //Miu
                double[]  pi = {0.01047,0.03125,0.12500,0.50000,0.25000,0.06250,0.020833};
                double mean = 0;
                double T_ = 0;
                double Xobs = 0;
                double pvalue = 0;

                // Workarrays
                int[] B_ = new int[M];
                int[] C = new int[M];
                int[] T = new int[M];
                int[] P = new int[M];

                // Limpia Miu
                for(i = 0; i < K+1; i++ ){
```

```
                nu[i] = 0;
        }

        // Carga la secuencia
        int b = 0;
int[] cad = new int[n]; // Cadena en formato de arreglo

while(b < n){
                if (e.charAt(b) =='0') {
                        cad[b]=0;
                } else {
                        if (e.charAt(b) =='1'){
                                cad[b]=1;
                        }
                }
        b++;
        }

        for(ii=0; ii<N; ii++ ) {

                // Limpia los Arreglos
                for(i=0; i<M; i++ ) {
                        B_[i] = 0;
                        C[i] = 0;
                        T[i] = 0;
                        P[i] = 0;
                }

                // Inicia variables
                L = 0;
                m = -1;
                d = 0;
                C[0] = 1;
                B_[0] = 1;

                // Determinar la complejidad lineal
                N_ = 0;
                while( N_ < M ) {
                        d = (int)cad[(M+N_)]; //*ii
                        for( i=1; i<=L; i++ )
                                d += (int)C[i]*(int)cad[ii*M+N_-i];
                        d = d%2;
                        if ( d == 1 ) {
                                for( i=0; i<M; i++ ) {
                                        T[i] = C[i];
                                        P[i] = 0;
                                }
                                for(j=0; j<M; j++ )
                                        if ( B_[j] == 1 )
                                                P[j+N_-m] = 1;
                                for( i=0; i<M; i++ )
                                        C[i] = (C[i] + P[i])%2;
                                if ( L <= N_/2 ) {
                                        L = N_ + 1 - L;
                                        m = N_;
                                        for( i=0; i<M; i++ )
                                                B_[i] = T[i];
                                }
                        }
                        N_++;
                }

                // Verifica la paridad
                if ( (parity = (M+1)%2) == 0 )
                        sign = -1;
                else
                        sign = 1;
                mean = M/2. + (9.+sign)/36. - 1./Math.pow(2,M) * (M/3. + 2./9.);
                if ( (parity = M%2) == 0 )
                        sign = 1;
                else
                        sign = -1;
                T_ = sign * (L - mean) + 2./9.;

                // Almacena los valores de T
                if ( T_ <= -2.5 )
```

```
                        nu[0]++;
                else if ( T_ > -2.5 && T_ <= -1.5 )
                        nu[1]++;
                else if ( T_ > -1.5 && T_ <= -0.5 )
                        nu[2]++;
                else if ( T_ > -0.5 && T_ <= 0.5 )
                        nu[3]++;
                else if ( T_ > 0.5 && T_ <= 1.5 )
                        nu[4]++;
                else if ( T_ > 1.5 && T_ <= 2.5 )
                        nu[5]++;
                else
                        nu[6]++;
        }


        //for(i = 0; i < K+1; i++ )
        //      fprintf(stats[TESTS_LINEAR_COMPLEXITY], "%4d ",(int)nu[i]);

        // Calculo del estadistico
        for(i = 0; i < K+1; i++ )
                Xobs += Math.pow(nu[i]-N*pi[i],2)/(N*pi[i]);

        pvalue = SpecialFunction.igamc(K/2.0,Xobs/2.0);

        return pvalue;
}

/** SERIAL()
 * @Method serial()
 * Funcion que implementa la prueba de SERIAL Test
 * El foco de esta prueba es la frecuencia de los
 * patrones de m-bits que se traslapan en la secuencia.
 * @return p-value.
***/
public double serial(int M) throws Exception {

        // Particionar la secuencia
        int n = this.size(); // Tamano de la secuencia
        int N = (int)(n/M);       // Nro. de Bloques

        // Declaracion de variables
        int state;
        char[] assignment = new char[7];
        double p_value1, p_value2, psim0, psim1, psim2, del1, del2;
        double p_value = 0;

        psim0 = psi2(M, n);
        psim1 = psi2(M-1, n);
        psim2 = psi2(M-2, n);
        del1 = psim0 - psim1;
        del2 = psim0 - 2.0*psim1 + psim2;
        p_value1 = SpecialFunction.igamc((Math.pow(2,M-1)/2),(del1/2.0));
        p_value2 = SpecialFunction.igamc((Math.pow(2,M-2)/2),(del2/2.0));

        // Conclusion 1
        if ( p_value1 < RandomnessTests1.alfa ) {
                System.out.print("Serial: " + p_value1 + "  \t ");
                System.out.println("\n -> La secuencia NO es aleatoria.");
                state = 0;
        }
        else {
                System.out.print("Serial: " + p_value1 + "  \t ");
                System.out.println("\n -> La secuencia es aleatoria.");
                state = 1;
        }

        // Conclusion 2
        if ( p_value2 < RandomnessTests1.alfa ) {
                System.out.print("Serial: " + p_value2 + "  \t ");
                System.out.println("\n -> La secuencia NO es aleatoria.");
                state = 0;
        }
        else {
                System.out.print("Serial: " + p_value2 + "  \t ");
                System.out.println("\n -> La secuencia es aleatoria.");
```

```
                        state = 1;
                }

                // Retorna un valor que no se usa ya que en el mismo método se emite la conclusion
                return 0;
        }


  /** ****************************************************
                                FUNCIONES ADICIONALES
  ***************************************************** **/

        /** CALCULAR_PSI2()
         * @Method psi2()
         * Funcion que calcula la frecuencia para el test serial
         * @param int M: Longitud en bits de cada bloque.
         * @param int n: Longitud de la cadena de bits.
         * @return La frecuencia o 0
        ***/
        private double psi2(int M, int n) throws Exception{

                // Declaracion de variables
                int i, j, k, powLen;
                double sum, numOfBlocks;

                // Valida longitud de cada bloque
                if ( (M == 0) || (M == -1) )
                        return 0;

                numOfBlocks = n;
                powLen = (int)Math.pow(2,M+1)-1;

        int b = 0;
        int LenCad = this.size();
        int[] P = new int[powLen];
        int[] cad = new int[LenCad]; // Cadena en formato de arreglo

                while(b < LenCad){
                        if (e.charAt(b) =='0') {
                                cad[b]=0;
                        } else {
                                if (e.charAt(b) =='1'){
                                        cad[b]=1;
                                }
                        }
                b++;
                }

                for(i = 1; i < powLen-1; i++ )
                        P[i] = 0;          // Inicializa
                for( i=0; i < numOfBlocks; i++ ) {        // Calculo de la frecuencia
                        k = 1;
                        for(j = 0; j < M; j++ ) {
                                if (cad[(i+j)%n] == 0 )
                                        k *= 2;
                                else if (cad[(i+j)%n] == 1 )
                                        k = 2*k+1;
                        }
                        P[k-1]++;
                }

                // Retorna la frecuencia
                sum = 0;
                for( i=(int)Math.pow(2,M)-1; i<(int)Math.pow(2,M+1)-1; i++ )
                        sum += Math.pow(P[i],2);
                sum = (sum * Math.pow(2,M)/(double)n) - (double)n;

                for(i = 1; i < powLen-1; i++ )
                        P[i] = 0;          // Limpia

                return sum;
        }
}

/*
 * Created on 26/03/2005
```

```
 *
 * CLASE UTILIDADES V.1.0
 * Clase donde se implementan diferentes tipos de funciones comunes que son de utilidad
 * para la implementación de los métodos estadísticos.
 */

/**
 * @author Sergio Garcia - 200217365
 *
 * Universidad de los Andes
 * Proyecto de Grado - Magister en Ing. de Sistemas
 *
 * Asesor: Milton Quiroga
 */

package util;

public class utilidades {

 /** HEXTOBINARY()
  * @Method HextoBinary()
  * Funcion que convierte de hexa a binario.
  * @param  Char a: Caracter a convertir.
  * @return String: Cadena en binario.
 ***/
 public static String HextoBinary(char a) throws Exception{

   String out = "";      // Cadena de salida (Binaria)
   switch (a){
    case '0':
      out = "0000";
      break;
    case '1':
      out = "0001";
      break;
    case '2':
      out = "0010";
      break;
    case '3':
      out = "0011";
      break;
    case '4':
          out = "0100";
      break;
    case '5':
          out = "0101";
      break;
    case '6':
          out = "0110";
      break;
    case '7':
          out = "0111";
      break;
    case '8':
          out = "1000";
      break;
    case '9':
         out = "1001";
      break;
    case 'A':
         out = "1010";
      break;
    case 'B':
        out = "1011";
      break;
    case 'C':
        out = "1100";
      break;
    case 'D':
        out = "1101";
      break;
    case 'E':
        out = "1110";
      break;
    case 'F':
        out = "1111";
```

```
      break;
    default:
      System.out.println("Error: Caracter no válido - " + a);
     break;
   }
    return out;
 }

 /** CHARTOINT()
   * @Method CharToInt()
   * Funcion que convierte de caracter a entero.
   * @param  Char c: Caracter a convertir.
   * @return Int: Equivalente en entero.
  ***/
 public static int CharToInt(char c) throws Exception{
   switch (c){
     case '0':
       return 0;
     case '1':
       return 1;
     case '2':
       return 2;
     case '3':
       return 3;
     case '4':
       return 4;
     case '5':
       return 5;
     case '6':
       return 6;
     case '7':
       return 7;
     case '8':
       return 8;
     case '9':
       return 9;
     case 'A':
       return 10;
     case 'B':
       return 11;
     case 'C':
       return 12;
     case 'D':
       return 13;
     case 'E':
       return 14;
     case 'F':
       return 15;
     default :
       throw new Exception("Error: Caracter Invalido - " + c);
   }
 }

 /** TOBIN()
  * @Method toBin()
  * Funcion que convierte una cadena a un binario (descarta caracteres invalidos (<> de 0 o 1)
  * @param  String cad: Cadena a convertir.
  * @return StringBuffer out: Cadena en binario.
 ***/
public static StringBuffer toBin(String cad) throws Exception{
      StringBuffer out = new StringBuffer("");
      int b = 0;
          while(b < cad.length()){
            if (cad.charAt(b) =='0') {
                 out.append("0");
            } else {
                 if (cad.charAt(b) =='1') {
                         out.append("1");
                 } else {
                         if (cad.charAt(b) =='2') {
                         out.append("010");
                         } else {
                                         if (cad.charAt(b) =='3') {
                                                 out.append("011");
                                         } else {
                                                 if (cad.charAt(b) =='4') {
```

```
                                        out.append("100");
                                } else {
                                        if (cad.charAt(b) =='5') {
                                                out.append("101");
                                        } else {
                                                if (cad.charAt(b) =='6') {
                                                out.append("110");
                                                } else {
                                                        if (cad.charAt(b) =='7') {
                                                        out.append("00111");
                                                } else {
                                                        if (cad.charAt(b) =='8') {
                                                        out.append("1000");
                                                        } else {
                                                                if (cad.charAt(b) =='9') {
                                                                out.append("1001");
                                                }
                                                }
                                        }
                                        }
                                }
                                }
                        }
                }
        }
           b++;
         }
         return out;
}

/** TOBIN()
 * @Method toBin()
 * Funcion que convierte una cadena a un binario (descarta caracteres invalidos (<> de 0 o 1)
 * @param  String cad: Cadena a convertir.
 * @return StringBuffer out: Cadena en binario.
***/
public static StringBuffer toBin(String cad, int f) throws Exception{
     StringBuffer out = new StringBuffer("");
     int b = 0;
     f = 0;
          while(b < cad.length()){
           if (cad.charAt(b) =='0' || cad.charAt(b) =='1'){ // Descarta caracteres diferentes a 0 o 1
                        out.append(cad.charAt(b));
           }
          b++;
          }
         return out;
}

/** HEXTOBIN()
  * @Method HextoBin()
  * Funcion que convierte una cadena en hexa a binario.
  * @param  StringBuffer cad: Cadena a convertir.
  * @return StringBuffer out: Cadena en binario.
 ***/
public static StringBuffer HextoBin(StringBuffer cad)throws Exception{
         StringBuffer out= new StringBuffer("");
    int b = 0;
         while(b < cad.length()){
        if ((cad.charAt(b) >='0' && cad.charAt(b) <='9')||(cad.charAt(b) >='A' && cad.charAt(b) <='F')){
                out.append(HextoBinary(cad.charAt(b)));
                }
                b++;
        }
        return out;
}

/** HEXTOBIN()
  * @Method HextoBin()
  * Funcion que convierte una cadena en hexa de tamaño fijo a binario.
  * @param  StringBuffer cad: Cadena a convertir, int tam_c: Tamaño de la cadena, int tam: Tamaño total
  * @return StringBuffer out: Cadena en binario.
 ***/
public static StringBuffer HextoBin(StringBuffer cad, int tam_c,int tam)throws Exception{
      StringBuffer out = new StringBuffer("");
```

```
      int b = 0;
           while(b < cad.length() && tam_c + out.length( ) < tam){
           if  ((cad.charAt(b)  >='0'  &&  cad.charAt(b)  <='9')||(cad.charAt(b)  >='A'  &&  cad.charAt(b)
<='F')){
                     out.append(HextoBinary(cad.charAt(b)));
                }
                     b++;
          }
          return out;
}

}
```