GENERACIÓN DE TOKENS PARA PROTEGER LOS DATOS DE TARJETAS BANCARIAS

Trabajo terminal No. 2017-B008

Presentan

Daniel Ayala Zamorano

Laura Natalia Borbolla Palacios

Ricardo Quezada Figueroa

Directora

Dra. Sandra Díaz Santiago

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Índice

Notación

A continuación se describe la notación que se utilizará a lo largo de este documento.

Símbolo	Descripción
K	Llave
pk	Llave pública
sk	Llave privada o secreta
k_i	$I\acute{e}$-sima subllave
K_I	Llave de entrada o *key derivation key* de una función de derivación de llaves
K_O	Material de llaves obtenido de una función de derivación de llaves
IV	Vector de inicialización
E	Operación de cifrado
E_K	Operación de cifrado utilizando la llave K
D	Operación de descifrado
D_K	Operación de descifrado utilizando la llave K
h	Función hash
h_k	Función hash que utiliza una llave k para calcular el valor
M	Mensaje en claro
C	Mensaje cifrado
$\mathbb{Z}_n$	Conjunto de los números enteros módulo n
$\{0,1\}^r$	Cadena de bits de longitud r
$\{0,1\}^*$	Cadena de bits de longitud arbitraria
mód	Operación módulo
mcd	Máximo común divisor
$\oplus$	Operación XOR
$\boxplus$	Suma modular
$\boxminus$	Resta modular
$\|\|$	Operación de concatenación
$\{X\}$	Indicación de que el uso de X es opcional
$[X]$	El entero más pequeño que es mayor o igual que X
$[X]_2$	Representación binaria del número X
φ	Función ϕ de Euler
$\emptyset$	Conjunto vacío

Tabla 1: Notación

Es menester aclarar que un mensaje no consiste solo en letras y números; el *mensaje* se refiere al conjunto de datos que van a ser cifrados o descifrados.

Las palabras o frases CON ESTE ESTILO DE LETRA representan referencias a otras partes del documento: entradas al glosario, a la lista de siglas y acrónimos, referencias cruzadas. Si está leyendo esto en una versión digital, puede ocupar las referencias para navegar por el documento. En una versión digital, las referencias a la bibliografía, los números de página y los números de referencia a otras secciones también funcionan como enlaces para navegar por el documento.

Capítulo 1

Introducción

«In the beginning the Universe was created. This has made a lot of people very angry and has been widely regarded as a bad move.»
Douglas Adams.

A finales de los ochenta, el uso de las computadoras y el internet comenzó a popularizarse; compañías ya establecidas, como aerolíneas y tiendas departamentales, y comerciantes independientes vieron una oportunidad de expandirse y el comercio en línea comenzó a tomar fuerza; sin embargo, casi nadie previó el impacto y auge que iba a tener, por lo que la mayoría de los sitios no se encontraban preparados para los ataques y robos de información. Las principales emisoras de tarjetas (MasterCard y Visa) reportaron entre 1988 y 1998 pérdidas de 750 millones de dólares debidas a fraudes con tarjetas bancarias: el crecimiento del comercio electrónico aunado a sistemas débilmente protegidos dio lugar a un rápido crecimiento de los fraudes relacionados con tarjetas bancarias; de hecho, para el año 2001, según [1], se tuvieron pérdidas de 1.7 miles de millones de dólares y, para el siguiente año habían aumentado a 2.1 miles de millones de dólares.

Las compañías emisoras de tarjetas comenzaron a proponer soluciones y, a finales de 1999, Visa publicó un documento con una serie de recomendaciones de seguridad para quienes realizaban transacciones en línea llamado Cardholder Information Security Program (CISP), este programa es el primer precursor del estándar actual Payment Card Industry (PCI) Data Security Standard (DSS). Lamentablemente, las recomendaciones no estaban unificadas y había inconsistencias entre ellas, por lo que pocas compañías fueron capaces de satisfacer completamente alguna de las normas publicadas.

Fue hasta finales de 2004 cuando se publicó el primer estándar unificado, respaldado por las compañías emisoras de tarjetas más importantes: el PCI DSS 1.0, en el cuál se indica a los comercios cómo mantener los datos bancarios seguros mediante protocolos de seguridad, y se hizo obligatorio para todos aquellos que realizaran más de 20,000 transacciones anuales. Aunque cada vez más compañías comenzaron a seguirlo e invertir para satisfacerlo, de nuevo fueron pocas las que alcanzaron a cumplirlo completamente, pues tiene una gran cantidad de requerimientos (controles estrictos de acceso, monitoreo regular de las redes, mantener programas de vulnerabilidades y políticas de seguridad de información, etcétera) y las compañías tendían a subestimar los costos que implica seguir el estándar [2], [3].

A finales del 2006 ocurrió una de las primeras grandes violaciones de datos que puso en guardia a todos: hubo una intrusión en los servidores de la empresa americana TJX y piratas cibernéticos robaron información de tarjetas de crédito, débito y transacciones de 94 millones de clientes registrados en su sistema. Ataques como este han continuado a lo largo de los años: la cadena de supermercados Hannaford Bros. sufrió un embate en 2008 y se vieron comprometidas 4.2 millones de cuentas, Target fue atacado en 2013 y 40 millones de cuentas fueron afectadas, y, al año siguiente, arremetieron contra Home Depot y la información de 56 millones de usuarios fue robada.

Durante la primera década del siglo XXI, el enfoque que se tenía era salvaguardar la información sensible en todo el sistema; tómese por ejemplo el caso de una tienda en línea: el número de tarjeta queda registrado en el área de clientes, pues se puede asociar con un perfil y evita tener que estar ingresando continuamente toda la información de la tarjeta; también queda registrada en el área de ventas, pues queda asociada a una compra o transacción; la información sensible parece estar en todos lados (al menos, no

está concentrada) y tener que protegerla constantemente resulta muy costoso. Pasados unos años, surge la idea de cambiar completamente el paradigma que se había estado utilizando hasta ahora: ¿por qué intentar proteger la información sensible en todos los lados donde sea que se encuentre, cuando se puede cambiar esa información por un valor sustituto y solo proteger esa pequeña parte del sistema? Así que, a mediados del 2011, el PCI DSS publicó un documento cuyo título en inglés es «*PCI DSS tokenization guidelines*»; a su vez, varias compañías comenzaron a ofrecer soluciones de tokenización[1] que quitaban casi por completo el peso de cumplir con el PCI DSS a los comerciantes, pues ellas se encargan de generar los TOKENS y almacenar los datos sensibles; mientras que ellos, los comerciantes, se quedan solo con el TOKEN; así, si la seguridad de su sitio es violada, los datos siguen estando seguros, pues no se puede robar lo que no existe dentro del sistema.

Aunque en este trabajo se hace especial referencia a la tokenización de los números de tarjetas, o PERSONAL ACCOUNT NUMBER (PAN), este proceso sirve para proteger otros datos, como números de seguridad social, claves de registro, números de teléfono, etcétera. De hecho, la tokenización no está limitada al mundo digital y ha estado presente desde hace mucho tiempo, por ejemplo, los billetes y monedas, o las fichas en un casino: uno deposita dinero y obtiene su equivalente en fichas (este proceso es el de tokenización) para jugar; aunque las fichas (o TOKENS) están vigiladas, si alguien las roba, el casino no pierde tanto (siempre y cuando no sean cambiadas por dinero, o sea, realizar el proceso de detokenización), pues es una mera representación, y no el dinero mismo. Lo mismo sucede con los TOKENS digitales: sustituyen información valiosa, por valores que carecen de significado y cuya pérdida no representa un peligro inminente o una violación de la privacidad.

Todos los métodos para generar TOKENS que se presentan en este trabajo (y la gran mayoría de los métodos conocidos) competen enteramente a la criptografía. Entre otras cosas, la criptografía permite proteger información de terceros no autorizados, esto es, permite obtener confidencialidad. La idea básica es transformar un mensaje de forma que solo el destinatario sea capaz de hacer la transformación inversa y leer el mensaje original. Aunque con métodos un poco distintos a los actuales, la criptografía tiene una larga historia: el uso más antiguo del que se tiene noticia es en jeroglíficos egipcios de alrededor de 1900 a. C.; a partir de ahí, hay evidencias de su uso en muchas otras culturas antiguas [4]. La historia de la criptografía moderna está muy relacionada con la historia de la computación: en muchas ocasiones, la principal motivación para el desarrollo de máquinas más potentes fue la posibilidad de crear un método de cifrado infalible (o del método para romperlo); el ejemplo más claro de esto es el desarrollo de métodos para descifrar los mensajes de los nazis, durante la segunda guerra mundial, lo que permitió a Alan Turing sentar las bases de la computación actual [5]. Esta dinámica de juego, en la cuál cada participante está siempre buscando un método que los adversarios no pueden romper, hace de la criptografía una ciencia en constante movimiento: una vez que se encuentra una nueva vulnerabilidad en algún método, se trabaja

[1] El término *tokenización* es un anglicismo y, en este documento, se refiere a la acción y efecto de *tokenizar*; es decir, dado un valor, obtener un valor sustituto (TOKEN). La *detokenización* es el proceso inverso: dado el valor sustituto (TOKEN), obtener el valor original. Estos préstamos lingüísticos se utilizan de manera constante a lo largo de este reporte.

para corregirlo o para reemplazarlo.

Uno de los aspectos más importantes de las herramientas criptográficas modernas es que buscan probar, dentro de los límites posibles, la seguridad de un algoritmo. Es por esto que hoy en día no basta con describir un nuevo método y esperar que sea lo suficientemente fuerte, sino que la presentación de un método debe de ir acompañada de los argumentos (las pruebas) que garantizan su seguridad. Hacer esto no es tarea sencilla (si lo fuera, el juego ya estaría ganado para uno de los bandos) pues involucra hacer suposiciones sobre las capacidades de los adversarios y, basándose en esto, garantizar que los recursos necesarios para romper el método probado se encuentran muy por encima de las capacidades supuestas en un inicio. En este contexto, las capacidades de un adversario se determinan por su poder de cómputo: ¿cuántas computadoras tiene a su disposición?, ¿qué tan rápidas son?, etcétera.

Lo anterior nos lleva a otro principio muy importante de la criptografía moderna: la seguridad se encuentra en los datos, no en el método. La historia de la criptografía muestra que durante mucho tiempo, la seguridad de un método consistía totalmente en mantenerlo en secreto: mientras los adversarios no supieran cómo se estaba cifrando algo, los mensajes permanecerían seguros; sin embargo, si el método se filtraba, todos los mensajes cifrados con ese método se veían comprometidos. Este esquema presenta un serio problema, pues hace dependiente de un solo secreto, el método, la seguridad de todos los mensajes. Esto llevó a la introducción del concepto de la llave dentro de la criptografía: un valor que solo conocen las entidades autorizadas a leer el mensaje, sin el cuál no pueden descifrarlo; de esta forma la seguridad de un mensaje se basa tanto en el conocimiento del método, como en el conocimiento de la llave. En la mayoría de los métodos usados hoy en día, la seguridad se basa enteramente en la llave, dejando el método abierto al escrutinio de todos. Hacer esto presenta varias ventajas: una muestra de que el algoritmo está bien diseñado y es lo suficientemente fuerte es que no necesita mantenerse en secreto; cuando el algoritmo es público y es usado por muchas entidades (entre más mejor) son mayores las posibilidades de encontrar vulnerabilidades y el tiempo de respuesta de las correcciones es menor.

La presentación que se hace en este trabajo de los métodos de tokenización tiene un enfoque totalmente criptográfico. De esta forma se busca demostrar que la tokenización es una aplicación de la criptografía, no una alternativa a esta. Es por esto que todo el capítulo del marco teórico está dedicado a los mecanismos criptográficos que se relacionan con los métodos para generar TOKENS.

1.1. Objetivos

1.1.1. Objetivo general

Implementar algoritmos criptográficos y no criptográficos para generar TOKENS con el propósito de proveer confidencialidad a los datos de las tarjetas bancarias[2].

[2]Tomando en cuenta la clasificación propuesta por el PAYMENT CARD INDUSTRY (PCI) SECURITY STANDARD COUNCIL (SSC)

1.1.2. Objetivos específicos

- Revisar diversos algoritmos para la generación de TOKENS.
- Diseñar e implementar un servicio web que proporcione el servicio de generación de TOKENS de tarjetas bancarias a, al menos, una tienda en línea.
- Implementar una tienda web que use el servicio de generación de TOKENS.

1.2. Justificación

Dado que es un tema relativamente nuevo, la desinformación y la desconfianza respecto a la generación de TOKENS es latente aún, pues, por ejemplo, aunque el PAYMENT CARD INDUSTRY (PCI) DATA SECURITY STANDARD (DSS) describe los requerimientos que debe cumplir un sistema tokenizador, no indica qué hacer para satisfacerlos. Muchas empresas se aprovechan de esta desinformación para promocionarse como sistema tokenizador sin ser claras sobre sus métodos. A continuación se muestran las principales deficiencias encontradas como resultado de una investigación comparativa de las principales soluciones que existen en el mercado.

Shift4 Esta compañía reclama el crédito de haber inventado el paradigma de la tokenización. Denuncia que otras compañías y el mismo PCI SECURITY STANDARD COUNCIL (SSC) desvirtuaron el término; por esta razón, la compañía se refiere a su propio método cómo *TrueTokenization*. Aunque mencionan en todas partes las supuestas ventajas de sus métodos, lo único que dicen con claridad sobre la generación de TOKENS es que se trata de valores aleatorios, únicos y alfanuméricos. Esta descripción no representa en modo alguno una garantía, pues podrían estar usando un método cuya seguridad no esté formalmente probada [6], [7].

Bluepay TokenShield En este servicio, a pesar de que se ofrecen dos formas distintas de tokenizar, nunca se aclara cómo es que funciona alguno de estos procesos. Además, en [8] hay una breve descripción del flujo de datos del servicio, donde se dice que con el TOKEN se recupera y descifra la información bancaria sensible, lo que genera confusión, pues no se sabe si la tokenización se hace por medio de de cifrados que preservan el formato[3], o por medio de bases de datos con tablas que contienen pares PERSONAL ACCOUNT NUMBER (PAN)-TOKEN.

Braintree Compañía que ofrece distintos SOFTWARE DEVELOPMENT KIT (SDK) (para dispositivos móviles y para desarrollo web) que permiten interactuar con sus propios servidores para procesar pagos con tarjetas de crédito. En [9] dan una definición de TOKEN que no es compatible con el PCI SSC: la única forma de generar TOKENS, según su definición, es por métodos aleatorios. Aún

[3] Tipo de cifrado que permite que los mensajes ya cifrados se vean parecidos a los originales. Este tema se abarca en la sección 2.6.

aceptando esa definición, no se ofrecen mayores explicaciones: los clientes no saben de qué forma se están generando los valores aleatorios.

Merchant Link La solución de tokenización de Merchant Link es llamada TransactionVault. Su página da tres puntos importantes respecto a la tokenización:

- Utilizan tecnología de la *siguiente generación*.
- Los TOKENS son asociados con una tarjeta y una transacción.
- Todos los TOKENS generados tienen una longitud de 16 dígitos y pasan los controles de validez de las tarjetas bancarias (por ejemplo, el algoritmo de Luhn[4]).

Aunque en el tercer punto informan al cliente que los TOKENS creados preservan el formato, no indican cuáles algoritmos son utilizados para generarlos; aunado a esto, el segundo punto tampoco aporta mucha información: sí, utilizan algoritmos reversibles[5], pues los TOKENS quedan ligados a una tarjeta o a una transacción, pero siguen sin especificar sus métodos [10].

Jack Henry Card Processing Solution La solución de tokenización ofrecida por Jack Henry Banks indica que se encarga de sustituir el número de tarjeta con un TOKEN numérico que pasa los controles de validez de las tarjetas bancarias; sin embargo, no dice qué métodos utiliza para generar los TOKENS. Indica también que parte de su plataforma está integrada con los servicios de tokenización provistos por MasterCard Digital Enablement Service (MDES) y Visa Token Services (VTS); sin embargo, estos servicios tampoco indican cómo generan los tokens. Finalmente, no queda claro quién se encarga de generar los TOKENS, ¿lo hace MDES, VTS o Jack Henry? [11]-[13]

Securosis La publicación en [14] tiene por objeto esclarecer el papel de la tokenización. Explica de forma bastante clara las ventajas del paradigma y el flujo de datos entre tienda, sistema tokenizador y banco; sin embargo, al igual que las empresas anteriores, carece de una explicación sobre los propios TOKENS, y algunas de las referencias hacia estos solo consigue confundir. Por ejemplo, presentan a la tokenización como una alternativa a la criptografía, hablando de los TOKENS como valores aleatorios que nada tienen que ver con criptografía.

Muchas de las soluciones del mercado intentan argumentar que un TOKEN es un valor aleatorio y, por tanto, nada tiene que ver con criptografía; al parecer esta estrategia publicitaria tuvo mucho éxito, pues uno de los mensajes más comunes es que la criptografía es cosa del pasado, la tokenización permite sustituirla. Mientras esto puede ser cierto para los posibles clientes de un sistema tokenizador (tiendas que procesan pagos en línea), es una mentira manifiesta cuando se considera a todos los participantes.

La mayoría de las soluciones tratan a sus métodos como secretos de compañía: no explican la manera en que generan los TOKENS, por lo que esperan que el trato entre cliente y proveedor esté basado en la

[4] Este algoritmo es descrito en 3.1.

[5] Algoritmos que, dado un TOKEN, pueden regresar al PAN correspondiente. Esta clasificación es explicada en 3.2.

confianza que tiene el primero por el segundo. El modelo basado en la confianza es común al funcionamiento de casi todas las transacciones monetarias hechas por internet: dos partes que quieran realizar una transacción necesitan de un tercero (la institución financiera) para llevarla a cabo. Un estado ideal eliminaría la necesidad de la tercera parte (es lo que intentan hacer CRIPTOMONEDAS como *bitcoin* [15]), sin embargo, mientras este estado no se alcance, el modelo basado en la confianza es un mal necesario, y como tal, debe ser reducido al máximo: un proveedor del servicio de tokenización debería ser muy claro sobre lo que hace para tokenizar, permitiendo que sus clientes basen en esto su decisión.

La desinformación que hay alrededor del tema de la tokenización es la principal justificación para este trabajo: se busca implementar algoritmos públicos y presentar los resultado, permitiendo a los clientes de sistemas tokenizadores entender qué hay dentro de la caja negra. Parte de este combate a la desinformación consiste en dejar bien clara la relación entre la tokenización y la criptografía: la tokenización es una aplicación de la criptografía, por lo que los métodos de tokenización deben ser analizados con la misma formalidad con la que se estudian todas las demás formas de cifrado.

1.3. Organización del documento

El capítulo 2 provee un conjunto de resúmenes sobre los temas necesarios para la comprensión de este trabajo; adentrándose en la criptografía, sus objetivos y los métodos que emplea para la protección de información, tales como los cifradores por bloque y las funciones hash.

En el capítulo 3 se describen algunos de los estándares pertinentes a la generación de TOKENS, así como las descripciones de los algoritmos existentes para realizar esta labor.

El capítulo 4 muestra el análisis técnico para la implementación del programa tokenizador; abordando sus requerimientos y el diseño del propio programa.

El capítulo 5 contiene los fragmentos de código más importantes para entender el funcionamiento tanto de los algoritmos de tokenización como el programa de generación de TOKENS; también presenta los resultados de las comparaciones de desempeño entre los distintos algoritmos tokenizadores.

Por último, en el capítulo 6 se concluyen los resultados del trabajo, hablando sobre los avances que se hicieron y el trabajo que aún falta por hacerse.

Capítulo 2

Marco teórico

«Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.»
Douglas Adams.

El marco teórico contiene la mayoría de los conceptos utilizados para la implementación de los algoritmos generadores de TOKENS. Comienza con la introducción a la criptografía, sus objetivos, ataques y tipos. Posteriormente, se tiene un resumen de los cifrados por bloque, haciendo énfasis en el funcionamiento de las redes Feistel y el cifrador ADVANCED ENCRYPTION STANDARD (AES) y los modos de operación ELECTRONIC CODEBOOK (ECB), CIPHER-BLOCK CHAINING (CBC) y COUNTER MODE (CTR); después, se agregan unas nociones básicas sobre cifradores por flujos, funciones hash, códigos MESSAGE AUTHENTICATION CODE (MAC) y TWEAKABLE ENCYPHERING SCHEME (TES). El capítulo finaliza con los cifrados que preservan el formato. Cada sección indica al inicio las fuentes bibliográficas que fueron consultadas en caso de que el lector desee profundizar en algún tema tratado.

2.1. Introducción a la criptografía

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [16], [17].

La palabra criptografía proviene de las etimologías griegas *Kriptos* (ocultar) y *Graphos* (escritura), y es definida por la REAL ACADEMIA ESPAÑOLA (RAE) como «el arte de escribir con clave secreta o de un modo enigmático». De manera más formal se puede definir a la criptografía como la ciencia encargada de estudiar y diseñar por medio de técnicas matemáticas, métodos y modelos capaces de resolver problemas en la seguridad de la información, como la confidencialidad de esta, su integridad y la autenticación de su origen.

La criptografía tiene como finalidad proveer los siguientes cuatro servicios:

Confidencialidad Es el servicio encargado de mantener legible la información solo a aquellos que estén autorizados a visualizarla.

Integridad Este servicio se encarga de evitar la alteración de la información de forma no autorizada, esto incluye la inserción, sustitución y eliminación de los datos.

Autenticación Este servicio se refiere a la identificación tanto de las personas que establecen una comunicación, garantizando que cada una es quien dice ser; como del origen de la información que se maneja, garantizando la veracidad de la hora y fecha de origen, el contenido, tiempos de envío, entre otros.

No repudio Es el servicio que evita que el autor de la información o de alguna acción determinada, pueda negar su validez, ayudando así a prevenir situaciones de disputa.

2.1.1. Criptoanálisis y ataques

La criptografía forma parte de una ciencia más general llamada CRIPTOLOGÍA, la cual tiene otras ramas de estudio, como es el criptoanálisis que es la ciencia encargada de estudiar los posibles ataques a sistemas criptográficos, que son capaces de contrariar sus servicios ofrecidos. Entre los principales objetivos del criptoanálisis están interceptar la información que circula en un canal de comunicación, alterar dicha información y suplantar la identidad, rompiendo con los servicios de confidencialidad, integridad y autenticación respectivamente.

Los ataques que se realizan a sistemas criptográficos dependen de la cantidad de recursos o conocimientos con los que cuenta el adversario que realiza dicho ataque, dando como resultado la siguiente clasificación.

Ataque con sólo texto cifrado En este ataque el adversario solamente es capaz de obtener la información cifrada, y tratará de conocer su contenido en claro a partir de ella. Esta forma de atacar es la más básica, y todos los métodos criptográficos deben poder soportarla.

Ataque con texto en claro conocido Esta clase de ataques ocurren cuando el adversario puede obtener pares de información cifrada y su correspondiente información en claro, y por medio de su estudio, trata de descifrar otra información cifrada para la cual no conoce su contenido.

Ataque con texto en claro elegido Este ataque es muy parecido al anterior, con la diferencia de que en este el adversario es capaz de obtener los pares de información cifrada y en claro con el contenido que desee.

Ataque con texto en claro conocido adaptativo En este ataque el adversario es capaz de obtener los pares de información cifrada y en claro con el contenido que desee, además tiene amplio acceso o puede usar de forma repetitiva el mecanismo de cifrado.

Ataque con texto en claro elegido adaptativo En este caso el adversario puede elegir información cifrada y conocer su contenido, dado que tiene acceso a los mecanismos de descifrado.

2.1.2. Clasificación de la criptografía

La criptografía puede clasificarse de forma histórica en dos categorías, la criptografía clásica y la criptografía moderna. La criptografía clásica es aquella que se utilizó desde la antigüedad, teniéndose registro de su uso desde hace más 4000 años por los egipcios, hasta la mitad del siglo XX. En esta los métodos utilizados para cifrar eran variados, pero en su mayoría usaban la transposición y la sustitución, además de que la mayoría se mantenían en secreto. Mientras que la criptografía moderna es la que se inició después la publicación de la *Teoría de la información* por Claude Elwood Shannon[18], dado que esta sentó las bases matemáticas para la CRIPTOLOGÍA en general.

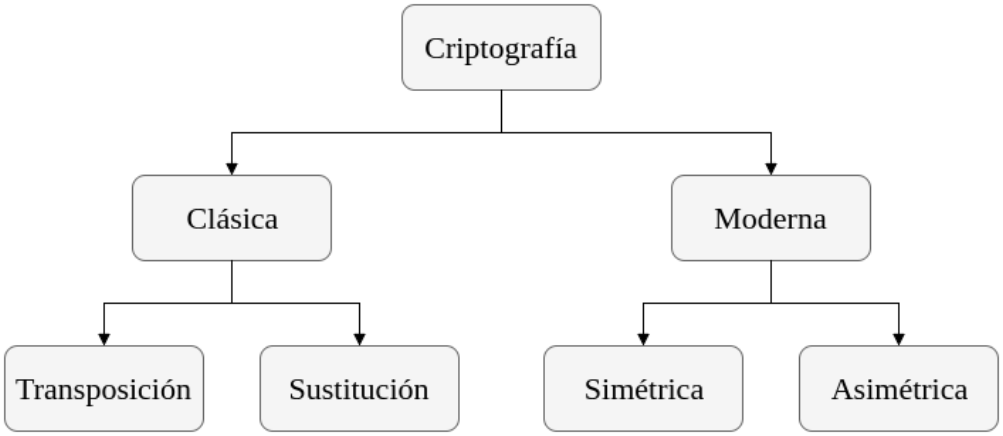


Figura 2.1: Clasificación de la criptografía.

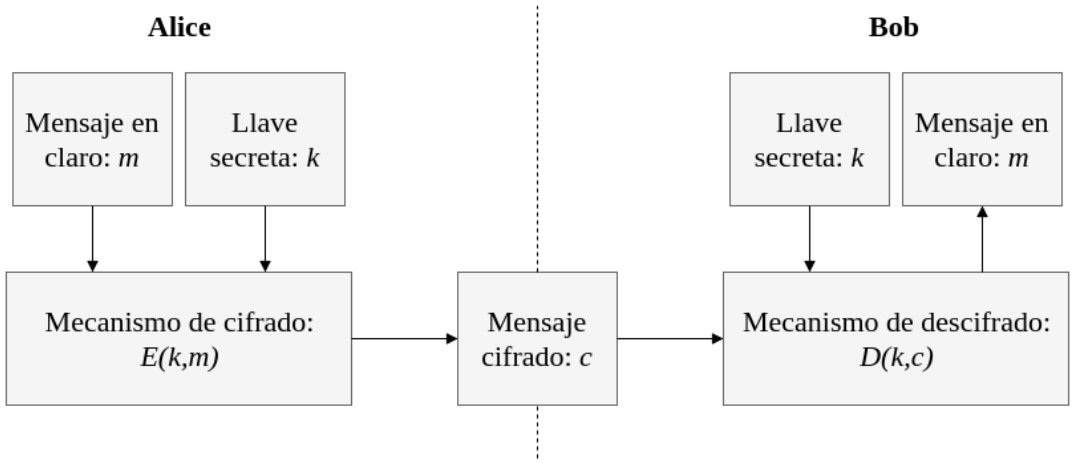


Figura 2.2: Canal de comunicación con criptografía simétrica.

Una manera de clasificar es de acuerdo a las técnicas y métodos empleados para cifrar la información, esta clasificación se puede observar en la figura 2.1.

Adentrándose en la clasificación de la criptografía clásica, se tienen los cifrados por transposición, los cuales se basan en técnicas de PERMUTACIÓN de forma que los caracteres de la información en claro se reordenen mediante algoritmos específicos, y los cifrados por sustitución, que utilizan técnicas de modificación de los caracteres por otros correspondientes a un alfabeto específico para el cifrado.

En cuanto a la criptografía moderna, esta tiene dos vertientes: la criptografía simétrica o de llave secreta, y la asimétrica o de llave pública. Hablando de la primer vertiente, se puede decir que es aquella que utiliza un modelo matemático para cifrar y descifrar un mensaje utilizando únicamente una llave que permanece secreta.

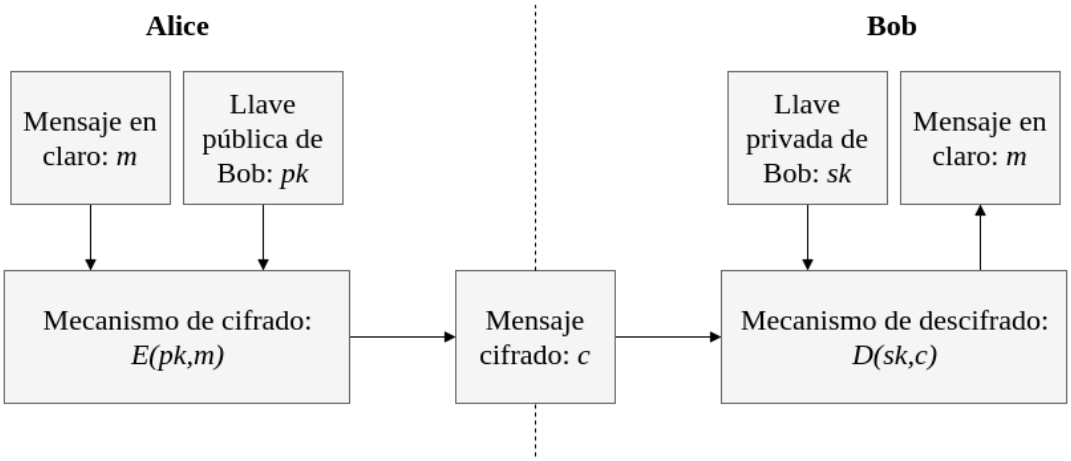


Figura 2.3: Canal de comunicación con criptografía asimétrica.

En la figura 2.2 se puede observar el proceso para establecer una comunicación segura por medio de la criptografía simétrica. Primero, tanto Alice como Bob deben de establecer una llave única y compartida k, para que después, Alice, actuando como el emisor, cifre un mensaje m usando la llave k por medio del algoritmo de cifrado $E(k, m)$ para obtener el mensaje cifrado c y enviárselo a Bob. Posteriormente Bob, como receptor, se encarga de descifrar c con ayuda de la llave k por medio del algoritmo de descifrado $D(k, c)$ para obtener el mensaje original m.

Gran parte de los algoritmos de cifrado que caen en este tipo de criptografía están basados en las redes Feistel, que son un método de cifrado propuesto por el criptógrafo Horst Feistel, mismo que desarrolló el Data Encryption Standard (DES) (sección 2.2.4) a principios de la década de los 70, que fue el cifrado usado por el gobierno estadounidense hasta 2002, año en que el Advanced Encryption Standard (AES) (sección 2.2.5) lo sustituyó.

Ahora, adentrándose en la criptografía asimétrica, se tiene que su idea principal es el uso de 2 llaves distintas para cada persona, una llave pública para cifrar, que esté disponible para cualquier otra persona, y una llave privada para descifrar, que se mantiene disponible solo para su propietario.

El proceso para establecer una comunicación segura por medio de este tipo de criptografía es el siguiente: primero, Alice nuevamente como el emisor, cifra un mensaje m con la llave pública de Bob pk, y usa el algoritmo de cifrado $E(pk, m)$ para obtener c y enviarlo. Después Bob como receptor, se encarga de descifrar c por medio del algoritmo de descifrado $D(sk, c)$ haciendo uso de su llave privada sk. Este proceso se refleja gráficamente el la figura 2.3.

Entre los uso que se le da a esta criptografía está el mantener la distribución de llaves privadas segura y establecer métodos que garanticen la autenticación y el no repudio; por ejemplo, en las firmas y certificados digitales.

2.2. Cifrados por bloques

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [16], [17].

Los cifrados por bloque son esquemas de cifrado que, como bien lo explica su nombre, operan mediante bloques de datos. Normalmente los bloques tienen una longitud de 64 o de 128 bits, mientras que las llaves pueden ser de 56, 128, 192 o 256 bits.

En muchos sistemas criptográficos, los cifrados por bloque simétricos son elementos importantes, pues su versatilidad permite construir con ellos generadores de números pseudoaleatorios, cifrados de flujo MACs y funciones hash. Sirven también como componentes centrales en técnicas de autenticación de mensajes, mecanismos de integridad de datos, protocolos de autenticación de entidad y esquemas de firma electrónica que usan llaves simétricas.

Los cifrados por bloque están limitados en la práctica por varios factores, tales como el límite de memoria, la velocidad requerida o restricciones impuestas por el hardware o el software en el que se implementan. Normalmente, se debe escoger entre eficiencia y seguridad

Idealmente, al cifrar por bloques, cada bit del bloque cifrado depende de todos los bits de la llave y del texto en claro; no debería existir una relación estadística evidente entre el texto en claro y el texto cifrado; el alterar tan solo un bit en el texto en claro o en la llave debería alterar cada uno de los bits del texto cifrado con una probabilidad de $\frac{1}{2}$; y alterar un bit del texto cifrado debería provocar resultados impredecibles al recuperar el texto en claro.

2.2.1. Definición

$$\begin{aligned} E : \{0,1\}^r \times \{0,1\}^n &\longrightarrow \{0,1\}^n \\ (k, m) &\longmapsto E(k, m) \end{aligned} \tag{2.1}$$

Utilizando una llave secreta k de longitud binaria r el algoritmo de cifrado E cifra bloques en claro m de una longitud binaria fija n y da como resultado bloques cifrados $c = E(k, m)$ cuya longitud también es n. n es el tamaño de bloque del cifrado. El espacio de llave está dado por $K = \{0,1\}^r$, para cada llave existe una función $D_k(c)$ que permite tomar un bloque cifrado c y regresarlo a su forma original m.

Generalmente, los cifrados por bloque procesan el texto claro en bloques relativamente grandes ($n \geq 64$), contrastando con los cifradores de flujo, que toman bit por bit. Cuando la longitud del mensaje en claro excede el tamaño de bloque, se utilizan los MODOS DE OPERACIÓN.

Los parámetros más importantes de los cifrados por bloque son los siguientes:

- Tamaño de bloque
- Tamaño de llave

2.2.2. Criterios para evaluar los cifrados por bloque

A continuación se listan algunos de los criterios que pueden ser tomados en cuenta para evaluar estos cifrados:

1. **Nivel de seguridad.** La confianza que se le tiene a un cifrado va creciendo con el tiempo, pues va siendo analizado y sometido a pruebas.

2. **Tamaño de llave.** La ENTROPÍA del espacio de la llave define un límite superior en la seguridad del cifrado al tomar en cuenta la búsqueda exhaustiva. Sin embargo, hay que tener cuidado con su tamaño, pues también aumentan los costos de generación, transmisión, almacenamiento, etcétera.

3. **Tamaño de bloque.** Impacta la seguridad, pues entre más grandes, mejor; sin embargo, tiene repercusiones en el costo de la implementación, además de que puede afectar el rendimiento del cifrado.

4. **Expansión de datos.** Es extremadamente deseable que los datos cifrados no aumenten su tamaño respecto a los datos en claro.

5. **Propagación de error.** Descifrar datos que contienen errores de bit puede llevar a recuperar incorrectamente el texto en claro, además de propagar errores en los bloques pendientes por descifrar. Normalmente, el tamaño de bloque afecta el error de propagación.

Entre los algoritmos de cifrado por bloque más usados se encuentran: DATA ENCRYPTION STANDARD (DES), ADVANCED ENCRYPTION STANDARD (AES), FAST DATA ENCIPHERMENT ALGORITHM (FEAL), INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA), SECURE AND FAST ENCRYPTION ROUTINE (SAFER) y RC5. A continuación se presentan las redes Fesitel, las cuales se utilizan para la construcción de otros cifrados por bloques, DES y AES.

2.2.3. Redes Feistel

Consiste en un CIFRADO ITERATIVO que mapea bloques de texto en claro de tamaño $2t$bits (separados en bloques L_0, R_0 de tamaño t) a un texto cifrado R_r, L_r mediante un proceso de r RONDAS.

Cada subllave K_i se obtiene de la llave K. Normalmente el número de rondas r es mayor o igual a tres y par. Además, casi siempre intercambia el orden de los bloques de salida al revés en la última ronda: (R_r, L_r) en vez de (L_r, R_r). El descifrado se realiza utilizando el mismo proceso de cifrado pero con las llaves en el orden inverso (comenzando con K_r hasta K_1).

```
1  inicio
2  para_todo i desde 1 hasta r:
3     L_i = R_{i-1}
4     R_i = L_{i-1} ⊕ f(R_{i-1}, K_i)
5  fin
6  fin
```

Pseudocódigo 2.1: Feistel, cifrado.

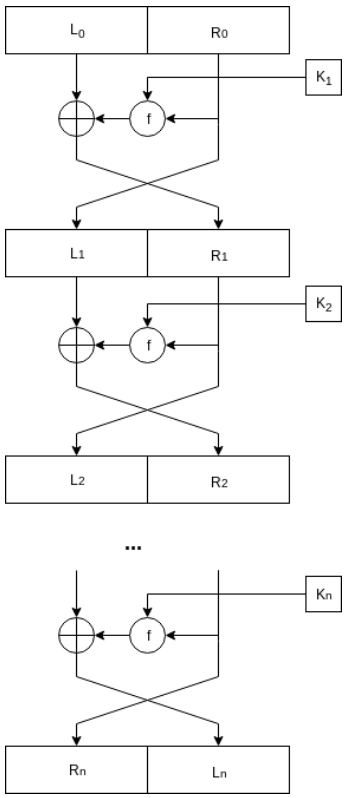


Figura 2.4: Diagrama genérico de una red Feistel.

Esta versión de las redes Feistel tiene como reestricción que la longitud del bloque a procesar debe ser par. Existen dos generalizaciones del esquema original que permiten procesar bloques de cualquier longitud: las redes Feistel desbalanceadas y las redes Feistel alternantes (figura 2.5) [19].

Redes Feistel desbalanceadas

Este tipo de redes (presentadas en [20]) permite que los bloques izquierdos y derechos sean de distintas longitudes (m y n, respectivamente). En la figura 2.5A se muestra un esquema del proceso, el cual es bastante similar al de la figura 2.4 con la única diferencia de que la función f debe cambiar la longitud de su entrada: de n a m. Si $m \leq n$, la red es pesada del lado de la fuente, y la función actúa como contracción (la entrada es más grande que la salida). Por otra parte, si $m \geq n$, la red es pesada del lado del objetivo, y la función actúa como una expansión (la entrada es más pequeña que la salida). El caso especial en el que $m = n$ es en el que la red está balanceada y corresponde al presentado originalmente (figura 2.4); es por esto que las redes Feistel desbalanceadas son consideradas una generalización del esquema inicial.

Para los esquemas pesados del lado de la fuente, la seguridad aumenta proporcionalmente al grado de desbalanceo. Por el lado contrario, en los esquemas pesados del lado del objetivo, entre más balanceada la red, mejor.

Redes Feistel alternantes

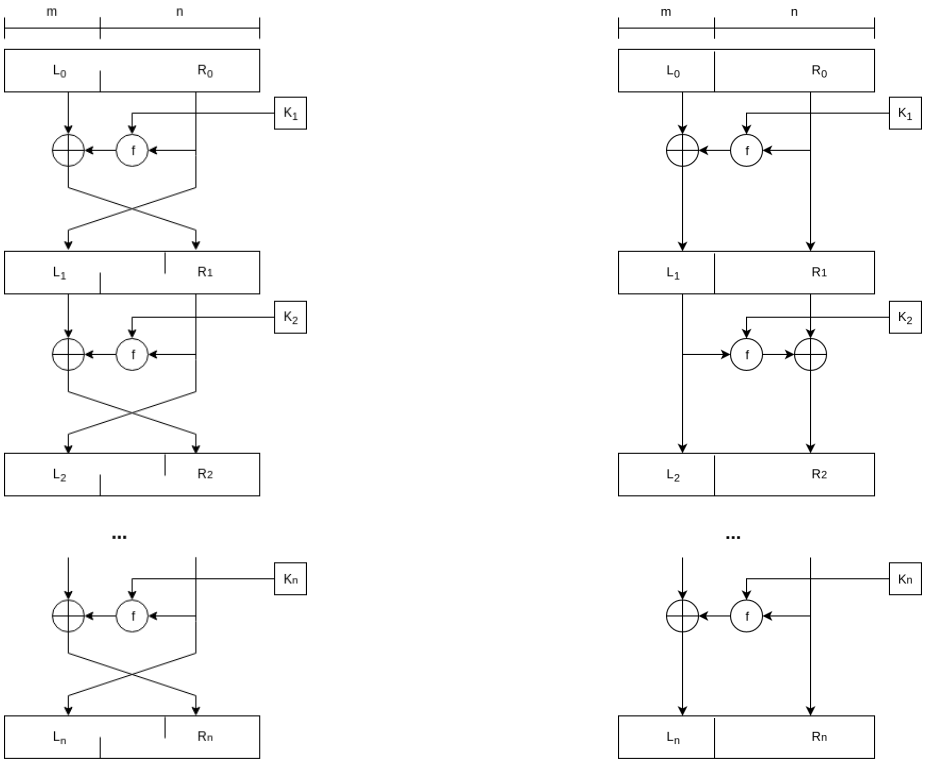
Un inconveniente de las redes desbalanceadas es el costo extra de hacer las particiones de los bloques intermedios. Las redes Feistel alternantes (figura 2.5B, presentadas en [21] y [22]) eliminan este inconveniente utilizando dos tipos de funciones, una contractora y la otra de expansión, en RONDAS alternas.

Es importante notar que las redes alternantes son también una generalización del esquema original, en la cuál la partición de los bloques es al centro, y se utliza una sola función.

2.2.4. Data Encryption Standard (DES)

Este es, probablemente, el cifrado simétrico por bloques más conocido; ya que en la década de los 70 estableció un precedente al ser el primer algoritmo a nivel comercial que publicó abiertamente sus especificaciones y detalles de implementación. Se encuentra definido en el estándar americano FEDERAL INFORMATION PROCESSING STANDARD (FIPS) 46-2.

DATA ENCRYPTION STANDARD (DES) es un cifrado Feistel que procesa bloques de $n = 64$ bits y produce bloques cifrados de la misma longitud. Aunque la llave es de 64 bits, 8 son de paridad, por lo que el tamaño *efectivo* de la llave es de 56 bits. Las 2^{56} llaves implementan, máximo, 2^{56} de las $2^{64}!$ posibles BIYECCIONES en bloques de 64 bits.



(a) Redes desbalanceadas. (b) Redes alternantes.

Figura 2.5: Generalizaciones de las redes Feistel.

```
entrada:  64 bits de texto en claro M = m_1 ... m_64;
          llave de 64 bits K = k_1 ... k_64.
salida:   bloque de texto cifrado de 64 bits C = c_1 ... c_64.
inicio
  Calcular 16 subllaves K_i de 48 bits partiendo de K.
  Obtener (L_0, R_0) de la tabla de permutaciones iniciales IP(m_1 m_2 ... m_64)
  para_todo i desde 1 hasta 16:
    L_i = R_{i-1}
    Obtener f(R_{i-1}, K_i):
      a) Expandir R_{i-1} = r_1 r_2 ... r_32 de 32 a 48 bits
         usando E: T ← E(R_{i-1}).
      b) T' ← T ⊕ K_i. Donde T' es representado
         como ocho cadenas de 6 bits cada una (B_1, ..., B_8).
      c) T'' ← (S_1(B_1), S_2(B_2), ... S_8(B_8))
      d) T''' ← P(T'')
    R_i = L_{i-1} \oplus f(R_{i-1}, K_i)
  fin
  b_1 b_2 ... b_64 ← (R_16, L_16).
  C ← IP^{-1}(b_1 b_2 ... b_64)
fin
```

Pseudocódigo 2.2: DES, cifrado.

Con la llave K se generan 16 subllaves K_i de 48 bits; una para cada RONDA. En cada RONDA se utilizan 8 *cajas-s* (mapeos de sustitución de 6 a 4 bits). La entrada de 64 bits es dividida por la mitad en L_0 y R_0. Cada RONDA i va tomando las entradas L_{i-1} y R_{i-1} de la RONDA anterior y produce salidas de 32 bits L_i y R_i mientras $1 \leq i \leq 16$ de la siguiente manera:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \\ &\quad donde \quad f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i)) \end{aligned} \tag{2.2}$$

E se encarga de expandir R_{i-1} de 32 bits a 48, P es una permutación de 32 bits y S son las cajas-s.

El descifrado DES consiste en el mismo algoritmo de cifrado, con la misma llave K, pero utilizando las subllaves en orden inverso: $K_{16}, K_{15}, \ldots, K_1$.

Llaves débiles

Tomando en cuenta las siguientes definiciones

- Llave débil: una llave K tal que $E_K(E_K(M)) = M$ para toda x; en otras palabras, una llave débil

permite que, al cifrar dos veces con la misma llave, se obtenga de nuevo el mensaje en claro.

- Llaves semidébiles: se tiene un par de llaves K_1, K_2 tal que $E_{K_1}(E_{K_2}(x)) = x$.

DES tiene cuatro llaves débiles y seis pares de llaves semidébiles. Las cuatro llaves débiles generan subllaves K_i iguales y, debido a que DES es un cifrado Feistel, el cifrado es autorreversible. O sea que al final se obtiene de nuevo el texto en claro, pues cifrar dos veces con la misma llave regresa la entrada original. Respecto a los pares semidébiles, el cifrado con una de las llaves del par es equivalente al descifrado con la otra (o viceversa).

2.2.5. Advanced Encryption Standard (AES)

Dado que el tamaño de bloque y la longitud de la llave de Data Encryption Standard (DES) se volvieron muy pequeños para resistir los embates del progreso de la tecnología, el National Institute of Standards and Technology (NIST) comenzó la búsqueda de un nuevo cifrado estándar en 1997; este cifrado debía tener un tamaño de bloque de, al menos, 128 bits y soportar tres tamaños de llave: 128, 192 y 256 bits.

Después de pasar por un proceso de selección, la propuesta Rijndael fue seleccionada. Se le hicieron algunas modificaciones, pues Rijndael soporta combinaciones de llaves y bloques de longitud 128, 169, 192, 224 y 256; mientras que Advanced Encryption Standard (AES) tiene fijo el tamaño de bloque y solo utiliza los tres tamaños de llave mencionados anteriormente. Dependiendo del tamaño de la llave, se tiene el número de rondas: 10 para las de 128 bits, 12 para las de 192 y 14 para las de 256.

El cifrado requiere de una matriz de 4×4 denominada matriz de estado.

Como todos los pasos realizados en las rondas son invertibles, el proceso de descifrado consiste en aplicar las funciones inversas a $SubBytes$, $ShiftRows$, $MixColumns$ y $AddRoundKey$ en el orden opuesto. Tanto el algoritmo como sus pasos están pensados con bytes. En el algoritmo Rijndael los bytes son considerados como elementos del campo finito $\mathbb{F}_{2^8}$ con 2^8 elementos; $\mathbb{F}_{2^8}$ es construido como una extensión del campo $\mathbb{F}_2$ con 2 elementos mediante el uso del polinomio irreducible $X^8 + X^4 + X^3 + X + 1$. Por lo tanto, las operaciones que se hagan a continuación de adición y el producto entre bytes significa sumarlos y multiplicarlos como elementos del campo $\mathbb{F}_{2^8}$.

SubBytes

Esta es la única transformación no lineal de Rijndael. Sustituye los bytes de la matriz de estado byte a byte al aplicar la función S_{RD} a cada elemento de la matriz. La función S_{RD} es también conocida como Caja-S y no depende de la llave. La misma caja es utilizada para los bytes en todas las posiciones.

```
1  entrada:    128 bits de texto en claro M; llave de n bits K.
2  salida:     bloque de texto cifrado de 64 bits C = c_1 ... c_64.
3  inicio
4    Obtener las subllaves de 128 bits necesarias: una para cada ronda y una extra.
5    Iniciar matriz de estado con el bloque en claro.
6    Realizar AddRoundKey(matriz_estado, k_0)
7    para_todo i desde 1 hasta num_rondas - 1:
8      SubBytes(matriz_estado)
9      ShiftRows(matriz_estado)
10     MixColumns(matriz_estado)
11     AddRoundKey(matriz_estado, k_i)
12   fin
13   SubBytes(matriz_estado)
14   ShiftRows(matriz_estado)
15   MixColumns(matriz_estado)
16   AddRoundKey(matriz_estado, k_num_rondas)
17   regresa matriz_estado
18 fin
```

Pseudocódigo 2.3: AES, cifrado.

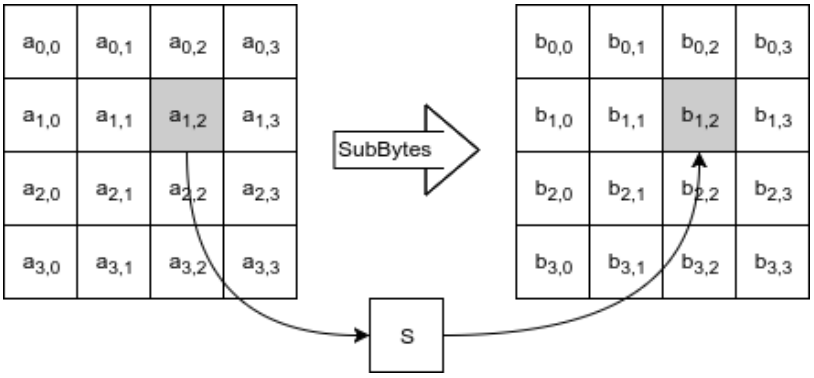


Figura 2.6: Diagrama de la operación *SubBytes*.

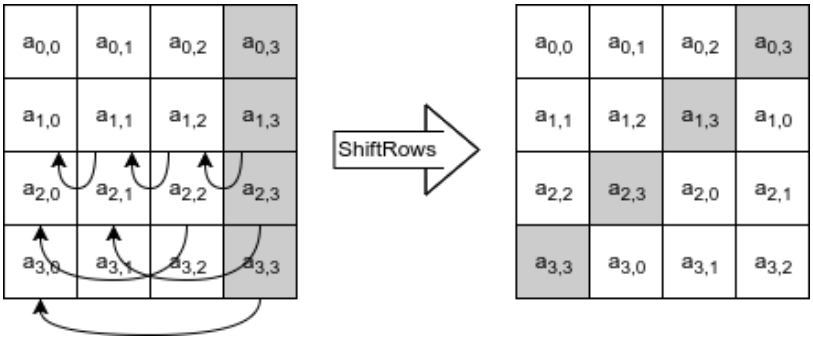


Figura 2.7: Diagrama de la operación $ShiftRows$.

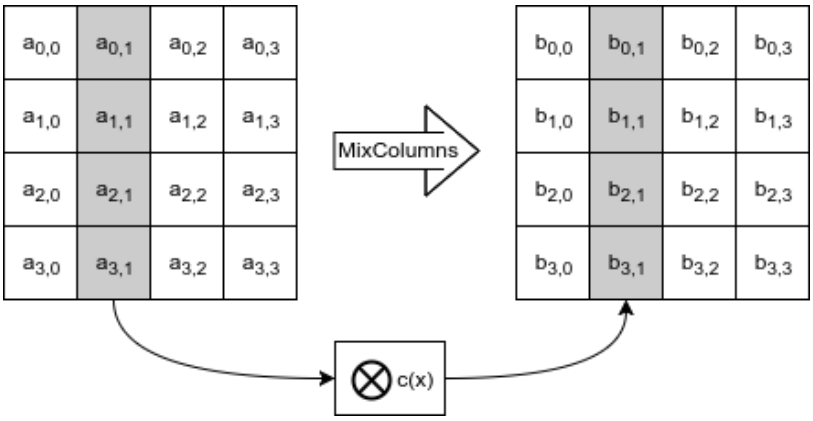


Figura 2.8: Diagrama de la operación $MixColumns$.

ShiftRows

Esta transformación hace un corrimiento cíclico hacia la izquierda de las filas de la matriz de estado. Los desplazamientos son distintos para cada fila y dependen de la longitud del bloque (N_b).

MixColumns

Esta transformación opera en cada columna de la matriz de estado independientemente. Se considera una columna $a = (a_0, a_1, a_2, a_3)$ como el polinomio $a(X) = a_3X^3 + a_2X^2 + a_1X + a_0$. Entonces este paso transforma una columna a al multiplicarla con el siguiente polinomio fijo:

$$c(X) = 03X^3 + 01X^2 + 01X + 02 \tag{2.3}$$

y se toma el residuo del producto módulo $X^4 + 1$:

$$a(X) \mapsto a(X) \cdot c(X) \quad \text{mód } (X^4 + 1) \tag{2.4}$$

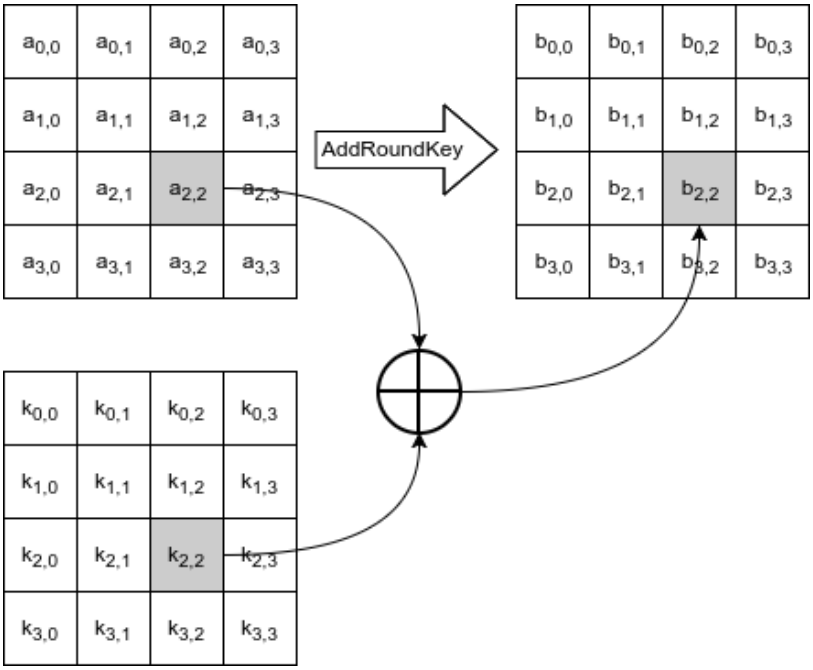


Figura 2.9: Diagrama de la operación *AddRoundKey*.

AddRoundKey

Esta es la única operación que depende de la llave secreta k. Añade una llave de ronda para intervenir en el resultado de la matriz de estado. Las llaves de ronda son derivadas de la llave secreta k al aplicar el algoritmo de generación de llaves. Las llaves de ronda tienen la misma longitud que los bloques. Esta operación es simplemente una operación XOR bit a bit de la matriz de estado con la llave de ronda en turno. Para obtener el nuevo valor de la matriz de estado se realiza lo siguiente:

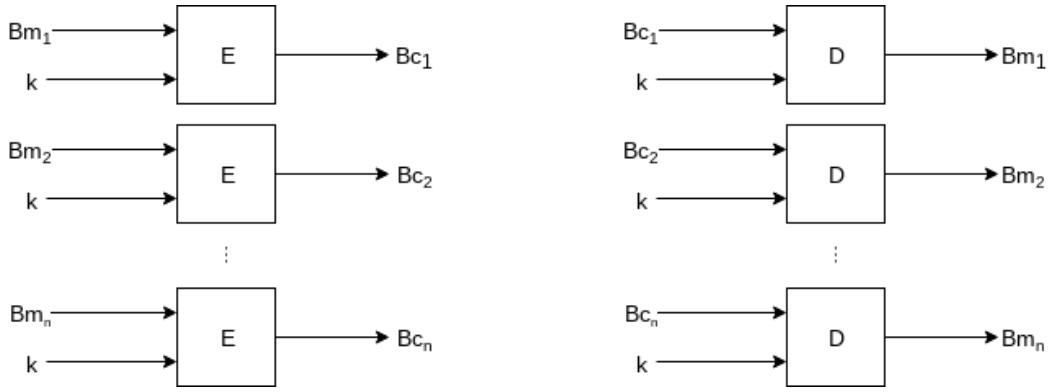
$$(matriz\_estado, k_i) \mapsto matriz\_estado \oplus k_i \tag{2.5}$$

Como se tiene una matriz_estado, la llave de ronda (k_i) también es representada como una matriz de bytes con 4 columnas y N_b columnas. Cada una de las N_b palabras de la llave de ronda corresponde a una columna. Entonces se realiza la operación XOR bit a bit sobre las entradas correspondientes de la matriz de estado y la matriz de la llave de ronda.

Esta operación, claro está, es invertible: basta con aplicar la misma operación con la misma llave para revertir el efecto.

2.2.6. Modos de operación

La información que aquí se presenta se puede consultar a mayor detalle en [23] y [16].



(a) Cifrado. (b) Descifrado.

Figura 2.10: Modo de operación ECB.

Por sí solos, los cifrados por bloques solamente permiten el cifrado y descifrado de bloques de información de tamaño fijo; donde, en la mayoría de los casos, los bloques son de menos de 256 bits, lo cual es equivalente a alrededor de 8 caracteres. Es fácil darse cuenta de que esta restricción no es ningún tema menor: en la gran mayoría de las aplicaciones, la longitud de lo que se quiere ocultar es arbitraria.

Los modos de operación permiten extender la funcionalidad de los cifrados por bloques para poder aplicarlos a información de tamaño irrestricto: reciben el texto original (de tamaño arbitrario) y lo cifran, ocupando en el proceso un cifrado por bloques.

Un primer enfoque (y quizás el más intuitivo) es partir el mensaje original en bloques del tamaño requerido y después aplicar el algoritmo a cada bloque por separado; en caso de que la longitud del mensaje no sea múltiplo del tamaño de bloque, se puede agregar información extra al último bloque para completar el tamaño requerido. Este es, de hecho, uno de los modos de operación que existen, el Electronic Codebook (ECB); su uso no es recomendado, pues es muy inseguro cuando el mensaje original es simétrico a nivel de bloque.

A pesar de que existen más modos de operación, como Cipher Feedback (CFB) u Output Feedback (OFB), en este trabajo solo se describirá el funcionamiento de ECB, Cipher-block Chaining (CBC) y Counter Mode (CTR), dado que son los modos de operación necesarios para el proceso de tokenización.

Electronic Codebook (ECB)

La figura 2.10 muestra un diagrama esquemático de este modo de operación. El algoritmo recibe a la entrada una llave y un mensaje de longitud arbitraria: la llave se pasa sin ninguna modificación a cada función del cifrado por bloques; el mensaje se debe de partir en bloques ($M = Bm_1||Bm_2||\ldots||Bm_n$).

```
1  entrada: llave k; bloques de mensaje Bm_1, Bm_2 ... Bm_n.
2  salida:  bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
3  inicio
4    para_todo Bm
5      Bc_i ← E_k(Bm_i)
6    fin
7    regresar Bc
8  fin
```

Pseudocódigo 2.4: MODO DE OPERACIÓN ECB, cifrado.

```
1  entrada: llave k; bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
2  salida:  bloques de mensaje original B_1, B_2 ... B_n.
3  inicio
4    para_todo Bc
5      Bm_i ← D_k(Bc_i)
6    fin
7    regresar Bm
8  fin
```

Pseudocódigo 2.5: MODO DE OPERACIÓN ECB, descifrado.

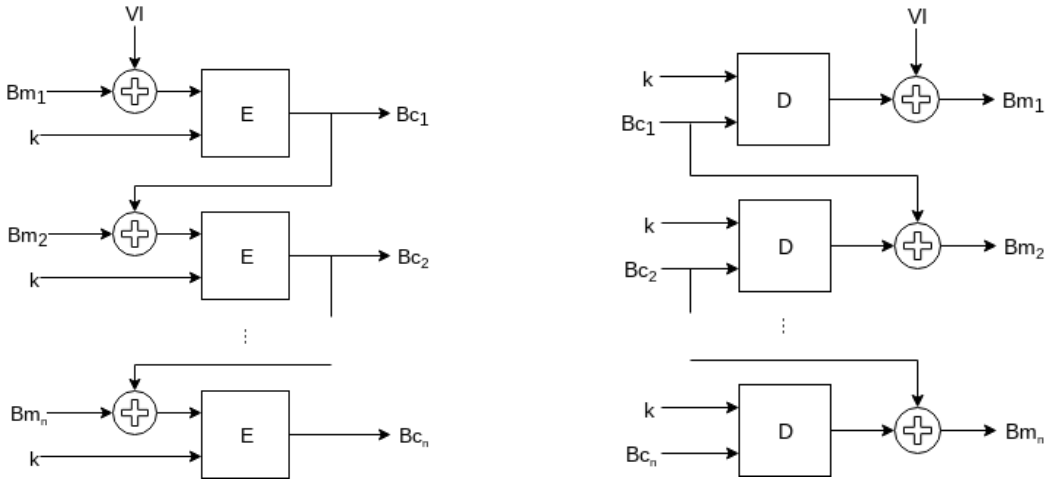
Cipher-block Chaining (CBC)

En CBC la salida del bloque cifrador uno se introduce (junto con el siguiente bloque del mensaje) en el bloque cifrador dos, y así en sucesivo. Para poder replicar este comportamiento en todos los bloque cifradores, este MODO DE OPERACIÓN necesita un argumento extra a la entrada: un VECTOR DE INICIALIZACIÓN. De esta manera la salida del bloque i depende de todos los bloques anteriores; esto incrementa la seguridad con respecto a ECB.

En la figura 2.11 se muestran los diagramas esquemáticos para cifrar y descifrar; en los pseudocódigos 2.6 y 2.7 se muestran unos de los posibles algoritmos a seguir. Es importante notar que mientras que el proceso de cifrado debe ser forzosamente secuencial (por la dependencias entre salidas), el proceso de descifrado puede ser ejecutado en paralelo.

Counter Mode (CTR)

Este toma a la entrada un VECTOR DE INICIALIZACIÓN y en cada iteración lo incrementa y lo cifra. El resultado se obtiene combinando el VI cifrado con el bloque de texto cifrado (mediante una operación `xor`). El proceso de cifrado y descifrado se detallan en el pseudocódigo 2.8.



(a) Cifrado. (b) Descifrado.

Figura 2.11: Modo de operación CBC.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
4  inicio
5    Bc_0 ← VI                    // El vector de inicialización
6    para_todo Bm                 // entra al primer bloque.
7      Bc_i ← E(k, Bm_i ⊕ Bc_{i-1})
8    fin
9    regresar Bc
10 fin
```

Pseudocódigo 2.6: Modo de operación CBC, cifrado.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
3  salida:  bloques de mensaje original Bm_1, Bm_2 ... Bm_n.
4  inicio
5    Bc_0 ← VI
6    para_todo Bc
7      Bm_i ← D_k(Bc_i) ⊕ Bc_{i-1}
8    fin
9    regresar Bm
10 fin
```

Pseudocódigo 2.7: Modo de operación CBC, descifrado.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje (cifrado o descifrado) Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje (cifrado o descifrado) Bc_1, Bc_2 ... Bc_n.
4  inicio
5    para_todo Bm
6      Bc_i ← E_k((VI + i) mód 2^n) ⊕ Bm_i
7    fin
8    regresar Bc
9  fin
```

Pseudocódigo 2.8: Modo de operación CTR(cifrado y descifrado).

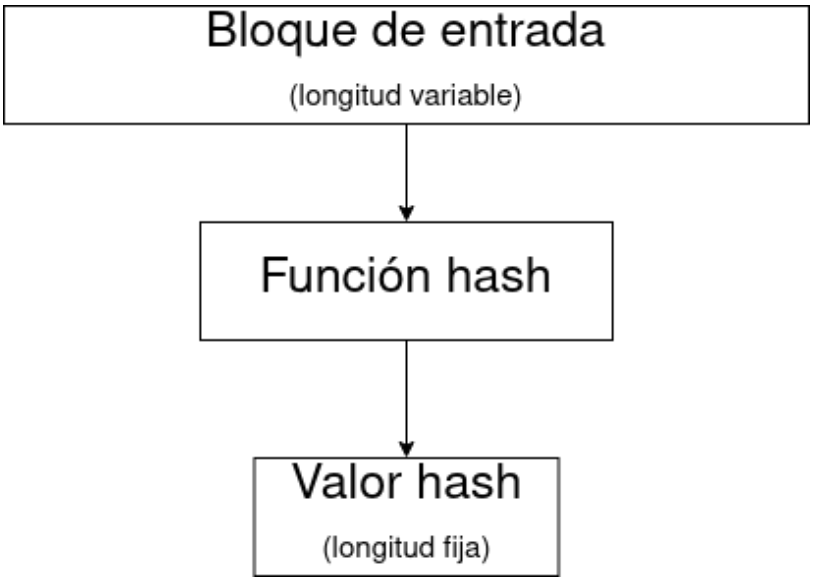


Figura 2.12: Diagrama del funcionamiento de una función hash.

En términos de eficiencia, el CTR es mejor que CBC, ya que sus operaciones (ambas) se pueden hacer en paralelo. La implementación es prácticamente la misma para el cifrado y descifrado (solamente se ocupa el cifrado del algoritmo por bloques subyacente).

2.3. Funciones hash

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [16], [17], [24], [25].

Se refiere al conjunto de funciones computacionalmente eficientes que mapean cadenas binarias de una longitud arbitraria a cadenas binarias de una longitud fija, llamadas valores hash.

Matemáticamente, una función hash es una función

$$\begin{aligned} h : \{0,1\}^* &\longrightarrow \{0,1\}^n \\ m &\longmapsto h(m) \end{aligned} \tag{2.6}$$

La longitud de n suele ser entre 128 y 512 bits. Las funciones hash h tienen las siguientes propiedades:

1. Compresión: h mapea una entrada x (cuya longitud finita es arbitraria) a una salida $h(x)$ de longitud fija n.

2. Facilidad de cómputo: dada x y h, $h(x)$ es calculada ya sea sin necesitar mucho espacio, tiempo de cómputo, o requiere pocas operaciones, etcétera.

De manera general, las funciones hash se pueden dividir en dos categorías: las que no utilizan llave y su único parámetro es la entrada x, y las que necesitan una llave secreta k y la entrada x.

Sea una función hash sin llave h con entradas x, x' y salidas y y y', respectivamente. A continuación se listan algunas de las propiedades que puede tener:

1. Resistencia de PREIMAGEN: no es computacionalmente factible para una salida específica y encontrar una entrada x' que dé como resultado el mismo valor hash $h(x') = y$ si no se conoce x. Esta propiedad también es llamada *de un sentido*.

2. Resistencia de segunda PREIMAGEN: no es computacionalmente factible encontrar una segunda entrada x' que tenga la misma salida que una entrada específica x: $x \neq x'$ tal que $h(x) = h(x')$. Esta propiedad también es conocida como *de débil resistencia a colisiones*.

3. Resistencia a las colisiones: no es computacionalmente factible encontrar dos entradas distintas x, x' que lleven al mismo valor hash, o sea, $h(x) = h(x')$. A diferencia de la anterior, la selección de ambas entradas no está restringida. Esta propiedad también es conocida como *de gran resistencia a colisiones*.

Una función hash h que cumple con las propiedades de resistencia de PREIMAGEN y resistencia de segunda PREIMAGEN es conocida como una función hash de un solo sentido o ONE-WAY HASH FUNCTION (OWHF). Las que cumplen con la resistencia de segunda PREIMAGEN y resistencia a las colisiones son conocidas como funciones hash resistentes a colisiones o COLLISION-RESISTANT HASH FUNCTION (CRHF). Aunque casi siempre las funciones CRHF cumplen con la resistencia de PREIMAGEN, no es obligatorio que lo hagan.

Algunos ejemplos de las funciones OWHF son el SECURE HASH ALGORITHM (SHA)-1 y el MESSAGE DIGEST-5 (MD5). En los esquemas de firma electrónica, se obtiene el valor hash del mensaje ($h(m)$) y se pone en el lugar de la firma. Los valores hash también son utilizados para revisar la integridad de las llaves

públicas y, al utilizarse con una llave secreta, las funciones criptográficas hash se convierten en códigos de autenticación de mensaje (Message Authentication Code (MAC), por sus siglas en inglés), una de las herramientas más utilizadas en protocolos como Secure Sockets Layer (SSL) e IPSec para revisar la integridad de un mensaje y autenticar al remitente.

Una de las aplicaciones más conocidas de las funciones hash es la de cifrar las contraseñas: en un sistema, en vez de almacenar la contraseña *clave*, se guarda su valor hash $h(clave)$. Así, cuando un usuario ingresa su contraseña, el sistema calcula su valor hash y lo compara con el que se tiene guardado. Realizar esto ayuda a evitar que las contraseñas sean conocidas para los usuarios con privilegios, como pueden ser los administradores.

2.3.1. Message Digest-4 (MD4)

En la década de 1990 esta función hash fue diseñada por Ronald Rivest. Tiene entradas de longitud arbitraria y la longitud de la salida procesada es de 128 bits. El Message Digest-4 (MD4) fue innovador y clave en el diseño para los algoritmos venideros de esta clase (como el Message Digest-5 (MD5)).

2.3.2. RIPEMD

Esta función hash, publicada en 1996, está basada en Message Digest-4 (MD4) y fue diseñada por Hans Dobbertin y otros. Consiste en dos formas equivalentes de la función de compresión de MD4. El algoritmo original (RIPEMD-160) devuelve bloques *procesados* de 160 bits; cuando en 1996 Hans descubrió una colisión en dos rondas, se desarrollaron nuevas versiones mejoradas: RIPEMD-128, RIPE-256, RIPE-320; las cuales dan bloques procesados de 128, 256 y 320 bits respectivamente.

2.3.3. *Secure Hash Algorithm (SHA)*

El algoritmo Secure Hash Algorithm (SHA) fue publicado por National Institute of Standards and Technology (NIST) y National Security Agency (NSA) en 1993; este algoritmo produce bloques de 160 bits y fue desarrollado para reemplazar al Message Digest-4 (MD4); sin embargo, poco después de haber sido publicado tuvo que ser quitado por problemas de seguridad. Actualmente, SHA es conocido como SHA-0.

En 1995, SHA-0 fue reemplazado por SHA-1; tiene una salida de la misma longitud que su predecesor y es una de las funciones hash más populares. Hay que destacar que la seguridad que brinda esta función es limitada, pues tiene el mismo nivel que un cifrado por bloques de 80 bits.

En 2002 NIST publicó tres funciones hash más: SHA-256, SHA-384 y SHA-512; esta familia de funciones hash es conocida como SHA-2 y fue desarrollada para cubrir la necesidad de una llave más

grande para poder empatar su tamaño con Advanced Encryption Standard (AES). Dos años más tarde, una nueva función hash fue agregada a la familia SHA-2: SHA-224.

Finalmente, en 2008, NIST inició un concurso para buscar al SHA-3 y en 2012 anunció al ganador: Keccak, una función hash desarrollada por Guido Bertoni, Joan Daemen, Michael Peeters y Gilles Van Assche. Esta función tiene una construcción completamente distinta a las familias anteriores.

2.4. Códigos de Autenticación de Mensaje (MAC)

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [16], [17], [26].

Las funciones Message Authentication Code (MAC) son la técnica simétrica estándar utilizada tanto para la autenticación como para la protección de la integridad de los mensajes. Dependen de unas llaves secretas que son compartidas entre las partes que se van a comunicar; cada una de las partes puede producir el MAC correspondiente para un mensaje dado. Como se explica a continuación, los MAC pueden ser obtenidos mediante cifradores de bloque, cifradores de flujo o de funciones hash criptográficas.

Las funciones hash con llave cuyo propósito específico es la autenticación de origen y garantizar la integridad de datos del mensaje son llamadas MAC. Estas funciones tienen como entrada una llave secreta k y un mensaje de longitud arbitraria y dan como resultado un mensaje de longitud n.

$$h_k : \{0,1\}^* \longrightarrow \{0,1\}^n \tag{2.7}$$

El algoritmo MAC más usado basado en un cifrador de bloque utiliza el modo de operación Cipher-block Chaining (CBC) (véase sección 2.2.6). Cuando Data Encryption Standard (DES) es utilizado como el cifrado de bloque E, el tamaño de bloque es de 64 bits y la llave MAC es de 56 bits.

Otra manera de construir MAC es mediante un algoritmo de Message Digest Cipher (MDC) que incluya una llave secreta k como parte de la entrada. Un ejemplo de esto es el algoritmo MD5-MAC; donde la función de compresión depende de la llave secreta k, que interviene en todas las iteraciones.

El algoritmo Message Authenticator Algorithm (MAA) fue diseñado en 1938 específicamente para obtener MAC en máquinas de 32 bits. El tiempo de ejecución es directamente proporcional a la longitud del mensaje y alrededor de cuatro veces más largo que el Message Digest-4 (MD4).

Pseudorandom Function Es posible generar códigos de verificación mediante el uso de funciones pseudoaleatorias Pseudorandom Function (PRF). Se define en 2.9 el algoritmo para obtener el código de verificación y en 2.10 el algoritmo para verificar si es correcto o no el MAC.

```
entrada:    llave k y mensaje m
salida:     código de autenticación t.
Sea F una función pseudoaleatoria tal que F:(K × X) → Y.
inicio
  t := F(k, m)
  regresar t
fin
```

Pseudocódigo 2.9: MAC mediante PRF, obtener código.

```
entrada:    llave k, mensaje m, código t.
salida:     resultado de la verificación r.
inicio
  si t = F(k, m) entonces:
    regresar verdadero
  si_no:
    regresar falso
fin
```

Pseudocódigo 2.10: MAC mediante PRF, verificar código.

CBC-MAC Este algoritmo está basado en el modo de operación CBC y una función F que puede ser, por ejemplo, un cifrador por bloques. Se encarga de cifrar con una llave l_1 todo el mensaje m, pero lo único que se toma en cuenta es el último bloque, que es tomado como el código de autenticación (véase figura 2.13). En algunos casos, se cifra de nuevo, con la misma función F, pero utilizando una llave l_2 distinta (véase figura 2.14).

CBC-MAC es considerado seguro cuando el mensaje m tiene una longitud múltiplo del tamaño de bloque del cifrador por bloques. Como se le han encontrado varias vulnerabilidades, algoritmos basados en CBC-MAC han sido propuestos, tales como el *eXtended CBC* o el One-key MAC (OMAC).

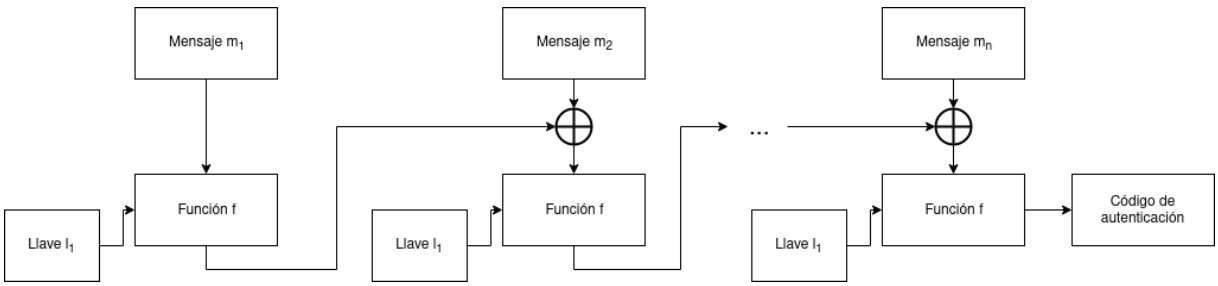


Figura 2.13: Esquema de CBC-MAC simple.

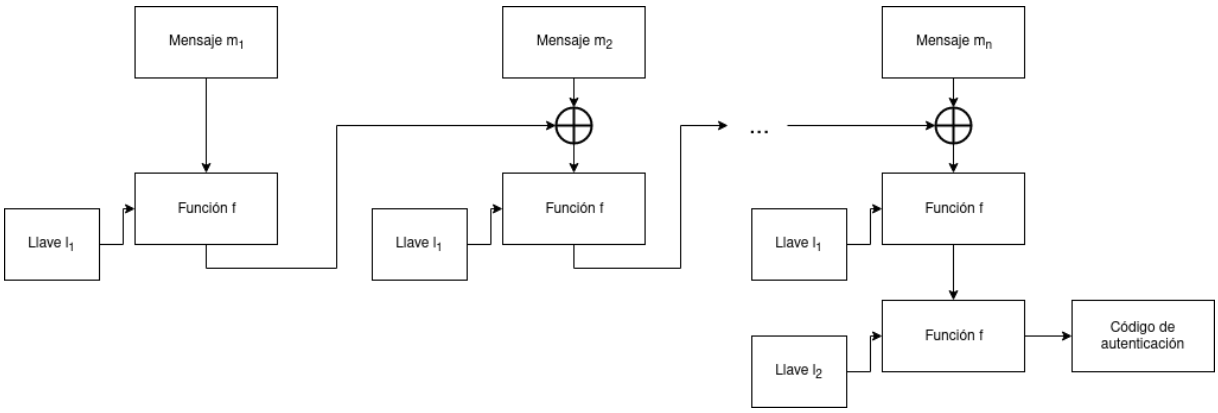


Figura 2.14: Esquema de CBC-MAC con el último bloque cifrado.

Keyed-Hashed Message Authentication Code Similar a Nested MAC (NMAC), este algoritmo utiliza una función h One-Way Hash Function (OWHF) para producir códigos de autenticación (véase figura 2.15). Tiene tres parámetros: $relleno_e$, $relleno_s$ y una llave l; los primeros dos se encargan de modificar la llave secreta l, por lo que se recomienda que los valores que asumen cambien lo más posible a l mediante la función h.

Este algoritmo es ampliamente utilizado, por ejemplo, se encuentra presente en los protocolos Secure Sockets Layer (SSL) y Secure Shell (SSH).

2.5. *Tweakable Encyphering Eschemes* (TES)

La información que aquí se presenta puede ser consultada con mayor detalle en [27] y [28].

La principal motivación para el diseño de Tweakable Encyphering Scheme (TES) son los Tweakable Block Cipher (TBC), los cuales fueron pensados como un medio de proveer de variabilidad a los cifrados por bloques. En el esquema original, un cifrado por bloques es totalmente determinístico: para el mismo mensaje y la misma llave, el texto cifrado generado es siempre el mismo. Los TBC agregan un *tweak* a la entrada de un cifrador por bloques para darle variabilidad; de esta manera, una misma llave y un mismo mensaje, ya no producen siempre el mismo texto cifrado. El papel del *tweak* es bastante similar al del vector de inicialización, solo que este último opera a nivel de los modos de operación.

Es importante resaltar las diferencias entre el papel del *tweak* y el papel de la llave: esta provee de incertidumbre a un adversario, mientras que el *tweak* proporciona variabilidad. Mantener el *tweak* en secreto no debería proporcionar mayores niveles de seguridad; de hecho, una condición en el diseño de TBC es que la seguridad del cifrado por bloques no se ve incrementada (ni decrementada) por la introducción del *tweak*.

En la ecuación 2.8 se muestra la firma para un cifrado por bloques con un *tweak*. Comparando esta

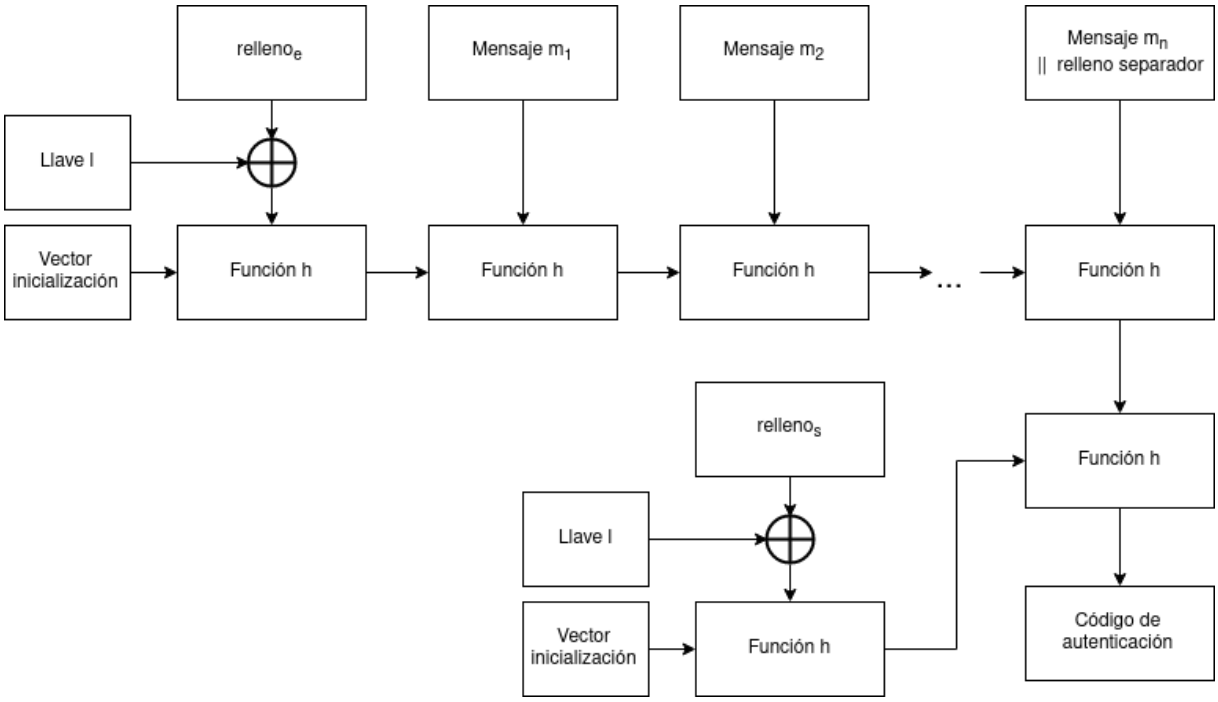


Figura 2.15: Esquema de HMAC.

función con la de la ecuación 2.1 (véase sección de cifrados por bloques, 2.2) se agrega un conjunto más al producto cruz de la entrada: una cadena de bits de tamaño t (el espacio de los *tweaks*).

$$\tilde{E} : \{0,1\}^k \times \{0,1\}^y \times \{0,1\}^n \longrightarrow \{0,1\}^n \tag{2.8}$$

En las siguientes ecuaciones se muestran algunas de las construcciones para TBC propuestas a la fecha:

$$\tilde{E}_k(T, M) = E_k(T \oplus E_k(M)) \tag{2.9}$$

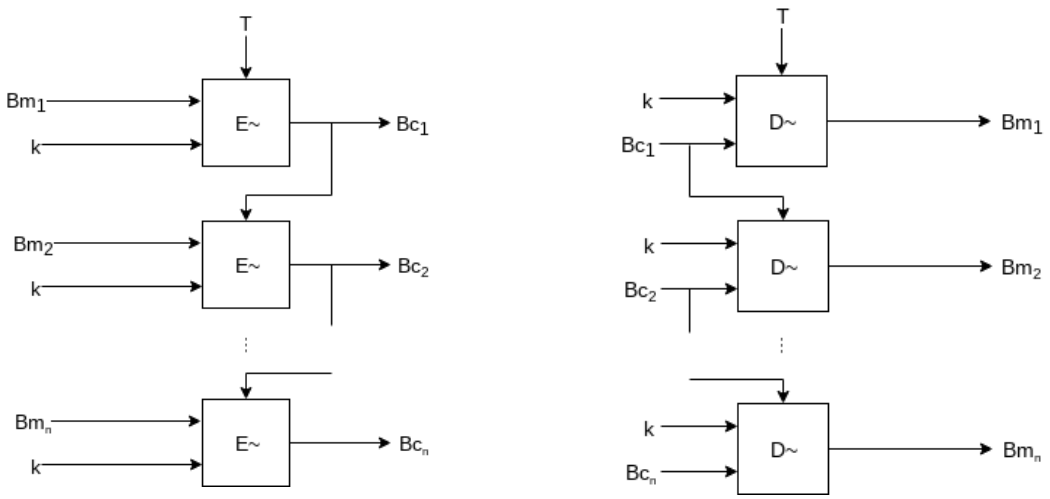
$$\tilde{E}_{k,h}(T, M) = E_k(M \oplus h(T)) \oplus h(T) \tag{2.10}$$

$$\tilde{E}_k(T, M) = E_k(M \oplus E_k(T)) \tag{2.11}$$

$$\tilde{E}_k(T, M) = E_k(M \oplus E_k(T)) \oplus E_k(T) \tag{2.12}$$

Con los TBC se pueden crear modos de operación análogos a los que se usan con los cifrados por bloques estándares. En la figura 2.16 el Tweakable Block Chaining (TBC), que es análogo a Cipher-block Chaining (CBC) (sección 2.2.6).

Los TBC son usados para la construcción de TES. Para estos existen dos clasificaciones: *encrypt-mask-encrypt* y *hash-counter-hash*. En la primera clasificación se encuentran aquellos que cuentan con dos capas de cifrado y una de enmascaramiento; algunos ejemplos de esta son CBC-Mask-CBC (CMC),



(a) Cifrado.

(b) Descifrado.

Figura 2.16: Modo de operación TBC.

Electronic Codebook (ECB)-Mask-ECB (EME) y Arbitrary Block Length Mode (ABL). La segunda consiste en dos funciones hash con un cifrado por bloques en modo de operación de contador (sección 2.2.6); algunos ejemplos son Extended Codebook (XCB), Hash Counter Mode (CTR) (HCTR) y HCH.

La cuestión que queda por esclarecer ahora es, ¿cuál es la ventaja de utilizar TBC en lugar de cifrados por bloques normales?, o en otro nivel, ¿cuál es la ventaja de utilizar modos de operación basados en *tweaks* a modos de operación con vectores de inicialización? Una primera impresión es que se trata de lo mismo: ambos, el *tweak* y el vector de inicialización, son entradas extras que permiten introducir variabilidad en los cifrados por bloques.

El primer punto clave es que los *tweaks* introducen una división semántica más en el análisis y diseño de modos de operación. En el esquema tradicional solo existen dos divisiónes: el nivel bajo, referente a los cifrados por bloques; y el nivel alto, referente a los modos de operación. Con los *tweaks*, el problema de los modos de operación se divide en dos: cómo diseñar buenos TBC, y cómo diseñar buenos modos de operación basados en TBC. Este nuevo enfoque permite filtrar nuevos problemas en un nivel en particular sin que se afecten a las construcciones de los otros niveles. El segundo punto clave es que los TES pueden ser diseñados para recibir mensajesde longitud arbitraria y regresar cifrados de su misma longitud.

2.6. Cifrados que preservan el formato

El objetivo de los cifrados que preservan el formato (en inglés, Format Preserving Encryption (FPE)) está bastante bien definido: lograr que los mensajes cifrados *se vean* igual que los mensajes en claro. Obviamente para que esto tenga sentido hay que definir qué es lo que hace que un mensaje se parezca a otro; los cifradores tradicionales, como Advanced Encryption Standard (AES), reciben cadenas

binarias y entregan cadenas binarias, por lo que, en cierto sentido, son ya un tipo de cifrados que preservan el formato. En una clase más general de algoritmos FPE el formato debe ser una parámetro: cadenas binarias, caracteres American Standard Code for Information Interchange (ASCII) imprimibles, dígitos decimales, dígitos hexadecimales, etcétera.

Desde un inicio, los cifrados que preservan el formato se perfilaron como una posible solución para el problema de la tokenización: usando un alfabeto de dígitos decimales se logra que los *tokens* y los Personal Account Number (PAN) tengan el mismo aspecto. De las posibles soluciones presentadas en la sección 3.4, FPE es la que presenta un esquema más tradicional dentro de la criptografía, ya que el proceso de cifrado y descifrado es el mismo que un cifrador simétrico.

La utilidad de los cifrados que preservan el formato se centra principalmente en *agregar* seguridad a sistemas y protocolos que ya se encuentran en un entorno de producción. Estos son algunos ejemplos de dominios comunes en FPE:

- Números de tarjetas de crédito.
  `5827 5423 6584 2154` → `6512 8417 6398 7423`
- Números de teléfono.
  `55 55 54 75 65` → `55 55 12 36 98`
- CURP.
  `GHUJ887565HGBTOK01` → `QRGH874528JUHY01`

2.6.1. Clasificación de los cifrados que preservan el formato

En [19] Phillip Rogaway propone una clasificación para los cifrados que preservan el formato que se basa en el tamaño del espacio de mensajes ($N = |X|$):

Espacios minúsculos El espacio es tan pequeño que es aceptable gastar $O(N)$ en un preproceso de cifrado. Esto es, cifrar todos los posibles mensajes de una sola vez, para que las subsiguientes solicitudes de cifrado y descifrado consistan en simples consultas a una base de datos.

El tamaño de N depende del contexto en el que se vaya a utilizar el algoritmo; para el contexto del problema de la tokenización ($N \approx 10^{16}$) no resulta viable utilizar esta técnica.

Algunos ejemplos de cómo hacer el preproceso de cifrado son el *Knuth shuffle* (también conocido como *Fisher-Yates shuffle*, pseudocódigo 2.11) o un cifrado con prefijo.

Espacios pequeños En esta clasificación se colocan a los espacios de mensaje cuyo tamaño no es más grande que 2^w en donde w es el tamaño de bloque del cifrado subyacente. Para AES, en donde $w = 128$, $N = 2^{128} \approx 10^{38}$.

```
1  entrada: lista l[0,...,n-1]
2  salida: misma lista barajeada
3  inicio
4    para_todo i desde n-1 hasta 0:
5      j ← rand(0, i)
6      swap(l[j], l[i])
7    fin
8  fin
```

Pseudocódigo 2.11: *Knuth shuffle*, [29].

En este esquema, el mensaje se ve como una cadena de n elementos pertenecientes a un alfabeto de cardinalidad m (i. e. $N = m^n$). Por ejemplo, para números de tarjetas de crédito, $n \approx 16$ y $m = 10$, por lo que $N = 10^{16}$ (diez mil trillones); lo cual es aproximadamente 2.93×10^{-21} % de 2^{128}.

Los algoritmos que preservan el formato expuestos en la sección 3.4 pertenencen a esta categoría.

Espacios grandes El espacio es más grande que 2^w. Para estos casos, el mensaje se ve como una cadena binaria. Las técnicas utilizadas incluyen cualquier cifrado cuya salida sea de la misma longitud que la entrada (p. ej. los TES: CMC, EME, HCH, etc. Sección 2.5).

Como se puede observar de los ejemplos dados, el problema de la tokenización de números de tarjetas de crédito es un problema de espacios pequeños.

Capítulo 3

Estándares y publicaciones sobre la generación de *tokens*

«*Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.*»
John von Neunmann.

Este capítulo inicia con un resumen sobre la composición de los números de tarjeta, posteriormente, se incluye una serie de resúmenes sobre los estándares publicados por el Payment Card Industry (PCI) Security Standard Council (SSC) y por el National Institute of Standards and Technology (NIST) necesarios para las implementaciones realizadas; por ejemplo, las recomendaciones para la generación y administración de llaves y la generación de bits pseudoaleatorios. Dado que generadores de números pseudoaleatorios son utilizados frecuentemente en los algoritmos tokenizadores, también se incluye un resumen de las pruebas a las que deben someterse. Finalmente, se clasifican los algoritmos que serán implementados y, para cada uno, se agrega una subsección donde se explica su notación, composición y funcionamiento.

3.1. Composición del número de una tarjeta

La información presentada a continuación puede consultarse con más detalle en los siguientes documentos [30]-[32].

Como se puede observar en la figura 3.1, un número de tarjeta Personal Account Number (PAN), se compone por tres partes: el número identificador del emisor (Issuer Identification Number (INN)), el número de cuenta y un dígito verificador; la longitud del PAN puede variar e ir desde los 12 hasta los 19 dígitos. A continuación se explica con más detalle cada uno de sus componentes

3.1.1. Identificador del emisor

El Issuer Identification Number (INN) comprende los primeros 6 dígitos del número de tarjeta. El primer dígito es conocido como Major Industry Identifier (MII) y se encarga de identificar la industria a la que pertenece el emisor; en la tabla 3.1, se puede observar la relación entre el dígito y el giro. El identificador del emisor provee, entre otros, los siguientes datos:

1. Emisor de la tarjeta

2. Tipo de la tarjeta (ej. crédito o débito)

3. Nivel de la tarjeta (ej. clásica, Gold, Black)

3.1.2. Número de cuenta

Todos los dígitos posteriores al Issuer Identification Number (INN) y anteriores al último dígito, comprenden el número de cuenta. La longitud de este puede variar, pero, máximo comprende 12 dígitos, por lo que cada emisor tiene 10^{12} posibles números de cuenta.

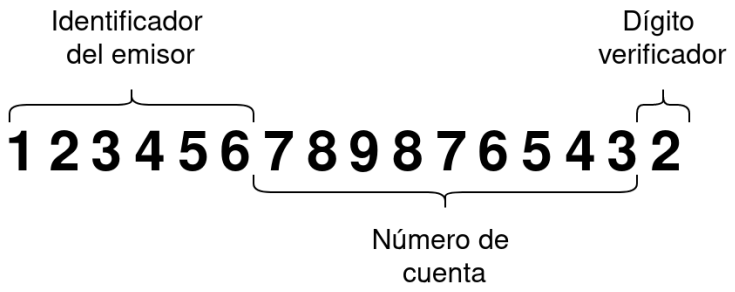


Figura 3.1: Componentes de un número de tarjeta.

Dígito	Industria
1, 2	Aerolíneas
3	Viajes y entretenimiento (ej. American Express)
4, 5	Bancos e industria financiera (ej. Visa, Mastercard)
6	Comercio (ej. Discover)
7	Industria petrolera
8	Telecomunicaciones
9	Asignación nacional

Tabla 3.1: Identificador de industria (MII).

3.1.3. Dígito verificador

Finalmente, se tiene el dígito verificador; este toma en cuenta todos los dígitos anteriores y se calcula mediante el algoritmo de Luhn, que se describe en 3.1.

3.2. Estándares del PCI DSS

En [33], el Payment Card Industry (PCI) Security Standard Council (SSC) divide a los tokens en reversibles e irreversibles. A su vez, los reversibles se dividen en criptográficos y en no criptográficos; mientras que los irreversibles se dividen en autenticables y no autenticables (figura 3.2).

Los tokens irreversibles no pueden, bajo ninguna circunstancia, ser reconvertidos al Personal Account Number (PAN) original. Esta restricción aplica tanto para cualquier entidad en el entorno del negocio (comerciante, proveedor de tokens, banco) como para cualquier posible atacante. Dados un PAN y un token, los identificables permiten validar cuando el primero fue utilizado para la creación del segundo, mientras que los no identificables, no.

La clasificación del PCI SSC con respecto a los reversibles resulta un poco confusa: establece que los criptográficos son generados utilizando criptografía fuerte, el PAN nunca se almacena, solamente se guarda una llave; los no criptográficos guardan la relación entre tokens y PAN en una base de datos. El

```
1  entrada: número de tarjeta, compuesto desde 12 hasta 19 dígitos:
2           {x_n x_{n-1} x_{n-2} ... x_3 x_2 x_1}
3  salida: dígito verificador x_1.
4  inicio
5    Sean los conjuntos x_par y x_impar:
6        si n es par:
7            x_par = {x_2, x_4, x_6, ..., x_n}
8            x_impar = {x_3, x_5, x_7, ..., x_{n-1}}.
9        si_no:
10           x_par = {x_2, x_4, x_6, ..., x_{n-1}}
11           x_impar = {x_3, x_5, x_7, ..., x_n}.
12   Obtener el doble de cada uno de los elementos del conjunto x_par:
13     x_pardoble = {2 × x_2, 2 × x_4, 2 × x_6, ...}.
14   ∀x_i ∈ x_pardoble > 9:
15       x_i = (x_i mód 10) + 1
16   Sumar los elementos de los conjuntos x_pardoble y x_impar.
17   x_1 = (S × 9) mód 10
18   Regresar x_1
19 fin
```

Pseudocódigo 3.1: Algoritmo de Luhn.

problema está en que no se menciona *cómo* generar los no criptográficos. A pesar del nombre, los métodos más comunes para esta categoría ocupan PRIMITIVAS CRIPTOGRÁFICAS (p. ej. generadores pseudoaleatorios); además de que, en una implementación real, para poder cumplir con el PCI DATA SECURITY STANDARD (DSS), la propia base de datos debe de estar cifrada [34].

3.2.1. Primitivas criptográficas

En esta sección se resumen los requerimientos mínimos que debe de tener cualquier PRIMITIVA CRIPTOGRÁFICA que se use dentro del sistema *tokenizador*. Esta información se presenta en el anexo C de [33], el cual está clasificado como *informativo* solamente (no *normativo*). Con respecto a las PRIMITIVAS CRIPTOGRÁFICAS, la parte *normativa* está controlada por el programa CRYPTOGRAPHIC ALGORITHM VALIDATION PROGRAM (CAVP) del NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST); el cual evalúa las implementaciones usadas de una serie de primitivas comunes[1].

En la tabla 3.2 se colocan los tamaños mínimos de llaves y MODOS DE OPERACIÓN (sección 2.2.6) asociados para las PRIMITIVAS CRIPTOGRÁFICAS de los «algoritmos criptográficos»[2] permitidos. Estos son:

[1]Esta lista se puede encontrar en HTTPS://CSRC.NIST.GOV/PROJECTS/CRYPTOGRAPHIC-ALGORITHM-VALIDATION-PROGRAM

[2]El PAYMENT CARD INDUSTRY (PCI) SECURITY STANDARD COUNCIL (SSC) parece dividir a las PRIMITIVAS CRIPTOGRÁFICAS en *algoritmos criptográficos*, *funciones hash* y *generadores de números pseudoaleatorios*; esto resulta confuso dado que las

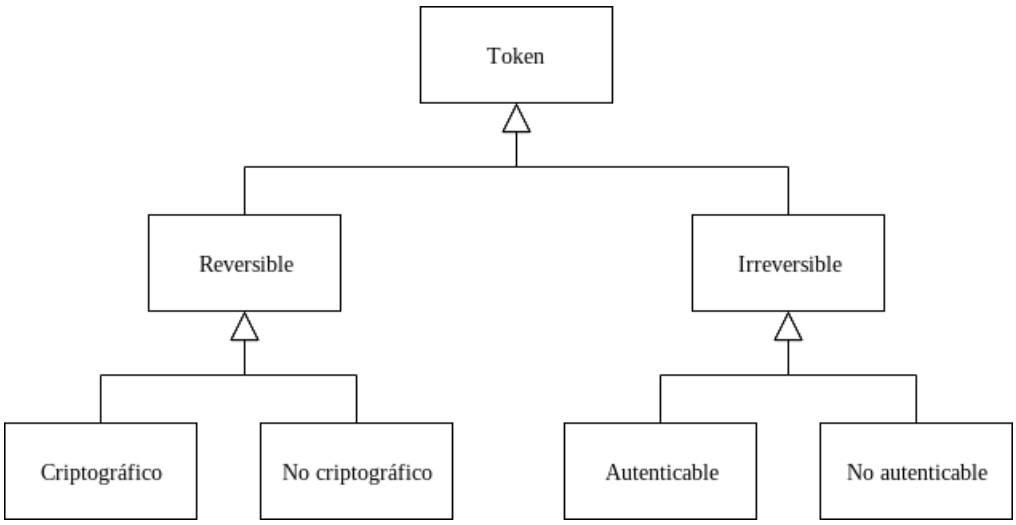


Figura 3.2: Clasificación de los TOKENS según PCI SSC.

Algoritmo	**Tamaño de llave**	**Modo de operación**
AES	128	CTR, OCB, CBC, OFB, CFB
RSA	3072	RSAES-OAEP
ECC	256	ECDH, ECMQV, ECDSA, ECIES
DSA/DH	3072/256	DHE

Tabla 3.2: Longitudes de llave mínimas y MODOS DE OPERACIÓN permitidos para algoritmos criptográficos

ADVANCED ENCRYPTION STANDARD (AES) (sección 2.2.5), RON RIVEST, ADI SHAMIR, LEONARD ADLEMAN (RSA) (sección 2.1.2), ELLIPTIC CURVE CRYPTOSYSTEM (ECC) y DIGITAL SIGNATURE ALGORITHM (DSA)/DIFFIE-HELLMAN (DH). Se hace especial énfasis en que TRIPLE DATA ENCRYPTION STANDARD (DES) (TDES) no está permitido.

La tabla 3.3 enlista los algoritmos hash (sección 2.3) permitidos. Para evitar introducir fallas de seguridad a través de las funciones hash, estas deben proveer al menos tantos bits de seguridad como el algoritmo criptográfico usado, y en cualquier caso, no menos de 128 bits (lo que deja fuera a MESSAGE DIGEST-4 (MD4) y MESSAGE DIGEST-5 (MD5)).

El número de bits de entropía utilizados para los generadores de números aleatorios debe de ser mayor o igual al número de bits de seguridad utilizados para las primitivas anteriores. Cuando se utilicen generadores determinísticos, estos deben seguir las recomendaciones del NIST en [35].

tres categorías pertenecen al campo de estudio de la criptografía.

Bits de seguridad	Algoritmo hash
128	Secure Hash Algorithm (SHA)-256
128	SHA3-256
192	SHA3-384
256	SHA-512
256	SHA3-512

Tabla 3.3: Algoritmos hash permitidos

3.3. Estándares del NIST

Generación de bits pseudoaleatorios (Sección 3.3.1)

800-90A Revision 1 *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Disponible en [35].

Administración de llaves (Apéndice A)

800-57 *Recommendation for Key Management*. Disponible en [36].

800-130 *A Framework for Designing Cryptographic Key Management Systems*. Disponible en [37].

Pruebas estadísticas para generadores pseudoaleatorios (Apéndice B)

800-22A Revision 1 *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Disponible en [38].

3.3.1. Generación de bits pseudoaleatorios

Existen dos maneras de generar bits aleatorios: la primera es producir bits de manera no determinística, donde el estado de cada uno (uno o cero) está determinado por un proceso físico impredecible. Estos generadores de bits aleatorios (Random Bit Generator (RBG)) son conocidos como generadores no determinísticos, o Non-deterministic Random Bit Generator (NRBG). La otra manera, que será explorada a continuación, es calcular determinísticamente los bits mediante un algoritmo; estos generadores determinísticos son conocidos como Deterministic Random Bit Generator (DRBG).

Un DRBG tiene un mecanismo que utiliza un algoritmo que produce una secuencia de bits partiendo de un valor inicial que es determinado por una semilla que, a su vez, está determinada por la salida de la fuente de aleatoriedad. Una vez que se tiene la semilla y se determina el valor inicial, el DRBG es instanciado y puede producir valores. Dado a su naturaleza determinística, se dice que los valores producidos por el DRBG son pseudoaleatorios y no aleatorios; si la semilla es mantenida oculta y el algoritmo fue bien diseñado, los bits de salida del DRBG serán impredecibles.

La entrada de ENTROPÍA es provista a un mecanismo DRBG para obtener una SEMILLA utillizando una fuente de aleatoriedad. La entrada de ENTROPÍA y la SEMILLA deben mantenerse secretas; que estos valores permanezcan secretos es una de las bases de la seguridad del DRBG. Otras entradas, como un NONCE o una cadena de personalización pueden ser utilizadas como entradas; estas pueden o no requerir ser mantenidas secretas también, y ser utilizadas para crear la SEMILLA inicial para el DRBG.

El estado interno es la memoria del DRBG y consiste en todos los valores que requiere el mecanismo (parámetros, variables, etcétera).

El mecanismo DRBG requiere cinco funciones; estas son explicadas con más detalle abajo:

1. Instanciación (*instantiate function*): obtiene la entrada de ENTROPÍA para crear una SEMILLA con la cual será creado un nuevo estado interno. La entrada puede ser combinada con una NONCE o una cadena de personalización.
2. Generación (*generate function*): genera bits pseudoaleatorios utilizando el estado interno actual; también tiene como salida un nuevo estado interno que es utilizado para el siguiente pedido.
3. Cambio de SEMILLA (*reseed function*): obtiene una nueva entrada de ENTROPÍA y la combina con el estado interno actual para crear una nueva SEMILLA y un nuevo estado interno.
4. Desinstanciación (*uninstantiate function*): elimina el estado interno actual.
5. Prueba de salud (*healt htest function*): determina que el mecanismo DRBG siga funcionando correctamente.

Cuando a un DRBG se le aplica la función de cambio de SEMILLA, es imperativo que la SEMILLA sea distinta a la que se utilizó en la función de instanciación. Cada SEMILLA define un nuevo periodo de SEMILLA (*seed period*) para la instanciación del DRBG. Una instanciación consiste en uno o más periodos de SEMILLA, estos comienzan cuando se obtiene una nueva SEMILLA y terminan cuando la siguiente SEMILLA es obtenida o el DRBG deja de utilizarse.

El estado interno deriva de la SEMILLA; este incluye el estado de trabajo (uno o más valores derivados de la SEMILLA que deben permanecer secretos, y la cuenta con el número de salidas que se han producido con esa semilla) y la información administrativa (el nivel de seguridad, etc). Es menester proteger el estado interno del DRBG. La implementación del mecanismo DRBG puede haber sido diseñado para tener múltiples instancias; en este caso, cada instancia debe tener su propio estado interno y el estado interno de una instancia DRBG jamás debe ser utilizado como estado interno para una instancia distinta. El estado interno no debe ser accesible a funciones distintas a las cinco del DRBG, ni a otras instancias del DRBG o a otros DRBG.

Los mecanismos especificados en [35] soportan cuatro niveles de seguridad: 112, 128, 192 y 256 bits. Este es uno de los parámetros que se necesitan para instanciar un DRBG; además, dependiendo de su

diseño, cada mecanismo DRBG tiene sus restricciones de nivel de seguridad. El nivel de seguridad depende de la implementación del DRBG y la cantidad de ENTROPÍA que se da como entrada a la función de instanciación.

Los bits pseudoaleatorios obtenidos mediante un DRBG no deben ser utilizados por una aplicación que requiera un nivel mayor de seguridad que con el que fue instanciado el DRBG. La concatenación de dos salidas del DRBG tampoco proveen un nivel de seguridad más alto que del que fueron instanciados (por ejemplo, dos cadenas concatenadas de 128 bits no dan como resultado una cadena de 256 bits con el nivel de seguridad de 256 bits).

Semillas

Las SEMILLAS deben ser obtenidas antes de generar bits pseudoaleatorios en el DRBG, pues esta es utilizada para instanciar al DRBG y determinar el estado inicial interno del mecanismo.

Cambiar la SEMILLA restaura el secreto de la salida del DRBG si el estado interno o la SEMILLA son conocidos. Hacer este cambio periódicamente es una buena manera de mantener a raya el peligro de que valores como la entrada de ENTROPÍA, la SEMILLA o el estado interno de trabajo; hayan sido comprometidos.

Los ingredientes para determinar una nueva SEMILLA para la función de instanciación son la entrada de entropía de una fuente aleatoria, un NONCE y una cadena de personalización (recomendada, pero no obligatoria). Para hacer un cambio de semilla, se necesita el estado interno actual, la entrada de ENTROPÍA y una entrada adicional opcional.

La longitud de la SEMILLA depende del mecanismo DRBG y el nivel de seguridad requerido; sin embargo, siempre debe ser de, al menos, el mismo número de bits de ENTROPÍA requerida.

La entrada de ENTROPÍA y la SEMILLA resultante deben de ser protegidas con el mismo cuidado con el que se protege la salida del DRBG; por ejemplo, si el mecanismo es utilizado para generar llaves, estos valores deben protegerse con la misma seguridad como son protegidas las llaves generadas. Además, la seguridad del DRBG depende de mantener en secreto la entrada de ENTROPÍA, por lo que esa entrada debe ser tratada como un parámetro crítico de seguridad (CRITICAL SECURITY PARAMETER (CSP)) y ser obtenido desde un módulo criptográfico que contenga la función necesaria o ser transmitido desde un canal seguro.

Cuando se requiera un NONCE para la construcción de la SEMILLA, esta debe cumplir con una de las siguientes dos condiciones: tener $nivel\_seguridad/2$ bits de ENTROPÍA o un valor que se espera no se repita más de lo que se repetiría una cadena aleatoria de $nivel\_seguridad/2$ bits. Aunque no debe ser mantenido en secreto, cada NONCE debe ser considerado como un CSP y debe ser único en el módulo criptográfico en donde se realiza la instanciación. El NONCE puede estar compuesto por uno o más de los siguientes valores:

1. Valor aleatorio generado para cada NONCE por un generador de bits aleatorios aprobado.
2. Una marca de tiempo con la resolución suficiente para que sea distinto cada vez que sea utilizado.
3. Un número de secuencia que se incremente constantemente.
4. Una combinación de una marca de tiempo y un número de secuencia que se incremente constantemente; tal que el número de secuencia regrese a su valor inicial solo cuando la marca de tiempo cambie.

Generar demasiadas salidas partiendo de una misma SEMILLA puede proveer suficiente información para ser capaz de predecir las salidas futuras; por lo que el cambio de SEMILLAS reduce riesgos de seguridad. Las SEMILLAS tienen una vida finita que depende el mecanismo DRBG utilizado. Es imperativo que las implementaciones respeten el límite de la vida de las SEMILLAS especificado para el mecanismo; y, cuando se alcance el límite de la vida de una SEMILLA, el DRBG no debe generar salidas hasta que se haya cambiado la SEMILLA o se cree una nueva instancia del DRBG (aunque se prefiere que se cambie la SEMILLA). Una SEMILLA jamás debe ser utilizada para inicializar o cambiar la semilla de otra instancia del DRBG o la suya.

La cadena de personalización (que es opcional pero recomendada) es utilizada para derivar la SEMILLA, puede ser obtenida dentro o fuera del módulo criptográfico y hasta puede ser una cadena vacía, pues el DRBG no depende de esta cadena para obtener ENTROPÍA. De hecho, que el adversario conozca la cadena de personalización no disminuye el nivel de seguridad de una instancia de DRBG siempre y cuando la entrada de ENTROPÍA se mantenga desconocida. Esta cadena no es considerada un CSP; puede introducir datos adicionales al DRBG, tales como identificadores de usuario, aplicación, versiones, protocolos, marcas de tiempo, direcciones de red, números de serie, etcétera.

Funciones

Un DRBG necesita cinco funciones para poder funcionar correctamente.

Instanciación Antes de generar bits pseudoaleatorios, el DRBG debe ser instanciado; esta función se encarga de revisar que los parámetros de entrada sean válidos, determina el nivel de seguridad para la instancia de DRBG que se generará, obtiene la entrada de ENTROPÍA capaz de soportar el nivel de seguridad y el NONCE (solo si es requerido), determina el estado interno inicial y, si se tienen varias instancias del DRBG simultáneas, obtiene un manejador de estado. Las entradas de la función son las siguientes:

1. Nivel de seguridad requerido.
2. Bandera que indica si se necesita o no la resistencia de predicción.
3. Cadena de personalización.
4. Entrada de ENTROPÍA.

```
entrada: nivel_seguridad_requerido, bandera_prediccion, cadena_personalizacion
         entrada_entropia, nonce
salida: estado, manejador_estado
inicio
  si nivel_seguridad_requerido > mayor_nivel_seguridad_soportado:
    regresar BANDERA_ERROR, invalido
  si bandera_prediccion =verdadero Y resistencia_prediccion no es soportado:
    regresar BANDERA_ERROR, invalido
  si longitud(cadena_personalizacion) > longitud_maxima:
    regresar BANDERA_ERROR, invalido
  Asignar a nivel_seguridad el nivel más bajo de seguridad mayor o igual
    a nivel_seguridad_requerido del conjunto {112, 128, 192, 256}
  (estado, entrada_entropia) = obtener_entropia(nivel_seguridad, ...
    ... longitud_min, longitud_max, bandera_prediccion)
  si (estado ≠ EXITOSO):
    regresar estado, invalido
  Obtener nonce
  estado_trabajo_inicial = INSTANCIAR_ALGORITMO(entrada_entropia, ...
    ... nonce, cadena_personalizacion, nivel_seguridad)
  Obtener manejador_estado para el estado interno vacío.
  si no se encuentra un estado interno vacío:
    regresar BANDERA_ERROR, invalido
  Configurar el estado interno de la nueva instancia con los valores iniciales
    y la información administrativa.
  regresar (EXITOSO, manejador_estado)
fin
```

Pseudocódigo 3.2: DRBG, instanciación.

5. NONCE

Las primeras entradas deben ser provistas por la aplicación consumidora. La salida de la función de instanciación a esta aplicación consiste en:

1. El estado del proceso; regresa un *EXITOSO* o un estado inválido indicando el error que hubo al instanciar al DRBG.
2. El manejador de estado, utilizado para identificar el estado interno de esta instancia para generar, cambiar la SEMILLA y demás funciones.

El algoritmo de la función de instanciación se puede observar en el pseudocódigo 3.2.

Cambio de semilla Cambiar la SEMILLA de una instancia no es requerido, pero es recomendado que se realice este proceso cada que sea posible. Cambiar la SEMILLA puede:

- Ser solicitado expresamente por la aplicación consumidora.

- Realizado cuando la aplicación consumidora requiere resistencia de predicción.
- Disparada por la función generadora cuando se alcanza un número predeterminado de salidas generadas.
- Disparada por eventos externos.

La función se encarga de revisar la validez de los parámetros de entrada, obtiene la entrada de ENTROPÍA de una fuente de aleatoriedad y, mediante el algoritmo de cambio de SEMILLA, combina el estado de trabajo actual con la nueva entrada de ENTROPÍA y valores adicionales para determinar el nuevo estado de trabajo actual. Las entradas para la función de cambio de SEMILLA son las siguientes:

1. El manejador de estado.
2. Bandera que indica si se requiere o no la resistencia de predicción.
3. Entradas adicionales (opcionales).
4. Entrada de ENTROPÍA.
5. Valores del estado interno e información administrativa.

Las primeras tres entradas deben ser provistas por la aplicación consumidora. La salida consiste en lo siguiente:

1. Estado que regresa la función.
2. Nuevo estado de trabajo interno.

El proceso para cambiar la SEMILLA se observa en el pseudocódigo 3.3.

Generación Esta función es utilizada para generar bits pseudoaleatorios después de haber utilizado la función de instanciación o de cambio de SEMILLA. Se encarga de validar los parámetros de entrada, llamar a la función de cambio de SEMILLA cuando sea requerida ENTROPÍA extra, generar los bits pseudoaleatorios, actualizar el estado de trabajo y regresar los bits a la aplicación consumidora que los pidió. La función tiene las siguientes entradas:

1. Manejador de estado.
2. El número de bits que se requieren.
3. El nivel de seguridad que se necesita.
4. Si debe ser resistente a predicción o no.
5. Entradas adicionales.
6. El estado de trabajo actual e información administrativa.

Después de haber sido generado, la salida consiste en lo siguiente

1. Estado.
2. Bits pseudoaleatorios.

```
entrada:  bandera_prediccion, manejador_estado, entrada_adicional
          entrada_entropia, estado_interno
salida:   estado, estado_trabajo_interno
inicio
  si manejador_estado indica un estado inválido o sin uso:
    regresar BANDERA_ERROR
  si se requiere resistencia de predicción y bandera_prediccion =falso:
    regresar BANDERA_ERROR
  si longitud(entrada_adicional) > longitud_max_entrada_adicional:
    regresar BANDERA_ERROR
  (estado, entrada_entropia) = obtener_entropia(nivel_seguridad, ...
    ... longitud_min, longitud_max, bandera_prediccion)
  si (estado ≠ EXITOSO):
    regresar estado
  estado_trabajo_nuevo = ALGORITMO_CAMBIAR_SEMILLA(estado_trabajo, ...
    ... entrada_entropia, entrada_adicional)
  Reemplazar el valor de estado_trabajo con estado_trabajo_nuevo
  regresar (EXITOSO)
fin
```

Pseudocódigo 3.3: DRBG, cambio de semilla.

3. Nuevo estado de trabajo.

El proceso para generar bits se puede observar en el pseudocódigo 3.4. Hay que tomar en cuenta cuando la implementación no tiene la capacidad de hacer el cambio de SEMILLA en el algoritmo: se quitan los pasos 6 y 7, y, si el estado indica que se necesita hacer el cambio, se regresa un error que indique que el DRBG no puede seguir siendo utilizado y hay que eliminar la instancia. También se debe tener en mente cuando se termina el ciclo de vida de la SEMILLA: cada que se llama al algoritmo generador, se revisa si el contador ha alcanzado el valor máximo que se encuentra en el estado interno; en caso de ser así, la función debe avisar que se requiere un cambio de SEMILLA.

Desinstanciación Esta función se encarga de liberar el estado interno de una instancia al borrar el contenido de su estado interno. La función requiere del manejador de estado de la instancia que se va a eliminar y regresa el estado de la función. El proceso para eliminar una instancia se puede observar en el pseudocódigo 3.5.

Mecanismos basados en funciones hash

Los mecanismos DRBG pueden estar basados en funciones hash de un solo sentido; dos mecanismos basados en estas funciones son los HASH_DRBG y HMAC_DRBG. El nivel de seguridad que puede soportar cada uno es el nivel de resistencia a la PREIMAGEN que tiene la función hash (véase sección 2.3).

```
1  entrada:  bandera_prediccion, manejador_estado, entrada_adicional
2            nivel_seguridad_requerido, no_bits_requeridos, estado_interno
3  salida:   estado, estado_trabajo, bits_pseudoaleatorios
4  inicio
5    si manejador_estado indica un estado inválido o sin uso:
6      regresar (BANDERA_ERROR, NULL)
7    si no_bits > no_bits_maximo
8      regresar (BANDERA_ERROR, NULL)
9    si nivel_seguridad_requerido > nivel_seguridad indicado en el estado interno:
10     regresar (BANDERA_ERROR, NULL)
11   si longitud(entrada_adicional) > longitud_max_entrada_adicional:
12     regresar (BANDERA_ERROR, NULL)
13   si se requiere resistencia de predicción y bandera_prediccion =falso:
14     regresar (BANDERA_ERROR, NULL)
15   Limpiar bandera_cambio_semilla_necesario
16   si bandera_cambio_semilla_necesario o bandera_prediccion están puestas:
17     estado = CAMBIO_SEMILLA(manejador_estado, prediccion_resistencia, entrada_adicional)
18     si estado ≠ EXITOSO:
19       regresar (estado, NULL)
20     Obtener el nuevo estado interno
21     entrada_adicional = NULL
22     Poner en cero bandera_cambio_semilla_necesario
23   (estado, bits_pseudoaleatorios, nuevo_estado_trabajo) = ALGORITMO_GENERAR(estado_trabajo, ...
24     ... no_bits_requeridos, entrada_adicional)
25   si estado indica que se requiere cambiar la semilla antes de poder generar bits:
26     Activar bandera_cambio_semilla_necesario
27     si se requiere resistencia de predicción:
28       Activar bandera_prediccion
29     Regresar al paso 7.
30   Reemplazar el estado_trabajo con nuevo_estado_trabajo.
31   regresar (EXITOSO, bits_pseudoaleatorios)
32 fin
```

Pseudocódigo 3.4: DRBG, generación.

```
1  entrada:  manejador_estado
2  salida:   estado
3  inicio
4    si manejador_estado indica un estado inválido:
5      regresar (BANDERA_ERROR)
6    Eliminar los contenidos del estado interno indicados por manejador_estado
7    regresar (EXITOSO)
8  fin
```

Pseudocódigo 3.5: DRBG, desinstanciación.

El mecanismo HASH_DRBG requiere el uso de una misma función hash en las funciones de instanciación, cambio de SEMILLA y generación. El nivel de seguridad de la función hash a utilizar debe igualar o ser mayor al nivel que se requiere por la aplicación consumidora del DRBG. En cambio, el mecanismo HMAC_DRBG utiliza múltiples ocurrencias de una función hash con llave; la misma función hash debe ser utilizada a lo largo del proceso de instanciación. Al igual que HASH_DRBG, la función hash debe tener, al menos, el mismo nivel de seguridad que la aplicación consumidora requiere para la salida del DRBG.

Mecanismos basados en cifradores por bloque

Un cifrado por bloques DRBG está basado en un algoritmo de cifrado por bloques. La seguridad que puede alcanzar el DRBG depende del cifrado por bloques y el tamaño de llave utilizado. Se tiene al mecanismo CTR_DRBG, que utiliza un cifrado por bloques aprobado con el modo de operación de contador (véase sección 2.2.6); debe utilizarse el mismo algoritmo de cifrado y tamaño de llave para todas las operaciones de cifrado por bloques en el DRBG. El CTR_DRBG tiene una función actualizadora que es llamada por los algoritmos de instanciación, generación y cambio de semilla para ajustar el estado interno cuando hay nueva ENTROPÍA, se le dan entradas adicionales o cuando se actualiza el estado interno después de generar bits pseudoaleatorios.

Garantías

Los usuarios de un DRBG requieren una garantía de que el generador en verdad produce bits pseudoaleatorios, que el diseño, implementación y uso de servicios criptográficos son adecuados para proteger la información del usuario y, finalmente, necesita una garantía de que el generador sigue operando correctamente. La implementación debe ser validada por un laboratorio acreditado por NATIONAL VOLUNTARY LABORATORY ACCREDITATION PROGRAM (NVLAP) para tener la certeza de que el mecanismo está bien implementado. Se requiere, además, para garantizar el funcionamiento del mecanismo, lo siguiente:

Documentación mínima se debe proveer, al menos, el siguiente conjunto de documentos; una gran parte

podrían pertenecer al manual de usuario:

1. Documentación del método para obtener la entrada de entropía.
2. Documentar cómo la implementación fue diseñada para permitir la validación de la implementación y revisar su estado.
3. Documentar el tipo de mecanismo DRBG y las primitivas criptográficas utilizadas.
4. Documentar los niveles de seguridad soportados por la implementación.
5. Documentar las características que soporta la implementación.
6. Si las funciones del DRBG están distribuidas, especificar los mecanismos que se usan para proteger la confidencialidad e integridad de las partes del estado interno que son trasnferidas entre las partes distribuidas del DRBG.
7. Indicar si se utiliza una función de derivación; si no se utiliza, documentar que la implementacion solo puede utilizarse cuando la entrada de ENTROPÍA completa está disponible.
8. Documentar todas las funciones de soporte.
9. Si se requieren hacer pruebas periódicas para la función generadora, documentar los intervalos y justificarlos.
10. Documentar si las funciones del DRBG pueden ser puestas a prueba sobre demanda.

Pruebas de validación de la implementación El mecanismo DRBG debe ser diseñado para ser probado y poder asegurar que el producto está correctamente implementado. Debe proveerse una interfaz para realizarle pruebas que permita insertar los datos de entrada y obtener los datos de salida. Todas las funciones que se incluyen en la implementación deben ser probadas en la funcionalidad de las pruebas de salud.

Pruebas de salud La implementación DRBG debe realizarse pruebas de salud a sí mismo para asegurarse de que continua operando correctamente. Se deben realizar pruebas del tipo *known answer*, donde se tiene una entrada para la que ya se sabe la respuesta correcta.

Manejo de errores Se indica para cada función del mecanismo qué errores son los esperados; es menester indicar el tipo de error que ocurrió.

3.4. Algoritmos tokenizadores

En esta sección se documentan distintos algoritmos para generar TOKENS; cada apartado incluye una descripción del algoritmo y su pseudocódigo. Antes de la documentación, se presentan dos clasificaciones de los algoritmos tokenizadores: la primera es la propuesta por el PAYMENT CARD INDUSTRY (PCI) SECURITY STANDARD COUNCIL (SSC) y mencionada en la sección **??**; la segunda clasificación es propuesta por los autores de este documento, tomando en cuenta nociones criptográficas más estrictas.

Clasificación del Payment Card Industry Security Standard Council

- Tokens reversibles:
  - Tokens criptográficos:
    1. BPS (véase sección 3.4.2).
    2. FFX (véase sección 3.4.1).
  - Tokens no criptográficos:
    1. AHR (véase sección 3.4.4).
    2. TKR (véase sección 3.4.3).
    3. Deterministic Random Bit Generator (DRBG)

Cabe resaltar que, de acuerdo con esta clasificación, los tokens generados mediante DRBG pueden quedar también en la clasificación de token no reversible, no autenticable. Se propone una nueva clasificación, pues esta no parece muy acertada; por ejemplo, del lado de los irreversibles se pueden generar tokens autenticables (o no autenticables) mediante funciones hash criptográficas, sin embargo, estos tokens no son *criptográficos*. Respecto a los reversibles, se toman por ejemplo, a los algoritmos TKR, que hace uso de primitivas criptográficas, y AHR, que hace uso de un cifrado por bloques y una función hash criptográfica; ninguno de los cuales queda en la categoría de los criptográficos, pues se guardan las relaciones Personal Account Number (PAN)-token en una base de datos.

Clasificación propuesta

- Criptográficos
  - Reversibles:
    1. BPS (véase sección 3.4.2).
    2. FFX (véase sección 3.4.1).
  - Irreversibles:
    1. AHR (véase sección 3.4.4).
    2. TKR (véase sección 3.4.3).
    3. DRBG.
- No criptográficos:
  1. Non-deterministic Random Bit Generator (NRBG).

La clasificación propuesta divide primeramente entre los que utilizan criptografía y los que no; se pone como ejemplo a los item NRBG como algoritmos no criptográficos, ya que su aleatoriedad depende de fenómenos físicos. Luego, dentro de los criptográficos, se divide a su vez entre algoritmos reversibles e irreversibles: los primeros, al igual que en el PCI SSC, son aquellos algoritmos que, dados el TOKEN y la llave, pueden regresar y obtener el PAN; los algoritmos pertenecientes a la segunda subcategoría (irreversible) son aquellos que necesitan guardar la relación PAN-TOKEN para poder obtener el PAN perteneciente a un TOKEN. Dado que, de acuerdo con el NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST), los DRBG utilizan funciones hash o cifradores por bloque, pertenecen también a la categoría de los criptográficos irreversibles.

3.4.1. Algoritmo FFX

FORMAT-PRESERVING FEISTEL-BASED ENCRYPTION (FFX) es un cifrador que preserva el formato presentado en [39] por Mihir Bellare, Phillip Rogaway y Terence Spies. En su forma más general, el algoritmo se compone de 9 parámetros que permiten cifrar cadenas de cualquier longitud en cualquier alfabeto; los autores también proponen dos formas más específicas (dos colecciones de parámetros) para alfabetos binarios y alfabetos decimales: A2 y A10, respectivamente.

FFX ocupa redes Feistel (ver sección 2.2.3) junto con PRIMITIVAS CRIPTOGRÁFICAS adaptadas (usadas como función de ronda) para lograr preservar el formato. La idea general para las posibles funciones de ronda es interpretar la salida binaria de una primitiva (p. ej. un cifrador por bloques, una función hash, un cifrador de flujo) de forma que tenga el formato deseado; esto es, que tenga la misma longitud y se encuentre en el mismo alfabeto que la entrada. Es importante notar que esta función no tiene por qué ser invertible: las redes Feistel ocupan la misma operación tanto para cifrar como para descifrar.

La operación general del algoritmo es la misma que la operación de una red Feistel. Para redes desbalanceadas:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= F_k(R_{i-1}) \oplus L_{i-1} \end{aligned} \tag{3.1}$$

Y para redes alternantes:

$$\begin{aligned} L_i &= \begin{cases} F_k^1(R_{i-1}) \oplus L_{i-1}, & \text{si } i \text{ es par} \\ L_{i-1}, & \text{si } i \text{ es impar} \end{cases} \\ R_i &= \begin{cases} R_{i-1}, & \text{si } i \text{ es par} \\ F_k^2(L_{i-1}) \oplus R_{i-1}, & \text{si } i \text{ es impar} \end{cases} \end{aligned} \tag{3.2}$$

Parámetro	**Valor**	**Comentario**
radix	10	alfabeto de dígitos decimales
longitudes	$[4, 36]$	rango de cadenas aceptadas
llaves	$\{0,1\}^{128}$	misma longitud que para Advanced Encryption Standard (AES)
tweaks	$\text{BYTE}^{\leq 2^{64}-1}$	cadena de longitud arbitraria
suma	por bloque	combinación por operaciones a nivel de bloque
método	alternante	redes Feistel alternantes
split	0	lo más cercano al centro posible
rondas	12, 18 o 24	depende de longitud de mensaje

Tabla 3.4: Colección de parámetros FFX A10.

Especificación de parámetros

A continuación se espifican los 9 parámetros de FFX.

1. ***Radix***. Número que determina el alfabeto usado. $C = \{0, 1, \ldots, \text{radix} - 1\}$. Tanto el texto en claro como el texto cifrado pertenecen a este alfabeto.

2. **Longitudes**. El rango permitido para longitudes de mensaje.

3. **Llaves**. El conjunto que representa al espacio de llaves.

4. ***Tweaks***. El conjunto que representa al espacio de *tweaks*.

5. **Suma**. El operador utilizado en la red Feistel para combinar la parte izquierda con la salida de la función de ronda.

6. **Método**. El tipo de red Feistel a ocupar: desbalanceada o alternante.

7. ***Split***. El grado de desbalanceo de la red Feistel.

8. **Rondas**. El número de rondas de la red Feistel.

9. **F**. La función de ronda. Recibe la llave, el *tweak*, el número de ronda y un mensaje; regresa una cadena del alfabeto de la longitud apropiada.

FFX A10

De las dos colecciones de parámetros presentadas en [40], la que se adapta mejor al dominio de los números de tarjetas de crétdito es A10, por lo que es la única que se presenta aquí. En la tabla 3.4 se muestran los valores con los que la colección FFX A10 inicializa FFX para crear un cifrador en el dominio decimal.

```
entrada: Arreglo número; llave k; tweak t; información adicional i
salida:  Arreglo salida
inicio
  entradaPrimitiva ← número || t || i
  mac ← CBC_MAC_AES(entradaPrimitiva, k)
  y′ ← mac[1 ... 64]
  y″ ← mac[65 ... 128]
  si m ≤ 9 entonces:
    z ← y″ mód 10^m
  sino:
    z ← (y′ mód 10^(m−9)) × 10^9 + (y″ mód 10^m)
  fin
  regresar z
fin
```

Pseudocódigo 3.6: *FFX A10*, función de ronda

La dependencia entre el número de rondas y la longitud del mensaje está dada por la siguiente relación (n es la longitud del mensaje):

$$\text{rondas} = \begin{cases} 12 & \text{si } 10 \leq n \leq 36 \\ 18 & \text{si } 6 \leq n \leq 9 \\ 24 & \text{si } 4 \leq n \leq 5 \end{cases} \tag{3.3}$$

En el pseudocódigo 3.6 se describe a la función de ronda. Esta concatena el *tweak*, el mensaje de entrada y los demás parámetros usados por la red en una sola cadena; esta cadena se cifra con AES Cipher-block Chaining (CBC) Message Authentication Code (MAC); la salida se parte en dos (mitad derecha y mitad izquierda) y es operada para que el número resultado tenga el mismo número de dígitos que la cadena de entrada. Esta última operación se describe con la sguiente ecuación ($y\prime$ es la parte izquierda y $y\prime\prime$ la derecha):

$$z = \begin{cases} y\prime\prime \mod 10^m, & \text{si } m \leq 9 \\ (y\prime \mod 10^{m-9}) \cdot 10^9 + (y\prime\prime \mod 10^m), & \text{en cualquier otro caso} \end{cases} \tag{3.4}$$

El valor de m corresponde al *split* en la ronda actual; esto es la longitud de la cadena de entrada.

3.4.2. Algoritmo *BPS*

La información que aquí se presenta puede ser consultada con mayor detalle en [41].

BPS es uno de los algoritmos de cifrado que preservan el formato existentes, y es capaz de cifrar cadenas de longitudes casi arbitrarias que estén formadas por cualquier conjunto de caracteres.

BPS está conformado por 2 partes fundamentales, un cifrado interno BC, encargado de cifrar bloques de longitud fija; y un modo de operación, encargado de extender la funcionalidad de el cifrador BC y permitir que *BPS* cifre cadenas de varias longitudes.

El cifrado interno BC

El cifrado por bloques que usa *BPS* internamente se define como

$$BC_{F,s,b,w}(X, K, T) \tag{3.5}$$

Donde:

- F es un cifrador por bloques de f bits de salida, por ejemplo: Triple Data Encryption Standard (DES) (TDES), Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA)-2.
- s es la cardinalidad del conjunto de caracteres del bloque a cifrar.
- b es la longitud del bloque a cifrar, cumpliendo con $b \leq 2 \cdot |log_s(2^{f-32})|$.
- w es el número (par) de rondas de la red Feistel interna (véase 2.2.3).
- X es la cadena o bloque de longitud b a cifrar.
- K es una llave acorde al cifrador por bloques F.
- T es un *tweak* de 64 bits.

Proceso de cifrado BC.

Para poder cifrar la cadena X:

1. Se tiene que dividir el *tweak* T de 64 bits en 2 sub*tweak*s T_L y T_R de 32 bits. Viendo a T como un número entero codificado en binario se puede calcular $T_R = T \mod 2^{32}$ y $T_L = (T - T_R)/2^{32}$
2. Igualmente, se tiene que dividir en 2 la cadena X para obtener las subcadenas X_L y X_R con una longitud l y r respectivamente. Dado que la longitud b de la cadena no siempre es par, se tiene que, si b es par, entonces tanto l como r son igual a $b/2$, pero en caso de que b sea impar, l va a ser igual a $(b+1)/2$ y r igual a $(b-1)/2$.

```
entrada:    bloque N_w de longitud n.
salida:     bloque Y_N
inicio
  para i = 0 hasta n − 1
    Y_N[i] = N_w mod s
    N_w = (N_w − Y_N[i])/s
fin
```

Pseudocódigo 3.7: Proceso de descomposición de L_w o R_w.

3. Partiendo de que el cifrador BC se compone de w rondas de una red Feistel, se define L_i y R_i (parte izquierda y parte derecha de la red en la i-ésima ronda), y se inicializan en:

$$L_0 = \sum_{j=0}^{l-1} X_L[j] \cdot s^j \tag{3.6}$$

$$R_0 = \sum_{j=0}^{r-1} X_R[j] \cdot s^j \tag{3.7}$$

4. Ahora por cada ronda $i < w$ y cifrando con el cifrador por bloques E.

   Si i es par:

$$L_{i+1} = L_i \boxplus E_K((T_R \oplus i) \cdot 2^{f-32} + R_i) \qquad (mod\ s^l) \tag{3.8}$$

$$R_{i+1} = R_i \tag{3.9}$$

   Si i es impar:

$$R_{i+1} = R_i \boxplus E_K((T_L \oplus i) \cdot 2^{f-32} + L_i) \qquad (mod\ s^r) \tag{3.10}$$

$$L_{i+1} = L_i \tag{3.11}$$

5. Por último se tiene que descomponer tanto a L_w como a R_w para obtener a Y_L y a Y_R respectivamente, las cuales concatenadas $(Y_L \parallel Y_R)$ dan la cadena de salida Y.

   El proceso para hacer la descomposición se muestra en el pseudocódigo 3.7.

   De forma general, el proceso de cifrado se describe en el pseudocódigo 3.8.

   Proceso de descifrado BC^{-1}.

   Para poder descifrar la cadena Y:

1. Se tiene que dividir en 2 la cadena Y, para obtener las subcadenas Y_L y Y_R con una longitud l y r respectivamente, de igual forma que se hizo con la cadena X en el proceso de cifrado.

entrada: la llave K,
la cadena X de longitud b formada por el conjunto S de cardinalidad s,
el tweak T,
la función de cifrado F, **y** el número de rondas w.
salida: La cadena cifrada Y.
inicio
  calcular $T_R = T \mod 2^{32}$ **y** $T_L = (T - T_R)/2^{32}$
  asignar $l = r = b/2$
  inicializar $L_0 = \sum_{j=0}^{l-1} X[j] \cdot s^j$
  inicializar $R_0 = \sum_{j=0}^{r-1} X[j+l] \cdot s^j$
  para $i = 0$ hasta $i = w - 1$
    si i es par
      $L_{i+1} = L_i \boxplus F_K((T_R \oplus i) \cdot 2^{f-32} + R_i) \quad (mod\ s^l)$
      $R_{i+1} = R_i$
    si i es impar
      $R_{i+1} = R_i \boxplus F_K((T_L \oplus i) \cdot 2^{f-32} + L_i) \quad (mod\ s^r)$
      $L_{i+1} = L_i$
  para $i = 0$ hasta $i = l - 1$
    $Y_L[i] = L_w \ mod\ s$
    $L_w = (L_w - Y_L[i])/s$
  para $i = l$ hasta $i = r - 1$
    $Y_R[i] = R_w \ mod\ s$
    $R_w = (R_w - Y_R[i])/s$
  determinar $Y = Y_L \parallel Y_R$
fin

Pseudocódigo 3.8: Proceso de cifrado BC.

2. Partiendo de que el proceso de descifrado se compone de w rondas, se define L_i y R_i y se inicializan en:

$$L_w = \sum_{j=0}^{l-1} Y_L[j] \cdot s^j \tag{3.12}$$

$$R_w = \sum_{j=0}^{r-1} Y_R[j] \cdot s^j \tag{3.13}$$

3. Ahora, comenzando con $i = w - 1$, para cada ronda $i \geq 0$.

   Si i es par:

$$L_i = L_{i+1} \boxminus E_K((T_R \oplus i) \cdot 2^{f-32} + R_{i+1}) \qquad (mod\ s^l) \tag{3.14}$$

$$R_i = R_{i+1} \tag{3.15}$$

   Si i es impar:

$$R_i = R_{i+1} \boxminus E_K((T_L \oplus i) \cdot 2^{f-32} + L_{i+1}) \qquad (mod\ s^r) \tag{3.16}$$

$$L_i = L_{i+1} \tag{3.17}$$

4. Finalmente se tienen que descomponer L_0 y R_0 (con el mismo proceso de descomposición descrito en el cifrado) para obtener a X_L y X_R, las cuales concatenadas ($X_L \parallel X_R$) dan la cadena de salida X.

   De forma general, el proceso de descifrado se describe en el pseudocódigo 3.9.

El modo de operación

En cuanto al modo de operación de *BPS*, se puede decir que es un equivalente al modo de operación CBC (véase 2.2.6), ya que el bloque BC_n utiliza el texto cifrado de la salida del bloque BC_{n-1}, con la distinción de que en lugar de aplicar operaciones *xor* usa sumas modulares carácter por carácter, y de que no utiliza un VECTOR DE INICIALIZACIÓN, a pesar de soportar su uso.

Algo importante a resaltar de este modo de operación es que, en caso de que el texto en claro no tenga una longitud total que sea múltiplo de la longitud de bloque b, al cifrar el último bloque se recorre el cursor que determina el inicio del mismo, hasta que su longitud concuerde con b, esto se puede ver de forma gráfica en la figura 3.3.

En la figura 3.4 se ve de manera gráfica el funcionamiento del modo de operación de *BPS*.

Otra particularidad del modo de operación es el uso del contador u de 16 bits, que es utilizado para aplicar una operación *xor* al *tweak* T que entra a cada uno de los bloques BC. Recordando que T es de 64 bits, el *xor* se aplica a los 16 bits más significativos de ambas mitades de *tweak*, esto debido a cada mitad

entrada: la llave K,
la tweak T,
la cadena Y de longitud b formada por el conjunto S
de cardinalidad s,
la función de cifrado F,
el número de rondas w.
salida: La cadena X.
inicio
calcular $T_R = T \mod 2^{32}$ **y** $T_L = (T - T_R)/2^{32}$
asignar $l = r = b/2$
inicializar $L_w = \sum_{j=0}^{l-1} Y[j] \cdot s^j$
inicializar $R_w = \sum_{j=0}^{r-1} Y[j+l] \cdot s^j$
para $i = w-1$ hasta $i = 0$
si i es par:
$L_i = L_{i+1} \boxminus F_K((T_R \oplus i) \cdot 2^{f-32} + R_{i+1}) \quad (mod\ s^l)$
$R_i = R_{i+1}$
si i es impar:
$R_i = R_{i+1} \boxminus F_K((T_L \oplus i) \cdot 2^{f-32} + L_{i+1}) \quad (mod\ s^r)$
$L_i = L_{i+1}$
para $i = 0$ hasta $i = l-1$
$X_L[i] = L_w \ mod\ s$
$L_w = (L_w - X_L[i])/s$
para $i = l$ hasta $i = r-1$
$X_R[i] = R_w \ mod\ s$
$R_w = (R_w - X_R[i])/s$
determinar $X = X_L \parallel X_R$
fin

Pseudocódigo 3.9: Proceso de descifrado BC^{-1}.

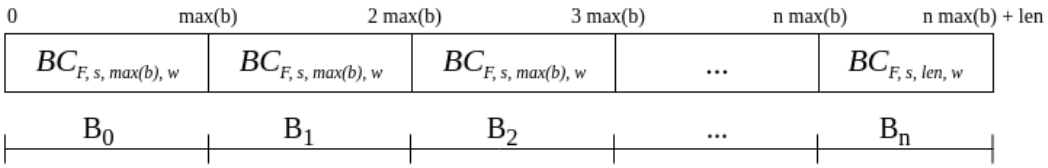


Figura 3.3: Corrimiento de cursor para la selección del último bloque en el modo de operación de *BPS*.

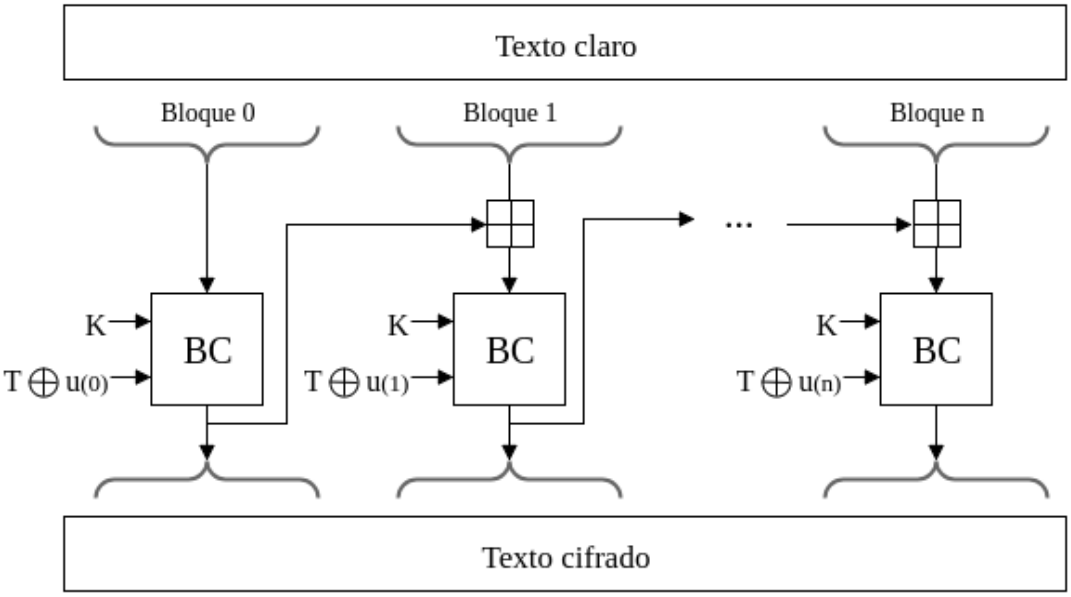


Figura 3.4: Modo de operación de *BPS*.

de *tweak* funciona de manera independiente en el cifrador BC, y a que no se desea un traslape entre el contador externo u y el contador i interno en BC.

Características generales

Como se observó, *BPS* está basado en las redes Feistel, lo cual puede verse como una ventaja, debido al amplio estudio que tienen. Además, usa algoritmos de cifrado o funciones hash estandarizadas de forma interna, lo cual hace más comprensible y fácil su implementación.

BPS es un cifrado que preserva el formato capaz de cifrar cadenas de un longitud de 2 hasta $max(b) \cdot 2^b$ caracteres (donde $max(b)$ es el tamaño máximo de bloque), formadas por cualquier conjunto.

Se puede considerar que *BPS* es eficiente, debido a que la llave K usada en cada bloque BC es constante, y a que además usa un número reducido de operaciones internas en comparación con otros algoritmos de cifrado que preservan el formato.

Por último, se puede resaltar que el uso de *tweaks* protege a *BPS* de ataques de diccionario, los cuales son fáciles de cometer cuando el dominio de la cadena a cifrar es muy pequeño.

Recomendaciones

Se recomienda que el número de rondas w de la red Feistel sea 8, dado que es una número de rondas eficiente, y se ha estudiado la seguridad de *BPS* con este w.

```
1  entrada: PAN p; información asociada d; llave k
2  salida:  token
3  inicio
4    S1 ← buscarPAN(p)
5    S2 ← buscarInfoAsociada(d)
6    si S1 y S2 = 0:
7      t ← RN(k)
8      insertar(t, p, d)
9    sino:
10     t ← S1
11   fin
12   regresar t
13 fin
```

Pseudocódigo 3.10: *TKR2*, método de tokenización

Es recomendable que como *tweak* se use la salida truncada de una función hash, en donde la entrada de la función puede ser cualquier información relacionada a los datos que se deseen proteger, como por ejemplo fechas, lugares, o parte de los datos que no se deseen cifrar.

3.4.3. Algoritmo TKR2

En [42] se analiza formalmente el problema de la generación de TOKENS y se propone un algoritmo que no está basado en FORMAT PRESERVING ENCRYPTION (FPE). Hasta antes de la publicación de este documento, los únicos métodos para generar TOKENS cuya seguridad estaba formalmente demostrada eran los basados en FPE.

El algoritmo propuesto usa PRIMITIVAS CRIPTOGRÁFICAS para generar TOKENS aleatorios y almacena en una base de datos (CARD DATA VAULT (CDV)) la relación original de estos con los PERSONAL ACCOUNT NUMBER (PAN). Según la clasificación del PAYMENT CARD INDUSTRY (PCI) se trata de una algoritmo tokenizador reversible *no criptográfico*; sin embargo tratarlo de esta forma (como *no criptográfico*) resulta un tanto confuso, dado que toda la contrucción del algoritmo y las pruebas de su seguridad se basan en fundamentos de la criptografía.

En el pseudocódigo 3.10 se muestra el proceso de tokenización, mientras que en 3.11 está la detokenización.

La mayor parte del proceso de tokenización y toda la detokenización son bastante fáciles de comprender; lo único que queda por esclarecer es la función generadora de TOKENS aleatorios RN_k. Idealmente, esta función debe regresar un elemento uniformemente aleatorio del espacio de TOKENS. La propuesta que se hace en [42] para instanciar esta función se presenta en el pseudocódigo 3.12. Aquí, la variable *contador*

```
1  entrada: token t; información asociada d; llave k
2  salida: PAN
3  inicio
4    S1 ← buscarToken(t)
5    S2 ← buscarInfoAsociada(d)
6    si S1 y S2 = 0:
7      regresar error
8    sino:
9      p ← S1
10   fin
11   regresar p
12 fin
```

Pseudocódigo 3.11: *TKR2*, método de detokenización

mantiene un estado del algoritmo (mantiene su valor a lo largo de las distintas llamadas); el espacio de TOKENS contiene cadenas de longitud fija μ de un alfabeto AL cuya cardinalidad es l; el número de bits necesarios para enumerar a todo el alfabeto se guardan en $\lambda = \lceil \log_2 l \rceil$.

En el pseudocódigo 3.12 lo primero que se hace es utilizar una PRIMITIVA CRIPTOGRÁFICA f para generar una cadena binaria (hablaremos sobre este punto más adelante); esta cadena (nombrada X) se parte en subcadenas de λ bits; después se itera de manera consecutiva sobre estas subcadenas binarias, si la representación entera de la i-ésima está en el rango del alfabeto de los TOKENS, entonces se concatena al token resultado, sino, se pasa a la siguiente subcadena. La longitud de la cadena regresada por f debe ser, aproximadamente, $3\mu\lambda$: dado que se espera que el comportamiento de f sea EQUIPROBABLE, entonces el ciclo correrá un promedio de 2μ veces.

Existen varios candidatos viables para f: un cifrado de flujo (sección **??**), pues el flujo de llave de estos produce cadenas de aspecto aleatorio; un cifrado por bloques (sección 2.2), utilizando un modo de operación de contador (sección **??**); un TRUE RANDOM NUMBER GENERATOR (TRNG) para obtener secuencias de bits verdaderamente aleatorias.

Por último hay que aclarar que el algoritmo presentado en el pseudocódigo 3.10 debe recibir un par de modificaciones más: al momento de generar un TOKEN debe existir una validación que verifique que este sea único (para evitar que dos PAN tengan un mismo TOKEN); la base de datos debe estar cifrada, por lo que, antes de hacer inserciones y después de hacer consultas, deben existir las operaciones correspondientes.

3.4.4. Algoritmo Híbrido Reversible

Longo, Aragona y Sala [43], propusieron en 2017, un algoritmo de tipo híbrido reversible. Este está basado en un cifrado de bloques con una llave secreta y una entrada adicional.

```
1  entrada: llave k
2  salida:  token
3  inicio
4    X ← f(k, contador)
5    X_1, X_2, ..., X_m ← cortar(X, λ)
6    t ← ""
7    i ← 0
8    mientras |t| ≠ μ:
9      si entero(X_i) ≤ l:
10       t ← t + entero(X_i)
11     fin
12     i ← i + 1
13   fin
14   contador ← contador + 1
15   regresar t
16 fin
```

Pseudocódigo 3.12: *TKR2*, generación de TOKENS aleatorios

Se sabe que el número de una tarjeta (PERSONAL ACCOUNT NUMBER (PAN)) está conformado por tres partes concatenadas: el número que identifica al emisor de la tarjeta, el que identifica la cuenta y un número de verificación. En este algoritmo se reemplaza la primera parte con un TOKEN BANK IDENTIFIER NUMBER (BIN) y se cifra solo la parte que identifica la cuenta. Al final, se calcula un nuevo dígito de verificación.

Las entradas del algoritmo son la parte del PAN a cifrar y una entrada adicional. Esta última actúa como un *tweak* (véase sección 2.5), pues permite que se generen varios TOKENS para el mismo PAN.

El algoritmo necesita una función f pública que, dada una cadena de longitud m regrese una de longitud n (véase sección de funciones hash 2.3). Se toman solo cifrados cuyo tamaño de bloque sea de mínimo 128 bits. La función f se encarga de poner el relleno en la entrada para completar el bloque del cifrado y permitir la creación de varios TOKENS para el mismo PAN utilizando la misma llave en el proceso de cifrado. Finalmente, el algoritmo necesita una base de datos segura que se encargará de contener los pares PAN-TOKEN. Al momento de crear los TOKENS, se necesita acceder a la base de datos mediante una FUNCIÓN BOOLEANA *comprobar* que revisa si el TOKEN generado ya está almacenado en la base.

Como se desea obtener un TOKEN que tenga el mismo número de dígitos que el PAN (longitud l) ingresado, se deben tomar en cuenta solo una fracción de las posibles salidas del cifrado E; para resolver este problema, se utiliza un método conocido como el CIFRADO DE CAMINATA CÍCLICA.

```
1  entrada: PAN p; entrada_adicional u; llave k
2  salida:  token
3  inicio
4    t = f(u,p)|||[p̄]^s_b  (paso 1)
5    c = E(k,t)  (paso 2)
6    si  (c̄ mód 2^n) ≥ 10^l
7      t = c
8      Regresar al paso 2.
9    fin
10   token = [c̄ mód 2^n]^l_10
11   si comprobar(token) = verdadero
12     u = u + 1
13     Regresar al paso 1.
14   fin
15   regresar token
16 fin
```

Pseudocódigo 3.13: Híbrido reversible, método de tokenización

Notación

A continuación se definen una serie de notaciones que se utilizarán en el algoritmo:

- M Tamaño de bloque del cifrador por bloques que se usará.
- l Longitud de la entrada. En este caso, $13 \geq l \geq 19$.
- n Número de bits necesarios para representar a la entrada: $n = log_2(10^l)$.
- $[y]^s_b$ Indica que y es menor que b^s: $y < b^s$.
- $\bar{x}$ Representación de x en una cadena binaria cuando x es representado en su forma decimal y viceversa.

El pseudocódigo del algoritmo de tokenización se puede observar en **??**. ,

3.4.5. Tokenización mediante generador de números pseudoaleatorios

En la sección 3.3.1 se habló sobre el estándar del National Institute of Standards and Technology (NIST) para la generación de números aleatorios. En esta mostramos cómo estos métodos pueden ser utilizadas para tokenizar y detokenizar números de tarjetas. Primero se muestran dos posibles instanciaciones con primitivas criptográficas del método general descrito en 3.3.1: una basada en funciones hash (sección 2.3) y otra basada en cifradores por bloque (sección 2.2); ambas presentadas en [35].

```
1  entrada: número n de bytes deseados.
2  salida:  arreglo de n bytes.
3  inicio
4    longitud ← tamanioDeHash()
5    numeroDeBloques ← ⌈longitud/n⌉
6    datos ← semilla
7    para_todo i en numeroDeBloques:
8      resultado ||= hash(datos)
9      datos += 1
10   fin
11   regresar n bytes de resultado
12 fin
```

Pseudocódigo 3.14: Generación de bits pseudoaleatorios mediante función hash

DRBG basado en funciones hash

El método genérico define un solo valor crítico (un valor que se debe mantener en secreto, pues la seguridad en generador depende de esto): la semilla, tanto al inicio, como su valor a lo largo de su vida útil. Este método introduce un segundo valor crítico: una constante C cuyo valor depende también de la entrada de entropía. Por tanto, las funciones de instanciación, cambio de semilla y desinstanciación se deben modificar para incluir a este segundo valor.

En el pseudocódigo 3.14 se muestra la función de generación de bits pseudoaleatorios basada en una función hash. *tamanioDeHash* regresa el tamaño de los *digest* retornados por la función hash usada.

DRBG basado en cifrador por bloque

Al igual que el generador basado en funciones hash, este método también introduce un segundo valor crítico: el valor de la llave utilizada por el cifrador por bloque subyacente. Esta llave se genera a partir de la entrada de entropía y debe ser tratada de igual forma que la semilla por las funciones de instanciación, cambio de semilla y desinstanciación.

En el pseudocódigo 3.15 se muestra la función de generación de bits psudoaleatorios basada en un cifrador por bloques.

Uso como algoritmo tokenizador

Un Deterministic Random Bit Generator (DRBG) puede ser utilizado de manera bastante similar a la función RN del algoritmo TKR (sección 3.4.3): en lugar de utlizar la función f como método para obtener bits pseudoaleatorios, se ocupa el generador pseudoaleatorio. El pseudocódigo para esto se muestra

```
entrada: número n de bytes deseados.
salida:  arreglo de n bytes.
inicio
  mientras longitud(resultado) ≤ n:
    semilla += 1
    resultado ||= cifrar(semilla)
  fin
  regresar n bytes de resultado
fin
```

Pseudocódigo 3.15: Generación de bits pseudoaleatorios mediante cifrador por bloques

en 3.16; la función *drbg* es una llamada a la operación de generación de bits pseudoaleatorios de cualquiera de los dos generadores presentados anteriormente.

Al igual que con TKR, ambas operaciones (tokenización y detokenización) deben tener el soporte de una base de datos: al generar un nuevo TOKEN este se guarda en la base de datos, y la operación de detokenización es solamente una consulta en la base.

```
1  entrada: llave k
2  salida:  token
3  inicio
4    X ← drbg()
5    X_1, X_2, ..., X_m ← cortar(X, λ)
6    t ← ""
7    i ← 0
8    mientras |t| ≠ μ:
9      si entero(X_i) ≤ l:
10       t ← t + entero(X_i)
11     fin
12     i ← i + 1
13   fin
14   contador ← contador + 1
15   regresar t
16 fin
```

Pseudocódigo 3.16: Generación de *tokens* mediante DRBG

Capítulo 4

Análisis y diseño

«A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.»
Douglas Adams.

La metodología ocupada en el proyecto es una mezcla entre prototipos y Security Development Lifecycle (SDL). En este capítulo se abordan los procesos de análisis y diseño de los tres prototipos principales del proyecto: el módulo de algoritmos generadores de tokens, el servicio web, que sirve como interfaz al primer módulo, y la tienda en línea, que hace uso de la interfaz web anterior.

4.1. Generación de *tokens*

4.1.1. Requerimientos

El Payment Card Industry (PCI) Data Security Standard (DSS) define cuatro *dominios* de seguridad para el proceso de tokenización:

1. **Generación de tokens.** Para cada clase tokenizadora, este dominio se encarga de definir consideraciones para generación segura de tokens. Cubre los dispositivos, procesos, mecanismos y algoritmos que son utilizados para crear los tokens.

2. **Mapeo de tokens.** Este dominio, que se refiere al mapeo de los tokens con su Personal Account Number (PAN) origen, aplica solamente a los procesos de tokenización reversibles. Entre otras cosas, provee guías respecto a control de acceso necesarios para las peticiones de tokenización.

3. **Bóveda de datos de tarjeta.** Como el dominio pasado, solo aplica a las implementaciones de tokenización reversibles. Cubre el cifrado obligado del PAN y los controles de acceso necesarios para entrar a la Card Data Vault (CDV).

4. **Manejo criptográfico de llaves.** Define las buenas prácticas para el manejo criptográfico de las llaves y las operaciones realizadas con ellas por el producto tokenizador.

En esta sección se detallan los requerimientos que deben de satisfacer los tokens. Para comenzar, se enlistan los requerimientos aplicables a todos, sin importar su categoría; después de esto, se agrupan los requerimientos correspondientes a cada una de las categorías presentadas en la sección 3.2.

REQPCI-01 Validación de productos de hardware.
Si se usa un producto de hardware para la tokenización, este debe de ser validado por Federal Information Processing Standard (FIPS) 140-2 nivel 3 o superior (descrito en [44]).

REQPCI-02 Validación de productos de software.
Si se usa un producto de software para la tokenización, este debe de ser validado por FIPS 140-2 nivel 2 o superior (descrito en [44]).

REQPCI-03 Resistencia a texto claro conocido.
Un atacante con acceso a múltiples pares de tokens y PAN no debe de ser capaz de determinar otros PAN a partir de solamente tokens. En otras palabras, los tokens deben ser resistentes a ataques

con texto en claro conocido (sección 2.1.1).

REQPCI-04 Resistencia a sólo texto cifrado.

Recuperar un PAN a partir de un TOKEN debe de ser COMPUTACIONALMENTE NO FACTIBLE (resistencia a ataques con sólo texto cifrado, sección 2.1.1).

REQPCI-05 Detección de anomalías.

Se deben de implementar disparadores que permitan detectar irregularidades en el sistema (anomalías, funcionamientos erróneos, comportamientos sospechosos). El producto debe registrar dichos eventos y avisar al personal correspondiente.

REQPCI-06 Distinción entre tokens y PAN.

Se debe contar con un mecanismo para distinguir entre TOKENS y PAN. Los proveedores del servicio de tokenización deben compartir este mecanismo con la entidad (o entidades) que usa los TOKENS.

REQPCI-07 Guía de instalación.

Se debe de contar con una guía de instalación y uso para el correcto funcionamiento del producto de tokenización.

REQPCI-08 Integridad del proceso de tokenización.

Deben de implementarse mecanismos que garanticen la integridad del proceso de generación de TOKENS.

REQPCI-09 Acceso al proceso detokenización.

Solo los usuarios autenticados y componentes del sistema tienen permitido acceder al proceso de tokenización y detokenización. Los métodos utilizados deben ser al menos tan rigurosos como lo indicado en el requerimiento 8 del PCI DSS [34].

SUBREQPCI-09/1 Control de peticiones.

Todas las peticiones deben pasar a través de una APPLICATION PROGRAM INTERFACE (API) que controle todos los intentos de acceso y aplique de manera uniforma reglas de control de acceso.

SUBREQPCI-09/2 Registros de acceso.

Se deben registrar todos los eventos de acceso, de tokenización y de detokenización. Esta funcionalidad debe ser configurable de manera segura. Para esto se debe seguir el requerimiento 4 del PAYMENT APPLICATION (PA)-DSS [45].

SUBREQPCI-09/3 Autenticación multifactor.

El sistema debe soportar AUTENTICACIÓN MULTIFACTOR para todos los tipos de usuario: accesos administrativos, operaciones de tokenización y detokenización, mantenimiento, etcétera.

SUBREQPCI-09/4 Accesos a nivel de sistema.

Todos los accesos a nivel de sistema deben soportar AUTENTICACIÓN MUTUA, incluyendo a las peticiones de tokenización y detokenización.

SUBREQPCI-09/5 Accesos administrativos.

Se debe utilizar CRIPTOGRAFÍA FUERTE para todos los accesos administrativos que no se hagan desde consola.

REQPCI-10 Mapeos de token a token prohibidos.

No se debe poder pasar de un primer TOKEN válido a un segundo, también válido; forzosamente debe existir un estado intermedio: del primer TOKEN se pasa al PAN correspondiente (operación de detokenización) y de este se pasa al segundo TOKEN.

REQPCI-11 Protección contra vulnerabilidades comunes.

Se deben implementar medidas en contra de las vulnerabilidades de seguridad más comunes ([45], requerimiento 5.2). Algunas de estas medidas pueden ser el uso de herramientas de análisis de código estático, o el uso de lenguajes de programación especializados.

REQPCI-12 Primitivas criptográficas usadas.

Las primitivas criptográficas que se usen deben estar basadas en estándares nacionales (referentes a Estados Unidos) o internacionales (p. ej. ADVANCED ENCRYPTION STANDARD (AES)). Ver sección 3.2.1.

REQPCI-13 Sobre el manejo adecuado de llaves.

En donde se usen llaves para la generación y protección de TOKEN, se deben seguir buenas prácticas criptográficas para la administración de estas. En particular, se deben cumplir con las recomendaciones del NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST) en [36] y [37].

SUBREQPCI-13/1 Sobre el ciclo de vida.

La llave tokenizadora debe seguir la política de los ciclos de llaves descritos en el ISO/IEC 115681 (ver sección A).

SUBREQPCI-13/2 Descripción del periodo criptográfico activo.

La política sobre el tiempo de vida de la llave debe incluir una descripción sobre el periodo criptográfico activo de la llave tokenizadora en cuestión.

SUBREQPCI-13/3 Sobre la destrucción de las llaves.

El proveedor debe incorporar una función que permita la destrucción de sus llaves criptográficas sin tener que alterar o abrir el dispositivo.

SUBREQPCI-13/4 Exportar llave en claro prohibido.

Las llaves usadas para generar TOKENS no se deben poder exportar en claro desde el programa.

SUBREQPCI-13/5 Entropía de generación de llaves.

La fuente generadora de llaves debe tener, al menos, 128 bits de ENTROPÍA.

SUBREQPCI-13/6 Llaves de uso único.

Las llaves criptográficas usadas para generar TOKENS no deben ser usadas para ningún otro fin.

Irreversibles

REQPCI-14 Sobre la generación de tokens (irreversibles).

El mecanismo utilizado para la generación de los TOKENS no es reversible (o improbable).

SUBREQPCI-14/1 Sobre el mecanismo generador (irreversibles).

El proceso para crear TOKENS clasificado como irreversible debe asegurar que el mecanismo, proceso o algoritmo utilizado para crear el TOKEN no sea reversible. Si una función hash (véase sección 2.3) es utilizada, esta debe ser una PRIMITIVA CRIPTOGRÁFICA y utilizar una llave secreta k tal que el mero conocimiento de la función hash no permita la creación de un ORÁCULO.

SUBREQPCI-14/2 Contenido en claro (irreversibles).

Los TOKENS irreversibles no deben contener dígitos en claro del PAN original, excepto que estos dígitos sean una coincidencia.

SUBREQPCI-14/3 Creación de un diccionario (irreversibles).

La creación de una tabla o *diccionario* de TOKENS estáticos debería ser imposible, o, al menos, al punto de satisfacer que la probabilidad de predecir correctamente el PAN sea menor que $\frac{1}{10^6}$.

SUBREQPCI-14/4 Sobre el proceso de autenticación (irreversibles).

En el caso de los TOKENS autenticables, el proceso de autenticación no debe revelar información suficiente para realizar búsquedas, excepto una exhaustiva (PAN por PAN) y se deben implementar controles para detectar estas últimas.

Criptográficos reversibles

REQPCI-15 Probabilidad de adivinar relaciones (criptográficos).

La probabilidad de adivinar la relación entre un TOKEN y un PAN debe de ser menor que 1 en 10^6.

SUBREQPCI-15/1 Distribución uniforme (criptográficos).

Para un PAN dado, todos los TOKENS deben ser equiprobables; esto es, el mecanismo tokenizador no debe exhibir tendencias probabilísticas que lo expongan a ataques estadísticos.

SUBREQPCI-15/2 Permutación aleatoria (criptográficos).
El método de tokenización debe actuar como una familia de PERMUTACIONES aleatoria desde el espacio de PAN al espacio de TOKENS.

SUBREQPCI-15/3 Cambio de llave (criptográficos).
Un cambio en la llave se debe ver reflejado en un cambio en el TOKEN resultado.

SUBREQPCI-15/4 Cambio de PAN (criptográficos).
Un cambio en el PAN se debe ver reflejado en un cambio en el TOKEN resultado.

SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos).
Se debe tener un medio para verificar de forma práctica la aleatorización de dígitos, de acuerdo a lo establecido en NIST 800-90A [35].

REQPCI-16 Almacenamiento de tokens (criptográficos).
Los TOKENS generados no se deben almacenar en ningún punto del sistema.

El requerimiento anterior (REQPCI-16 ALMACENAMIENTO DE TOKENS (CRIPTOGRÁFICOS), RC1C en [33]) es un tanto difícil de interpretar; la versión original establece: *los* TOKENS *basados en el* PAN *completo no se deben almacenar si el producto tokenizador también almacena su* PAN *truncado correspondiente*. Es un requerimiento de los criptográficos reversibles, por lo que el TOKEN no se debería almacenar bajo ninguna circunstancia (según la propia clasificación del PCI SECURITY STANDARD COUNCIL (SSC)); la redacción del requerimiento se cambió para reflejar este hecho.

REQPCI-17 Seguridad de la administración de llaves (criptográficos).
Todas la operaciones sobre la administración de las llaves criptográficas deben realizarse en un dispositivo criptográfico seguro y aprobado: el PCI SSC se encarga de hacer validaciones; también puede ser cualquier dispositivo validado por FIPS 140-2 nivel 3 o superior [44] o por la INTERNATIONAL ORGANIZATION FOR STANDARIDIZATION (ISO) 13491-1.

REQPCI-18 Sobre la longitud de las llaves (criptográficos).
Las llaves para *tokenizar* deben tener una FUERZA EFECTIVA de, al menos, 128 bits. Cualquier llave utilizada para proteger o para derivar la llave del TOKEN debe de ser de igual o mayor FUERZA EFECTIVA.

REQPCI-19 Independencia estadística (criptográficos).
Si el espacio de llaves es usado para producir TOKENS es dos contextos distintos (p. ej. para distintos comerciantes), estas deben ser ESTADÍSTICAMENTE INDEPENDIENTES.

No criptográficos reversibles

REQPCI-20 Generación y almacenamiento de tokens (no criptográficos).

La generación de un TOKEN debe realizarse independientemente de su PAN, y la relación entre un PAN y su TOKEN sólo tiene que estar almacenada en la base de datos (CDV) establecida.

REQPCI-21 Probabilidad de encontrar un PAN (no criptográficos).

La probabilidad de encontrar un PAN a partir de su respectivo TOKEN debe de ser menor que 1 en 10^6.

SUBREQPCI-21/1 Distribución equiprobable (no criptográficos).

Para un PAN dado, todos sus TOKENS respectivos deben ser EQUIPROBABLES, esto es que el sistema *tokenizador* no debe exhibir patrones probabilísticos que lo vulneren a un ataque estadístico.

SUBREQPCI-21/2 Permutaciones aleatorias (no criptográficos).

El método de tokenización debe actuar como una familia de PERMUTACIONES aleatoria en el espacio efectivo de los PANs al espacio de TOKENS.

SUBREQPCI-21/3 Parámetros de tokenización (no criptográficos).

El método de tokenización debe incluir parámetros tales que, un cambio en estos parámetros resulte en un TOKEN diferente; por ejemplo, un cambio en la instancia del proceso debe derivar en una secuencia de TOKENS distintos, incluso cuando es usada la misma secuencia de TOKENS.

SUBREQPCI-21/4 Verificación de la aleatoriedad (no criptográficos).

Se debe tener un medio para verificar de forma práctica la aleatorización de dígitos, de acuerdo a lo establecido en NIST 800-90A [35].

En [33] se establece un subrequerimiento más de REQPCI-21 PROBABILIDAD DE ENCONTRAR UN PAN (NO CRIPTOGRÁFICOS): *Al cambiar parte de un* PAN*, debe cambiar su* TOKEN *resultante.* Es un requerimiento análogo a SUBREQPCI-15/4 CAMBIO DE PAN (CRIPTOGRÁFICOS), sin embargo en el contexto de los no criptográficos reversibles, tal restricción no tiene sentido, dado que la generación del TOKEN es independiente del PAN (requerimiento REQPCI-20 GENERACIÓN Y ALMACENAMIENTO DE TOKENS (NO CRIPTOGRÁFICOS)).

REQPCI-22 Distribución imparcial (no criptográficos).

El proceso de generación de TOKENS debe garantizar una distribución de TOKENS imparcial, esto significa que la probabilidad de cualquier par PAN/TOKEN debe ser igual.

REQPCI-23 Instancias estadísticamente independientes (no criptográficos).

Si varias o diferentes instancias de la bases de datos (CDV) son usadas, cada una de estas debe ser

estadísticamente independientes.

REQPCI-24 Proceso de detokenización (no criptográficos).

El proceso de detokenización debe realizarse por medio de una búsqueda de datos o un índice dentro de la base de datos (CDV), y no por medio de métodos criptográficos.

Un subrequerimiento de REQPCI-24 Proceso de detokenización (no criptográficos) que aquí se omite establece: «*El* PAN *y el* token *debe ser probabilísticamente independientes. Cualquier método lógico o matemático no debe ser usado para tokenizar el* PAN *o detokenizar el* token». La independencia entre PAN y token ya se establece en REQPCI-20 Generación y almacenamiento de tokens (no criptográficos). No es clara la ascepción de *método lógico matemático*; una solución común para generar tokens no criptográficos es usar Pseudorandom Number Generator (PRNG), los cuales son métodos matemáticos.

REQPCI-25 Cifrado de la base de datos (no criptográficos).

Dentro de la base de datos (CDV), los PAN deben ser cifrados con una llave de mínimo 128 bits de fuerza efectiva.

REQPCI-26 Seguridad de la administración de llaves (no criptográficos).

Todas la operaciones sobre la administración de las llaves criptográficas deben realizarse en un dispositivo criptográfico seguro y aprobado: el PCI SSC se encarga de hacer validaciones; también puede ser cualquier dispositivo validado por FIPS 140-2 nivel 3 o superior [44] o por la ISO 13491-1.

Los siguientes requerimientos son referentes al cumplimiento de distintos estándares y recomendaciones (principalmente del NIST); en la sección 3.1.3 se resume el contenido de cada uno de estos:

- REQPCI-01 Validación de productos de hardware.
- REQPCI-02 Validación de productos de software.
- REQPCI-09 Acceso al proceso detokenización.
- REQPCI-13 Sobre el manejo adecuado de llaves.
- SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos).

La tabla 4.1 es una relación entre la lista de requerimientos aquí presentada y la notación del PCI SSC en [33]. Sobre todo en cuanto a los requerimientos REQPCI-09 Acceso al proceso detokenización y REQPCI-13 Sobre el manejo adecuado de llaves el documento del PCI es bastante repetitivo: se colocan 3 versiones (una para cada categoría) con prácticamente el mismo contenido.

Requerimiento	**Equivalente PCI**
REQPCI-01 Validación de productos de hardware	GT1
REQPCI-02 Validación de productos de software	GT3
REQPCI-03 Resistencia a texto claro conocido	GT4
REQPCI-04 Resistencia a sólo texto cifrado	GT5
REQPCI-05 Detección de anomalías	GT6
REQPCI-06 Distinción entre tokens y PAN	GT7
REQPCI-07 Guía de instalación	GT8
REQPCI-08 Integridad del proceso de tokenización	GT9
SUBREQPCI-09/1 Control de peticiones	GT10.1, RC2A-1, RC2A-2 RN2B y RN3B
SUBREQPCI-09/2 Registros de acceso	GT10.2
SUBREQPCI-09/3 Autenticación multifactor	GT10.3
SUBREQPCI-09/4 Accesos a nivel de sistema	GT10.4
SUBREQPCI-09/5 Accesos administrativos	GT10.5
REQPCI-10 Mapeos de token a token prohibidos	GT11
REQPCI-11 Protección contra vulnerabilidades comunes	GT12
REQPCI-12 Primitivas criptográficas usadas	GT13
SUBREQPCI-13/1 Sobre el ciclo de vida	IT4A-1 y RC4B-1
SUBREQPCI-13/2 Descripción del periodo criptográfico activo	IT4A-2 y RC4B-2
SUBREQPCI-13/3 Sobre la destrucción de las llaves	IT4A-3 y RC4B-3
SUBREQPCI-13/4 Exportar llave en claro prohibido	RC1A-1
SUBREQPCI-13/5 Entropía de generación de llaves	RC1A-2
SUBREQPCI-13/6 Llaves de uso único	RC1A-3
SUBREQPCI-14/1 Sobre el mecanismo generador (irreversibles)	IT1A-1
SUBREQPCI-14/2 Contenido en claro (irreversibles)	IT1A-2
SUBREQPCI-14/3 Creación de un diccionario (irreversibles)	IT1A-3
SUBREQPCI-14/4 Sobre el proceso de autenticación (irreversibles)	IT1A-4
SUBREQPCI-15/1 Distribución uniforme (criptográficos)	RC1B-1
Continúa en siguiente página	

Continuación	
Requerimiento	**Equivalente PCI**
SUBREQPCI-15/2 Permutación aleatoria (criptográficos)	RC1B-2
SUBREQPCI-15/3 Cambio de llave (criptográficos)	RC1B-3
SUBREQPCI-15/4 Cambio de PAN (criptográficos)	RC1B-4
SUBREQPCI-15/4 Cambio de PAN (criptográficos)	RC1B-4
SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos)	RC1B-5
REQPCI-16 Almacenamiento de tokens (criptográficos)	RC1C
REQPCI-17 Seguridad de la administración de llaves (criptográficos)	RC4A
REQPCI-18 Sobre la longitud de las llaves (criptográficos)	RC4C
REQPCI-19 Independencia estadística (criptográficos)	RC4D
REQPCI-20 Generación y almacenamiento de tokens (no criptográficos)	RN1A
SUBREQPCI-21/1 Distribución equiprobable (no criptográficos)	RN1B-1
SUBREQPCI-21/2 Permutaciones aleatorias (no criptográficos)	RN1B-2
SUBREQPCI-21/3 Parámetros de tokenización (no criptográficos)	RN1B-3
SUBREQPCI-21/4 Verificación de la aleatoriedad (no criptográficos)	RN1B-5
REQPCI-22 Distribución imparcial (no criptográficos)	RN1C
REQPCI-23 Instancias estadísticamente independientes (no criptográficos)	RN1D
REQPCI-24 Proceso de detokenización (no criptográficos)	RN2A
REQPCI-25 Cifrado de la base de datos (no criptográficos)	RN3A
Continúa en siguiente página	

Continuación	
Requerimiento	**Equivalente PCI**
REQPCI-26 Seguridad de la administración de llaves (no criptográficos)	RN4A

Tabla 4.1: Resumen de requerimientos del PCI SSC para los sistemas tokenizadores.

4.1.2. Diseño de programa tokenizador

En la sección 3.4 se enlistaron y detallaron los algoritmos tokenizadores que implementaremos. En esta sección se describe, mediante Unified Modeling Language (UML), la estructura del programa generador de tokens.

Vista estática del programa

En el diagrama de la figura 4.2 se muestra una vista estática general del programa. Las clases se dividen en dos paquetes: implementaciones y utilidades. Las utilidades abarcan estructuras de datos, interfaces, clases de excepciones y clases de prueba; todas ellas no tienen que ver de forma directa con criptografía, sino que se trata de código de soporte para las implementaciones. En las implementaciones van no solamente los algoritmos presentados en la sección 3.4, sino que todas las demás clases (generalizaciones, utilidades, implementaciones de soporte) que están relacionadas con los algoritmos tokenizadores; en el caso de este paquete, sí todo está relacionado o con criptografía o con un entorno bancario.

La figura 4.1 muestra la dependencia entre los paquetes que componene al programa: las implementaciones importan el contenido de las utilidades; ambos paquetes cuentan con equivalentes de prueba (más adelante se detalla el funcionamiento de las pruebas), en donde cada uno importa el paquete que está probando.

Retornando al diagrama 4.2: muestra solamente las clases e interfaces directamente relacionadas con los algoritmos tokenizadores; en las próximas secciones se mostrarán algunas otras vistas. Las utilidades contienen las interfaces para definir los distintos tipos de funciones y la estructura de datos (Arreglo) que se usa de manera constante en todo el programa. Las implementaciones muestran las interfaces que definen y clasifican a los algoritmos tokenizadores; también se muestra (por razones de espacio, solo el nombre) las clases concretas de los sistemas tokenizadores.

Las interfaces de las funciones utilizan plantillas para definir las propias firmas de los métodos abstractos que declaran. Esta es una forma bastante efectiva para hacer diseños débilmente acoplados (ver acoplamiento): por ejemplo, las redes Fesitel necesitan de una función de ronda sobre la que *no necesitan* saber nada en particular, solo necesitan saber que la clase define un método **operar**, que es con lo que

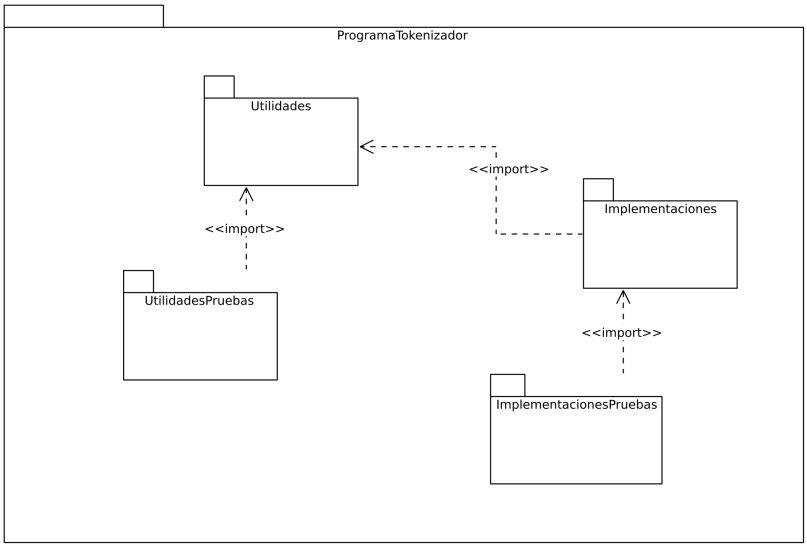


Figura 4.1: Diagrama de dependencias entre paquetes.

funciona el algoritmo definido por la red.

El Arreglo es una implementacion propia de una estuctura de datos de almacenamiento secuencial. Es solamente una interfaz a una sección de memoria (un arreglo tradicional); sin embargo, lo importante de este es que imita el formato de las estructuras de la librería estándar de C++ (esquema de constructores y destructores para gestión de memoria, uso de plantillas para programación genérica, etcétera). El arreglo de dígitos es una especialización que mantiene una representación interna tanto en cadena como en número; este arreglo es el que ocupan para comunicarse todos los algoritmos tokenizadores. Por esta última razón es por la que se mantienen las dos representaciones internas: algunos métodos requieren interpretar las entradas como números y otros como cadenas.

Los algoritmos tokenizadores implementan la interfaz definida por las funciones con inverso definiendo el tipo de ida y de vuelta como un arreglo de dígitos (Personal Account Number (PAN) y token). Las interfaces intermedias (algoritmos reversibles e irreversibles) permiten definir un comportamiento genérico para cada tipo de algoritmo; estas a su vez declaran los métodos abstractos que los algoritmos concretos deben de implementar.

El diagrama de la figura 4.3 muestra la división en componentes del programa. El componente principal es el llamado *Programa tokenizador*; este se comunica con el exterior mediante la interfaz de un algoritmo tokenizador (el puerto del costado derecho), y depende de un componente que implemente la interfaz de Card Data Vault (CDV) (el puerto del costado inferior). Existen varios clientes del componente principal: un ejecutable para generar tokens, el programa de pruebas y el programa de comparación de desempeño. En el diagrama también se muestra la composición interna del componente principal: la interfaz pública se divide en otras dos interfaces: la de los algoritmos reversibles y la de los irreversibles. En las

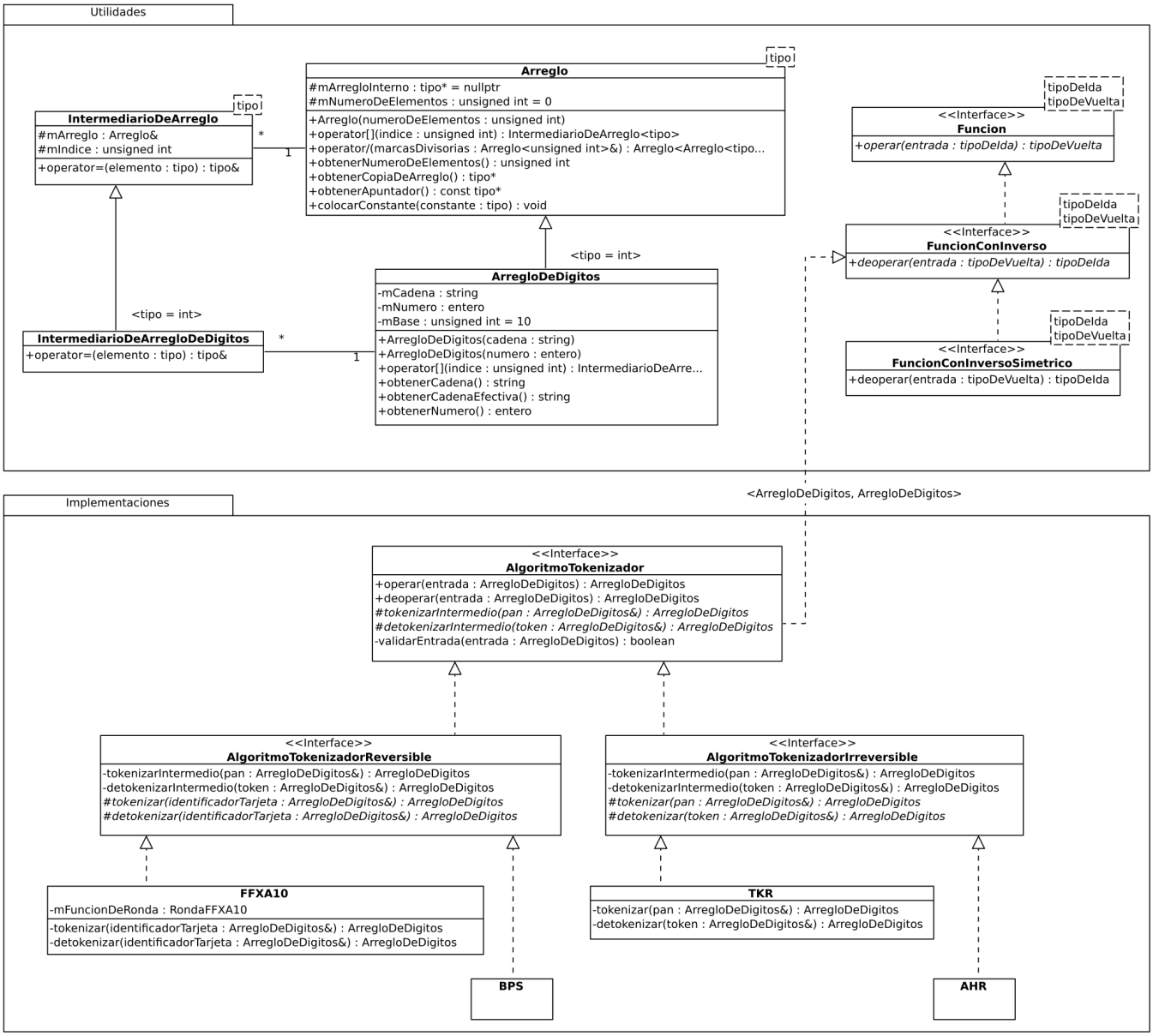


Figura 4.2: Diagrama de clases general.

próximas secciones se muestran los detalles de cada uno de los componentes internos.

Clases de FFX En la figura 4.4 se muestran las clases que conforman al módulo de FFX. Sin contar a las tres interfaces de funciones (que forman parte del paquete de utilidades) todas las clases son del paquete de implementaciones.

La clase de la red Feistel implementa la interfaz de una función con inverso (tiene tanto operación de ida, como de vuelta) y se compone (por medio de una relación de composición) de una función, utilizada como función de ronda, y de una función con inverso, utilizada como operador de combinación. Ambas clases hijas (redes Feistel alternantes o desbalanceadas) contienen un indicador de desbalanceo y sobreescriben ambas operaciones de la superclase; la red feistel alternante (ver sección 2.2.3) agrega una nueva función de ronda: una para las pares y otra para las impares.

En la parte superior de diagrama se muestran las dos posibles operaciones de combinación que soporta ffx: una a nivel de bloque y la otra nivel de caracter. La única clase en donde estas dos opciones son reestrictivas es desde la construcción de FFXA10; el desacoplamiento dado por las interfaces permite que lo único que necesite saber la red es que tiene una función que recibe un arreglo y entrega un arreglo.

Otra clase con la misma esructura que las dos anteriores es la de la función de ronda de FFXA10: implementa la interfaz de una función con inverso simétrico (que a su vez implementa un función con inverso); una instancia de esta clase en FFX es usada para construir a la red Feistel con la que se opera.

La clase de FFX es solamente un medio de comunicación con la red Feistel interna, esto es, no agrega ninguna lógica extra a los procesos de operación y operación inversa. Lo mismo ocurre con FFXA10; solo que mientras el contructor de la superclase es abierto a personalizar la red (FFX debería servir para cualquier cifrado que preserva el formato, no solo para un alfabeto de dígitos), el contructor de FFXA10 tiene parámetros bastante limitados, y su construcción de la superclase y la red Feistel ya es predefinida según la descripción de esta colección hecha en la sección 3.4.1. Es esta última clase (FFXA10) la que implementa la interfaz de los algoritmos tokenizadores reversibles.

Clases de BPS Primero, se tiene que mencionar que a pesar de que BPS puede verse como una versión mas específica de Format-preserving Feistel-based Encryption (FFX), esta clase no se realizó de forma directa a partir de la clase de la red Feistel.

A BPS se le puede considerar como un modo de operación que usa un cifrador interno BC, que a su vez usa un cifrador de ronda, situación por la que BPS es una clase que implementa el modo de operación descrito en 3.4.2, que utiliza la clase BC, cuyo funcionamiento se describe en 3.4.2, el cual se puede resumir como una red Feistel que opera sobre un cadenas construidas a partir de cualquier alfabeto, en vez de sobre cadenas binarias.

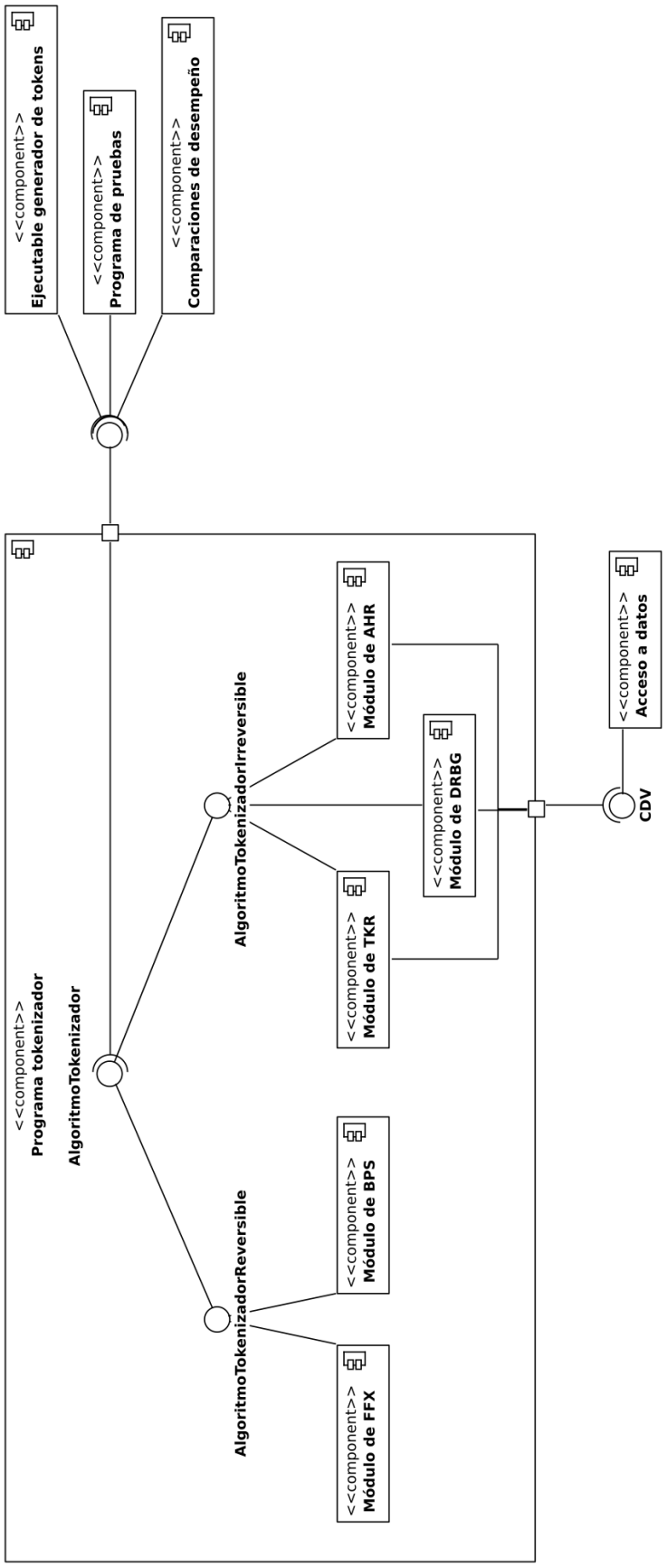


Figura 4.3: Diagrama de componentes de programa.

Figura 4.4: Diagrama de clases de módulo de FFX.

En cuanto al cifrador de ronda, esta clase se construyó usando la librería de Cripto++, dando la posibilidad de usar Advanced Encryption Standard (AES) o Data Encryption Standard (DES).

Clases de TKR En la figura 4.5 se muestran las clases relacionadas al módulo de TKR. Al igual que con el diagrama anterior, la gran mayoría de las clases pertenecen al paquete de implementaciones; las excepciones marcan el paquete al que pertenecen.

Este es el primer diagrama en donde es necesario mostrar el esquema de acceso a una fuente de datos. Este es descrito por una interfaz llamada CDV (Card Data Vault), que define las operaciones que debe de proveer cualquier fuente de datos que se desee usar dentro del programa. En un primer esquema existen dos clases concretas que implementan esta interfaz: un acceso trivial y un acceso a MySQL (ver sección 5.1). El acceso trivial permite definir métodos «de prueba» cuando se introduce una nueva operación en la interfaz; de esta forma, el código cliente puede ocupar esta implementación en lo que se desarrolla la verdadera, con conexión a base de datos. Esta manera de separar interfaz de implementación para el acceso a datos es un patrón de arquitectura conocido como Data Access Object (DAO); a grandes rasgos permite separar la definición de *qué* datos necesita la aplicación, del *cómo* los obtendrá; de esta manera, un cambio en el gestor de base de datos no debe afectar al código que usa la interfaz.

En la parte inferior del diagrama se muestra la clase *Registro*. Esta representa la única clase de datos del entorno del modelo; contiene la relación entre un PAN y un token. Como aclaración, es la única clase de datos del modelo del módulo del programa tokenizador; la interfaz web introducirá muchas otras clases al modelo.

TKR, como se explicó en la sección 3.4.3, necesita de una función que genere tokens pseudoaleatorios (la función RN) la cuál necesita de una función proveedora de bits pseudoaleatorios. En el diagrama se muestran tres versiones de esta última función: una trivial, una basada en AES y la tercera basada en un Deterministic Random Bit Generator (DRBG) (costado izquierdo del diagrama). La trivial es una primera implementación que genera números pseudoaleatorios de un modo que no es criptográficamente seguro. La basada en AES es tal cuál se decribe en el artículo [42]. Por último, la basada en un DRBG, es la que permite definir el algoritmo tokenizador que se describe en la sección 3.4.5; para este algoritmo se ocupa la misma estructura que para TKR, sin embargo, lo que importa en esta implementación es la propia creación del generador, que se muestra en secciones posteriores.

La relación entre la función RN y su mecanismo de generación interno está totalmente desacoplada: lo único que necesita la clase de la función RN es otra función que reciba enteros sin signo y entregue arreglos de bytes. Este modelo de desacoplamiento no se imitó en la relación entre la clase de TKR y la función RN porque la especificación de TKR es muy específica respecto a qué función usar.

La clase de TKR implementa la interfaz de un algoritmo tokenizador irreversible, por lo que debe

(programación por contrato) implementar los métodos para tokenizar y detokenizar.

Clases de AHR AHR utiliza la interfaz CDV para poder interactuar con la base de datos y realizar el proceso de detokenización, pues es un algoritmo irreversible. Hace uso también de algunas utilidades como potencias y el cálculo del dígito verificador mediante el algoritmo de Luhn desfasado. Utiliza también la clase del registro al momento de agregar un nuevo token a la base de datos y la clase de *Arreglo de dígitos* para implementar los métodos de *tokenizar* y *detokenizar* definidos por la interfaz *Algoritmo tokenizador irreversible*.

Ya que el algoritmo requiere de una función pública, se utiliza la clase SHA de la librería CryptoPP; de esta manera, se hace uso de SHA256 dentro uno de los métodos de AHR.

Finalmente, se utiliza la clase AES para realizar el cifrado por bloque; esta clase es una interfaz y permite utilizar el cifrado de AES por hardware si el procesador tiene las instrucciones necesarias. En caso de no tenerlo, cifra mediante las funciones de CryptoPP.

Clases de DRBG Las clases relacionadas con el generador de números pseudoaleatorio se muestran en la figura 4.6. Nuevamente, todas las clases pertenecen al paquete de las implementaciones.

Un DRBG implementa la interfaz de una función que recibe enteros sin signo y entrega arreglos de bytes. Esta clase define la estructura general de un generador: las cinco funciones definidas por el estándar del NIST (ver sección 3.3.1) y el conjunto de valores miembro que conforman el estado del generador. El DRBG declara la función abstracta para generar bytes; todos los posibles generadores concretos deben implementar esta función para poder tener un generador completo.

La clase del generador necesita una fuente de entropía (o fuente de aleatoriedad). Siguiendo el esquema de débil acomplamiento dado por las interfaces de las funciones, la fuente de entropia es una función que recibe un entero y regresa un arreglo de bytes. En el costado izquierdo del diagrama se muestra una de las posibles fuentes de entropia (aleatoriedad trivial), la cual lee de un archivo la entropía.

Existen dos implementaciones concretas para el generador de números pseudoaleatorios: una basada en una función hash y la otra basada en un cifrador por bloques (sección 3.4.5). Ambos agregan sus propios datos miembro al estado del generador, sobrecargan las funciones de cambio de semilla y desintanciación e implementan el método abstracto de la superclase. También se utilizan enumeraciones para indicar al código cliente cuales son las distintas funciones hash y cifradores por bloque que pueden ser ocupados.

Estructura de pruebas de unidades A la par que se implementa el programa descrito hasta ahora, se crearán PRUEBAS DE UNIDADES y PRUEBAS DE COMPONENTES. La idea de combinar el desarrollo de

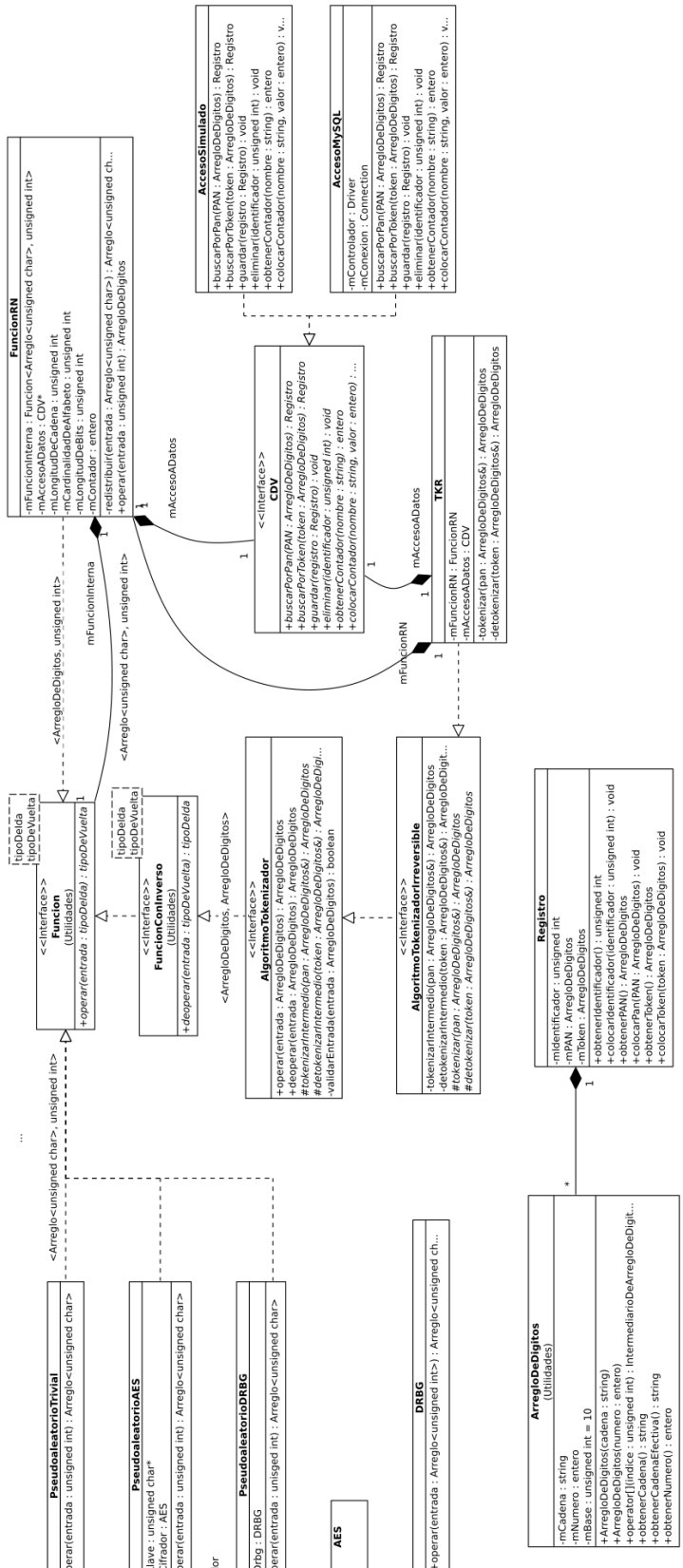


Figura 4.5: Diagrama de clases de módulo de TKR.

Figura 4.6: Diagrama de clases de módulo de DRBG.

pruebas con el proceso de implementación es encontrar problemas en una etapa temprana, facilitar el cambio de código existente y simplificar la integración de nuevos componentes. La ejecución de las pruebas está ligada a un proceso de integración continua, de forma que con cada pequeño cambio (con cada *commit*) se ejecute desde cero todo el proceso de compilación y pruebas.

Algunos lenguajes (como Python) cuentan con un modelo de pruebas integrado; otros (como Java o Javascript) cuentan con librerías muy populares (JUnit o Karma). En el caso de C++ no hay ningún soporte desde el propio lenguaje, y entre las librerías existentes tampoco hay ninguna que se haya establecido como favorita. Es por eso que se decidió crear una estructura de pruebas propia: en la figura 4.7 se muestra la estructura de las clases de pruebas para todo el programa. Este muestra a modo de ejemplo solamente algunas de las clases; todas las demás siguen el mismo esquema.

Las tres clases de la derecha, dentro del paquete de utilidades, son para agrupar y gestionar de una forma genérica a todas las pruebas. Un conjunto de pruebas tiene un lista de la clase Prueba. La clase Prueba tiene una lista de funciones de prueba. Las funciones de prueba tienen una descripción y un apuntador a una función (la propia prueba). Todas las pruebas dentro del programa deben ser especificaciones de la clase prueba. El estándar es que una clase de prueba defina todas las funciones (booleanas y estáticas) que necesite y las agregue a la lista de pruebas de la superclase.

Existen dos paquetes para agrupar a las pruebas: uno para lo que hay en implementaciones y otro para las utilidades. Lo ideal es que por cada parte del código en donde haya lógica no trivial se hagan funciones de prueba. Así, si hay una clase *Ejemplo*, debe de existir una clase *EjemploPrueba* que capture el comportamiento esperado de *Ejemplo*. En algunos casos, las clases de prueba deben tener acceso a lo que la propia clase encapsula (miembros privados o protegidos), por lo que la clase de prueba debe ser amiga de la clase probada.

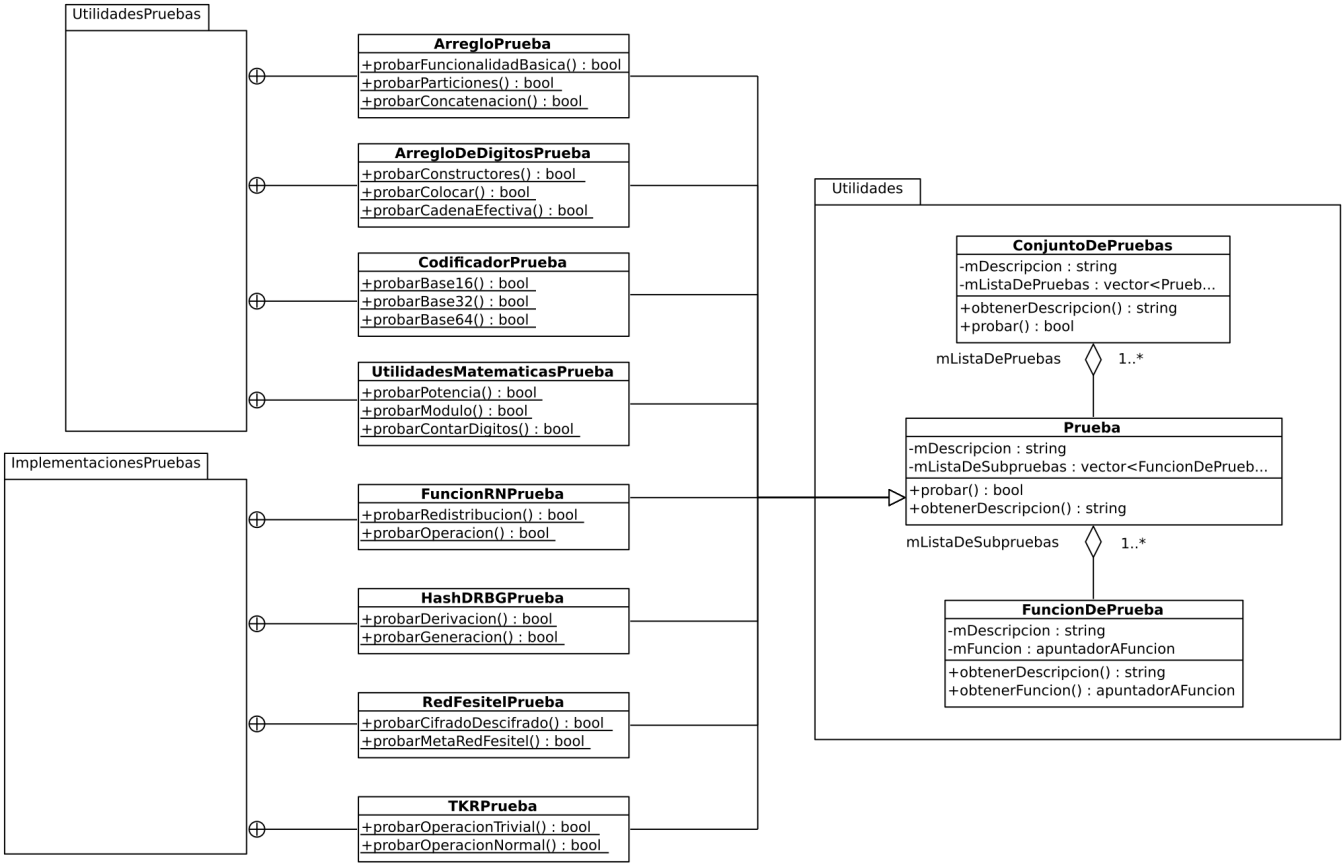


Figura 4.7: Diagrama de clases de la estructura de pruebas.

Capítulo 5

Implementación

«If 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself: 'Dijkstra would not have liked this', well that would be enough immortality for me.»
Edsger Dijkstra.

En este capítulo se presentan algunos de los aspectos más interesantes de la implementación del programa descrito en el capítulo 4. Comienza describiendo las tecnologías usadas para la implementación y desarrollo del proyecto. También se presentan los resultados de las comparaciones de desempeño y de las pruebas estadísticas de los generadores pseudoaleatorios.

5.1. Tecnologías

Las primeras decisiones tomadas con respecto a la implementación y al diseño fueron sobre el paradigma de programación y el lenguaje a utilizar. Para tomar estas decisiones se tomaron en cuenta varias consideraciones: primero, el paradigma orientado a objetos ofrece considerables ventajas con respecto a la programación estructurada, por lo que, dentro de lo posible, se buscaría un lenguaje con soporte a este paradigma; segudo, las implementaciones criptográficas tienen altos requerimientos de rendimiento, por lo que, de los posibles lenguajes, se necesitaba elegir uno que, aunque no el más rápido, sí se encontrara entre los de mayor velocidad.

A lo largo de la carrera se ha tenido contacto con bastantes lenguajes que, si bien no se dominan, sí se posee una base firme como para ser usada: C, C++, Java, Python, Javascript y PHP. Por la cuestión del rendimiento, todos los lenguajes interpretados (Python, Javascript, PHP) quedaron fuera de consideración; la decisión (dentro de los lenguajes con soporte al paradigma orientado a objetos) quedó entre Java y C++: Java (aún en las últimas versiones, en donde se han hecho considerables progresos) es mucho más lento que C++, dado que el trabajo de la máquina virtual en tiempo de ejecución es bastante considerable; por lo tanto, si de un lenguaje orientado a objetos se trataba, sería C++. La última decisión se dió entre C y C++: orientación a objetos contra rendimiento. Al final se optó por la orientación a objetos: aún cuando C es más rápido que C++, la diferencia no es tan grande, mientras que un programa con un buen diseño orientado a objetos sí puede ser mucho más mantenible que uno con un enfoque estructurado.

5.1.1. Dependencias

A continuación se enlistan las principales dependencias del proyecto. Para cada una se da una breve argumentación sobre su uso, se especifíca, en donde aplica, la licencia que tiene, y se coloca la versión ocupada.

DEP01 Compilador: GCC v7.3

Licencia: GPL v3

Página oficial: `HTTPS://GCC.GNU.ORG/`

La razón principal de su uso es porque se trata del compilador por defecto de los sistemas operativos GNU/Linux. La verisón de C++ que se utiliza es C++14 (opción `-std=c++14` del compilador). Por razones de compatibilidad con algunas otras dependencias (en particular, el conector de la base de datos)

no es posible utilizar la última versión al momento, C++17.

DEP02 Gestor de bases de datos: MariaDB v10.1

Licencia: GPL v2

Página oficial: https://mariadb.org/

El gestor de base de datos que más se ha utilizado en la carrera es MySQL, por lo que es el seleccionado para el programa tokenizador; en realidad, se trata de una bifurcación de MySQL: MariaDB, cuyo uso ha sido ampliamente difundido en varias distribuciones de GNU/Linux.

DEP03 Librería criptográfica: Crypto++ v7

Licencia: BSL v1

Página oficial: https://cryptopp.com/

Otra decisión importante a tomar antes de hacer las implementaciones de los algoritmos tokenizadores fue la librería de funciones criptográficas que se utilizaría. De todas las posibles librerías que están validadas por el National Institute of Standards and Technology (NIST) y que cuentan con una licencia para el uso público, al final se hicieron pruebas con dos: Openssl y Crypto++. Openssl está escrita en C y cuenta con un historial que le da mucha reputación (es la librería que utiliza Opengpg y Openssh). Crypto++ cuenta con una extensa red de colaboradores, implementa una gran cantidad de algoritmos y está escrita sólo en C++. De ambas, la que se encontró más fácil de usar fue Crypto++, por lo que es la que se utiliza en las implementaciones.

5.1.2. Dependencias de desarrollo

A continuación se enlistan las dependencias de desarrollo. Aunque estas no son necesarias para el producto final, sí lo son para el cambio y mantenimiento del programa. Muchas de estas dependencias se usan como servicios (v. gr. github o travis). Las elecciones de estas dependencias no siguieron siempre un proceso de eliminación como las anteriores, sino que fueron seleccionadas por cuestión de facilidad de uso (de los autores).

DEP04 Control de versiones: Git v2.17

Licencia: GPL v2

Página oficial: https://git-scm.com/

Es el Version Control System (VCS) más común en entornos GNU/Linux. Es muy importante su integración con Github.

DEP05 Servicio de *hosting* para el VCS: Github

Licencia: No aplica (servicio)

Página oficial: https://github.com

Aunque a la fecha soporta otros tipos de VCS, originalmente era exclusivo a Git. Su función principal es mantener una copia remota del código, desde la cuál un desarrollador baja la última versión y en donde actualiza sus cambios. También se llegó a utilizar algunas funcionalidades adicionales, como el reporte de problemas, los comentarios en código y la integración con Travis.

DEP06 Herramienta de integración continua: Travis

Licencia: No aplica (servicio)

Página oficial: HTTPS://TRAVIS-CI.ORG/

Entre sus principales tareas están compilar desde cero el proyecto, ejecutar todas las pruebas, generar y desplegar la documentación y el reporte de vuelta a github, ejecutar las pruebas de desempeño y guardar los resultados en la base de datos.

DEP07 Documentación de código: Doxygen

Licencia: GPL v2

Página oficial: HTTPS://WWW.STACK.NL/~DIMITRI/DOXYGEN/

Herramienta para generar documentación del código. Permite mantener la documentación en los mismos archivos fuente, ya que la salida es generada a partir de comentarios. Se cuentan con dos versiones, una en HTML (HTTPS://RQF7.GITHUB.IO/PROYECTO_LOVELACE/DOCUMENTACION_DOXYGEN/HTML/INDEX.HTML) y la otra como PDF (HTTPS://RQF7.GITHUB.IO/PROYECTO_LOVELACE/DOCUMENTACION_DOXYGEN/LATEX/REFMAN.PDF).

5.2. Programa para generar *tokens*

A continuación se explican las partes más importantes del programa: las implementaciones de los algoritmos tokenizadores mostrados en la sección 3.4.

En un intento por no complicar demasiado la exposición del programa, se redujo al máximo la aparición de código en el documento. Aquí solo se muestran las implementaciones de los algoritmos tokenizadores, y aún en estos casos, solamente se muestran las fracciones más relevantes. Al día de hoy, el programa cuenta con alrededor de 16000 líneas de código (contando comentarios y líneas en blanco), por lo que cualquier intento de incluir aquí a todo el programa resultaría en un documento de dimensiones estratosféricas. Si lo que se busca es una implementación en específico que no aparezca aquí, o mayor detalle en el código, se puede consultar la documentación en línea: HTTPS://RQF7.GITHUB.IO/PROYECTO_LOVELACE/DOCUMENTACION_DOXYGEN/HTML/INDEX.HTML. Durante el desarrollo del programa se han hecho muchos esfuerzos para mantenerla completa.

Antes de iniciar con la explicación sobre las implementaciones sería comveniente aclarar un par de puntos sobre las reglas que sigue el formato del código, para facilitar la lectura: todos los identificadores se encuentran en español (el conjunto permitido por los caracteres American Standard Code for Information Interchange (ASCII)); las clases y espacios de nombres siguen `EsteFormato` mientras

que las variables y funciones siguen `esteOtroFormato`; las variables miembro de una clase llevan simepre en el nombre una «m» como prefijo. Se pueden encontrar mayores detalles en: `HTTPS://RQF7.GITHUB.IO/PROYECTO_LOVELACE/REGLAS_DE_ESTILO.HTML`.

5.2.1. Módulo de FFX

La operación Format-preserving Feistel-based Encryption (FFX) A10 se describió en la sección 3.4.1. La parte medular de su operación se encuentra en la función de ronda que ocupa la red Feistel interna (pseudocódigo 3.6); la implementación de esta función se muestra en el código fuente 1. FFX A10 funciona con una red Feistel alternante (sección 2.2.3), por lo que también es importante la implementación de este proceso; el código fuente 2 muestra la operación de cifrado y 3 muestra el descifrado.

Como se mostró en el diseño del programa (capítulo 4), la función de ronda implementa la interfaz de una función con inverso simétrico (diagrama 4.4), por lo que la función `operar` correspone a la definida por la interfaz. Otro aspecto importante del código 1 es la primitiva criptográfica usada: una instancia de Advanced Encryption Standard (AES) Cipher-block Chaining (CBC) Message Authentication Code (MAC) de Crypto++. La operación de esa función se puede dividir en tres partes claramente distingibles: el armado de la entrada a la primitiva (hasta la línea 160); la operación de la primitiva (hasta la línea 166); y por último, el formateo de la salida para la preservación del formato (el resto de la función).

Una red Feistel alternante es una red Feistel, la cuál a su vez implementa a una función con inverso; es por esto que los métodos mostrados en los códigos fuente 2 y 3 se llaman `operar` y `deoperar`, respectivamente. La red feistel alternante tiene dos funciones de ronda: una para las pares y otra para las impares. Estas son clases que tengan la forma de una función con inverso: la clase de la función de ronda de FFX A10, presentada en 1, implementa a una función con inverso simétrico (la cual a su vez implementa a una función con inverso). De esta forma se puede instanciar a FFX manteniendo totalmente desacoplado al código de las redes Feistel (ver acoplamiento).

5.2.2. Módulo de BPS

El funcionamiento de BPS se describe de forma mas amplia en la sección 3.4.2.

A pesar de que se puede ver a BPS como una versión más especifica de Format-preserving Feistel-based Encryption (FFX), esta implementación se hizo independiente de la de FFX.

De forma general, el módulo de BPS consta de los 2 métodos principales, el de cifrar y el de descifrar, los cuales son la implementación del modo de operación utilizado por BPS y que se describe en la sección 3.4.2. Estos métodos se muestran en los códigos 4, 5, 6 y 7.

Otro parte crucial de BPS es el cifrador interno BC, por lo cual es importante mostrar su implemen-

```
125  /**
126   * Implementación de función de ronda con CBC-MAC-AES.
127   *
128   * \return Texto cifrado de la longitud necesaria.
129   */
130
131  template<typename tipo>
132  Arreglo<tipo> RondaFFXA10<tipo>::operar(
133    const std::vector<Arreglo<tipo>> &textoEnClaro    /**< Texto a cifrar. */
134  )
135  {
136    /* Determinar longitud de entrada. La única longitud variable es la del
137     * tweak. */
138    int longitudEntrada = 8 + mLongitudTweak + sizeof(int);
139    unsigned char entrada[longitudEntrada];
140
141    /* Datos fijos. */
142    entrada[0] = 0;
143    entrada[1] = static_cast<unsigned char>(mTipoDeRed);
144    entrada[2] = static_cast<unsigned char>(mTipoDeSuma);
145    entrada[3] = static_cast<unsigned char>(mRadix);
146    entrada[4] = static_cast<unsigned char>(mLongitud);
147    entrada[5] = static_cast<unsigned char>(mDesbalanceo);
148    entrada[6] = static_cast<unsigned char>(mNumeroDeRondas);
149    entrada[7] = 0;
150
151    /* Tweak. */
152    for (unsigned int i = 0, j = 8; i < mLongitudTweak; i++, j++)
153      entrada[j] = mTweak[i];
154
155    /* Representación numérica de mensaje. */
156    entero representacionNumero = convertirANumero<tipo, entero>(
157      textoEnClaro[0], mRadix);
158    for (int i = 0; i < 8; i++)
159      entrada[7 + mLongitudTweak + i + 1] =
160        static_cast<unsigned char>(representacionNumero >> (8 + i));
161
162    /* Generar MAC */
163    CryptoPP::CBC_MAC<CryptoPP::AES> cbcmac {mLlave};
164    cbcmac.Update(entrada, longitudEntrada);
165    unsigned char mac[cbcmac.DigestSize()];
166    cbcmac.TruncatedFinal(mac, cbcmac.DigestSize());
167
168    /* Partir por mitad */
169    Arreglo<int> ladoIzquierdo (8), ladoDerecho(8);
170    for (int i = 0; i < 16; i++)
171      if (i < 8)
172        ladoIzquierdo[i] = mac[i];
173      else
174        ladoDerecho[i - 8] = mac[i];
175
176    /* Formatear salida a longitud adecuada. */
177    entero numeroIzquierdo = convertirANumero<int, entero>(ladoIzquierdo, 256);
178    entero numeroDerecho = convertirANumero<int, entero>(ladoDerecho, 256);
179    entero z;
180    if (mLongitud <= 9)
181      z = modulo(numeroDerecho, potencia<entero>(mRadix, mLongitud));
182    else
183      z = (modulo(numeroIzquierdo, potencia<entero>(mRadix, mLongitud - 9))
184        * potencia<entero>(mRadix, 9))
185        + modulo(numeroDerecho, potencia<entero>(mRadix, 9));
186
187    return convertirAArreglo<tipo, entero>(z, mRadix, mLongitud);
188  }
```

Código fuente 1: Función de ronda de FFX

```
146  /**
147   * Realiza todas las rondas del proceso de cifrado. La operación de una
148   * ronda se resume como:
149   * Si es par, PI_{i} = F(PD_{i - 1}) \xor PI_{i - 1}
150   *            PD_{i} = PD_{i - 1}
151   * Si es impar, PD_{i} = F(PI_{i - 1}) \xor PD_{i - 1}
152   *              PI_{i} = PI_{i - 1}
153   * La función de ronda usada depende de si esta es par, o impar.
154   *
155   * \return Bloque cifrado.
156   */
157
158  template <typename tipo>
159  Arreglo<tipo> RedFeistelAlternante<tipo>::operar(
160    const std::vector<Arreglo<tipo>>& textoEnClaro     /**< Bloque a cifrar. */
161  )
162  {
163    Arreglo<Arreglo<tipo>> partes = textoEnClaro[0] / Arreglo<unsigned int>{
164      (textoEnClaro[0].obtenerNumeroDeElementos() / 2) + mDesbalanceo};
165    for (mRondaActual = 0; mRondaActual < mNumeroDeRondas; mRondaActual++)
166    {
167      if (mRondaActual % 2 == 0)
168        partes[0] = std::move(mOperadorSuma->operar(
169          {partes[0], mFuncionDeRonda->operar({partes[1]})}));
170      else
171        partes[1] = std::move(mOperadorSuma->operar(
172          {partes[1], mFuncionDeRondaImpar->operar({partes[0]})}));
173    }
174    return static_cast<Arreglo<int>>(partes[0])
175      || static_cast<Arreglo<int>>(partes[1]);
176  }
```

Código fuente 2: Cifrado con red Feistel alternante

```
178  /**
179   * Realiza todas las rondas del proceso de descifrado. La operación de las
180   * rondas es la misma que en el proceso de cifrado; no se hacen una sola
181   * función porque es necesario mantener la estructura dada por la superclase
182   * y porque es necesario llevar el contador de rondas en orden descendente.
183   *
184   * \return Bloque descifrado.
185   *
186   * \sa http://www.cplusplus.com/reference/utility/move/
187   */
188
189  template <typename tipo>
190  Arreglo<tipo> RedFeistelAlternante<tipo>::deoperar(
191    const std::vector<Arreglo<tipo>>& textoCifrado   /**< Bloque a descifrar. */
192  )
193  {
194    Arreglo<Arreglo<tipo>> partes = textoCifrado[0] / Arreglo<unsigned int>{
195      (textoCifrado[0].obtenerNumeroDeElementos() / 2) + mDesbalanceo};
196    for (mRondaActual = mNumeroDeRondas - 1; mRondaActual >= 0; mRondaActual--)
197    {
198      if (mRondaActual % 2 == 0)
199        partes[0] = std::move(mOperadorSuma->deoperar({
200          partes[0], mFuncionDeRonda->operar({partes[1]})}));
201      else
202        partes[1] = std::move(mOperadorSuma->deoperar({
203          partes[1], mFuncionDeRondaImpar->operar({partes[0]})}));
204      if (mRondaActual == 0)
205        break;
206    }
207    return static_cast<Arreglo<int>>(partes[0])
208      || static_cast<Arreglo<int>>(partes[1]);
209  }
```

Código fuente 3: Descifrado con red Feistel alternante

```
77
78 /**
79  * Este método sirve para cifrar la cadena dada con la llave y el tweak dados.
80  * El funcionamiento de este método es el del modo de operación del algoritmo
81  * de cifrado que preserva el formato BPS.
82  */
83
84 string CifradorBPS::cifrar(string mensaje, byte llave[], mpz_class tweak)
85 {
86   CifradorBC BC;
87   BC.colocarTipoCifrador(mTipoCifrador);
88   int tamCifradorDeRonda = BC.obtenerCifradorDeRonda().obtenerTamBloque();
89
90   /* Obtención del tamaño máximo de bloque del cifrador interno BC */
91   unsigned int tamTotal, tamMax;
92   tamTotal  = mensaje.size();
93   tamMax = 2 * ((tamCifradorDeRonda - 32) * log(2)) / log(mAlfabeto.size());
94
95   /* Configuración del cifrador interno BC */
96   BC.colocarAlfabeto(mAlfabeto);
97   BC.colocarTamBloque(tamMax);
98   BC.colocarNumRondas(mNumRondas);
99
100  /* En caso de que la cadena dada tenga una longitud menor al tamaño máximo
101  del cifrador BC, simplemente se cifra la cadena con BC */
102  if(tamTotal <= tamMax)
103  {
104    BC.colocarTamBloque(tamTotal);
105    return BC.cifrar(mensaje, llave, tweak);
106  }
107
108  string bloqueA     {""};
109  string salida      {""};
110  string salidaFinal {""};
111
112  unsigned int numBloques = tamTotal / tamMax;
113  unsigned int tamUltimoBloque = tamTotal % tamMax;
114  mpz_class contadorU = 0;
115  mpz_class tweakU = 0;
116
```

Código fuente 4: Función de cifrado de BPS (parte 1).

```
116
117   /* En caso de que la cadena dada tenga una longitud mayor al tamaño máximo
118   del cifrador BC, se van cifrando bloques x de longitud igual a la longitud
119   máxima de BC, donde x es igual a la suma modular de un bloque i con un
120   bloque i-1 */
121   BC.colocarTamBloque(tamMax);
122   for (unsigned int i = 0; i < numBloques; i++)
123   {
124     /* Xor del contador u = 2^16 + 2^48 con el tweak que entrara
125     al cifrador BC */
126     contadorU = i;
127     contadorU = (contadorU << 16) + (contadorU << 48);
128     tweakU = tweak^contadorU;
129
130     /* Cifrado de la suma modular del bloque i con el bloque i-1 */
131     bloqueA = mensaje.substr(i * tamMax, tamMax);
132     salida = BC.cifrar(mCodificador.sumaMod(bloqueA, salida), llave, tweakU);
133     salidaFinal += salida;
134   }
135
136   /* En caso de que la cadena dada tenga una longitud que no es múltiplo
137   del tamaño máximo de BC, el último bloque se cifra configurando BC
138   a su longitud */
139   if(tamUltimoBloque != 0)
140   {
141     /* Xor del contador u = 2^16 + 2^48 con el tweak que entrará
142     al cifrador BC */
143     BC.colocarTamBloque(tamUltimoBloque);
144     contadorU = numBloques;
145     contadorU = (contadorU << 16) + (contadorU << 48);
146     tweakU = tweak ^ contadorU;
147
148     /* Cifrado de la suma modular del bloque i con el bloque i-1 */
149     bloqueA = mensaje.substr(tamTotal - tamUltimoBloque, tamUltimoBloque);
150     salida = BC.cifrar(mCodificador.sumaMod(bloqueA, salida), llave, tweakU);
151     salidaFinal += salida;
152   }
153
154   return salidaFinal;
155 }
156
```

Código fuente 5: Función de cifrado de BPS (parte 2).

```
158
159 /**
160  * Este método sirve para descifrar la cadena dada con la llave y tweak dados.
161  * El funcionamiento de este método es el del modo de operación del algoritmo
162  * de cifrado que preserva el formato BPS.
163  */
164
165 string CifradorBPS::descifrar(string mensaje, byte llave[], mpz_class tweak)
166 {
167   CifradorBC BC;
168   BC.colocarTipoCifrador(mTipoCifrador);
169   int tamCifradorDeRonda = BC.obtenerCifradorDeRonda().obtenerTamBloque();
170
171   /* Obtención del tamaño máximo de bloque del cifrador interno BC */
172   unsigned int tamTotal, tamMax;
173   tamTotal = mensaje.size();
174   tamMax = 2 * ((tamCifradorDeRonda - 32) * log(2)) / log(mAlfabeto.size());
175
176   /* Configuración del cifrador interno BC */
177   BC.colocarAlfabeto(mAlfabeto);
178   BC.colocarTamBloque(tamMax);
179   BC.colocarNumRondas(mNumRondas);
180
181   /* En caso de que la cadena dada tenga una longitud menor al tamaño máximo
182   del cifrador BC, simplemente se descifra la cadena con BC */
183   if(tamTotal <= tamMax)
184   {
185     BC.colocarTamBloque(tamTotal);
186     return BC.descifrar(mensaje, llave, tweak);
187   }
188
189   string bloqueA     {""};
190   string bloqueB     {""};
191   string salida      {""};
192   string salidaFinal {""};
193
194   unsigned int numBloques = tamTotal / tamMax;
195   unsigned int tamUltimoBloque = tamTotal % tamMax;
196   mpz_class contadorU = 0;
197   mpz_class tweakU = 0;
198
```

Código fuente 6: Función de descifrado de BPS (parte 1).

```
198
199   /* En caso de que la cadena dada tenga una longitud que no es múltiplo
200   del tamaño máximo de BC, el último bloque se descifra configurando BC
201   a su longitud */
202   if(tamUltimoBloque != 0)
203   {
204     /* Xor del contador u = 2^16 + 2^48 con el tweak que entrara
205     al cifrador BC */
206     BC.colocarTamBloque(tamUltimoBloque);
207     contadorU = numBloques;
208     contadorU = (contadorU << 16) + (contadorU << 48);
209     tweakU = tweak^contadorU;
210
211     /* Obtención del bloque i e i-1 */
212     bloqueB = mensaje.substr(tamTotal - tamUltimoBloque, tamUltimoBloque);
213     bloqueA = mensaje.substr((numBloques - 1) * tamMax, tamMax);
214
215     /* Resta modular del resultado de decifrar el bloque i, menor el bloque i-1 */
216     salida = mCodificador.restaMod(BC.descifrar(bloqueB, llave, tweakU), bloqueA);
217     salidaFinal = salida + salidaFinal;
218   }
219
220   /* En caso de que la cadena dada tenga una longitud mayor al tamaño máximo
221   del cifrador BC, se van obteniendo bloques descifrados x de longitud igual
222   a la longitud máxima de BC, donde x es igual a la resta modular del
223   resultado de descifrar un bloque un bloque i con un bloque i-1 */
224   BC.colocarTamBloque(tamMax);
225   for(int i = numBloques - 1; i > 0; i--)
226   {
227     /* Xor del contador u = 2^16 + 2^48 con el tweak que entrara
228     al cifrador BC */
229     contadorU = i;
230     contadorU = (contadorU << 16) + (contadorU << 48);
231     tweakU = tweak ^ contadorU;
232
233     /* Obtención del bloque i e i-1 */
234     bloqueB = mensaje.substr(i * tamMax, tamMax);
235     bloqueA = mensaje.substr((i - 1) * tamMax, tamMax);
236
237     /* Resta modular del resultado de decifrar el bloque i, menor el bloque i-1 */
238     salida = mCodificador.restaMod(BC.descifrar(bloqueB, llave, tweakU), bloqueA);
239     salidaFinal = salida + salidaFinal;
240   }
241
242   /* Descifrado del primer bloque en caso de que la cadena dada fuese de una
243   longitud mayor al tamaño máximo del cifrador BC */
244   bloqueA = mensaje.substr(0, tamMax);
245   salida = BC.descifrar(bloqueA, llave, tweak);
246   salidaFinal = salida + salidaFinal;
247
248   return salidaFinal;
249 }
250
```

Código fuente 7: Función de descifrado de BPS (parte 2).

tación. Los códigos 8 y 9 están basados en el pseudocódigo para el cifrador BC descrito en 3.4.2.

Como se observa en el código 9, el cifrador interno usa a su vez un cifrador de ronda, que es un cifrador por bloques (en este caso Advanced Encryption Standard (AES) o Data Encryption Standard (DES)), el cual se hizo usando la librería de Crypto++.

5.2.3. Módulo de TKR

La descripción de los algoritmos que componen a este método se hizo en la sección 3.4.3. En la sección 4.1.2 se mostró la vista estática que tiene este módulo. En esta sección se muestran los aspectos más importantes de su implementación.

Los códigos fuentes 10 y 11 mustran los procesos de cifrado y descifrado, respectivamente. Estas operaciones están descritas por los pseudocódigos 3.10 y 3.11. La operación de cifrado depende de una función psudoaleatorioa; dentro del contexto de la clase TKR, esta es cualquier otra clase que implemente a una función que recibe enteros y regresa arreglos de dígitos. La operación de descifrado es simplemente una consulta en la base de datos; en caso de una búsqueda infructuosa, se lanza una excepción.

Como se muestra en el diagrama 4.5, TKR implementa la interfaz de un algoritmo tokenizador irreversible (que a su vez implementa a un algoritmo tokenizador), por lo que los métodos mostrados en los códigos fuente 10 y 11 son los definidos por la interfaz. Al tratarse de un método irreversible, ambos procesos (tokenización y detokenización) reciben el Personal Account Number (PAN) o el token completos, según sea el caso.

La función del código fuente 12 es la implementación del pseudocódigo descrito en 3.12. Esta clase implementa la interfaz de una función que recibe enteros y regresa arreglos de dígitos (siguiendo el modelo de débil acomplamiento dado por las interfaces de las funciones). Básicamente, se trata de la transformación de los bytes aleatorios dados por la función interna en el número de dígitos pedidos. En el contexto de esta clase, la función interna tiene el mismo aspecto: una función que recibe enteros y regresa arreglos de bytes.

Por último, el código fuente 13 muestra cómo se puede ocupar un cifrado de bloques (Advanced Encryption Standard (AES) de 128 bits, en este caso) para producir arreglos de bytes con aspecto aleatorio. La operación de esta función es análoga al modo de operación Counter Mode (CTR) (sección 2.2.6)

5.2.4. Módulo de AHR

En esta sección se explica, de manera general, la implementación del algoritmo AHR, descrito en la sección 3.4.4.

```
111
112 /**
113  * Este método sirve para cifrar la cadena dada con la llave y el tweak dados.
114  * El funcionamiento de este método es el del cifrador interno BC del
115  * algoritmo de cifrado que preserva el formato BPS.
116  */
117
118 string CifradorBC::cifrar(string mensaje, byte llave[], mpz_class tweak)
119 {
120   /* Mensaje de error para cuando la cadena dada es de una longitud
121   distinta al tamaño de bloque */
122   if(mensaje.size() != mTamBloque)
123   {
124     cout << "ERROR, cadena de longitud distinta a la establecida.";
125     cout << endl;
126     exit(-1);
127   }
128
129   UtilidadesBPS util;
130   unsigned int l, r;
131   string mensajeIzq{""};
132   string mensajeDer{""};
133
134   /* Obtención de las longitudes de la parte izquierda (l) y derecha (r)
135   de la cadena dada para cifrar */
136   l = (mTamBloque%2 == 0) ? mTamBloque/2 : (mTamBloque+1)/2;
137   r = (mTamBloque%2 == 0) ? mTamBloque/2 : (mTamBloque-1)/2;
138
139   /* Construcción de la parte izquierda y derecha del la cadena dada */
140   for(unsigned int i=0; i<mTamBloque; i++)
141     if(i<l) mensajeIzq += mensaje[i];
142     else    mensajeDer += mensaje[i];
143
144   /* Obtención de los valores de composición de la parte izquierda y derecha */
145   mpz_class ladoIzq = componer(mensajeIzq);
146   mpz_class ladoDer = componer(mensajeDer);
147
148   /* Obtención los subtweaks izquierdo y derecho */
149   mpz_class tweakDer = tweak & (0xFFFFFFFF);
150   mpz_class tweakIzq = (tweak >> 32) & (0xFFFFFFFF);
151
152   mpz_class tweakDerAux = 0;
153   mpz_class tweakIzqAux = 0;
154   mpz_class entradaCr   = 0;
155   mpz_class salidaCr    = 0;
156   mpz_class maxS1       = 0;
157   mpz_class maxSR       = 0;
158   string entradaCrStr   {""};
159   string salidaCrStr    {""};
160
161   /* Asignación de s^l y s^r en las variables maxS1 y maxSR */
162   mpz_ui_pow_ui(maxS1.get_mpz_t(), mCardinalidad, l);
163   mpz_ui_pow_ui(maxSR.get_mpz_t(), mCardinalidad, r);
164   int tamCifradorDeRonda = mCifradorDeRonda.obtenerTamBloque();
165
```

Código fuente 8: Cifrador interno BC de BPS (parte 2).

```
165
166   /* Ciclo de la red Feistel */
167   for (unsigned int i = 0; i < mNumRondas; i++)
168   {
169     if (i % 2 == 0)
170     {
171       /* Corrimiento a la izquierda de f-32 bits del subtweak derecho
172       xor con el contador, donde f es el tamaño de bloque del cifrador
173       de ronda */
174       tweakDerAux = (tweakDer ^ i) << (tamCifradorDeRonda - 32);
175
176       /* Suma del subtweak recorrido con el lado derecho */
177       entradaCr = tweakDerAux + ladoDer;
178       entradaCrStr = util.numeroACadena(entradaCr);
179
180       /* Cifrado de la suma anterior */
181       salidaCrStr = mCifradorDeRonda.cifrar(entradaCrStr,llave);
182       salidaCr = util.cadenaANumero(salidaCrStr);
183
184       /* Suma modular del lado izquierdo con el resultado del
185       cifrado anterior */
186       ladoIzq = util.mod(ladoIzq, maxS1) + util.mod(salidaCr, maxS1);
187       ladoIzq = util.mod(ladoIzq, maxS1);
188     }
189     else
190     {
191       /* Corrimiento a la izquierda de f-32 bits del subtweak izquierdo
192       xor con el contador, donde f es el tamaño de bloque del cifrador
193       de ronda */
194       tweakIzqAux = (tweakIzq ^ i) << (tamCifradorDeRonda - 32);
195
196       /* Suma del subtweak recorrido con el lado izquierdo */
197       entradaCr = tweakIzqAux + ladoIzq;
198       entradaCrStr = util.numeroACadena(entradaCr);
199
200       /* Cifrado de la suma anterior */
201       salidaCrStr = mCifradorDeRonda.cifrar(entradaCrStr, llave);
202       salidaCr = util.cadenaANumero(salidaCrStr);
203
204       /* Suma modular del lado derecho con el resultado del
205       cifrado anterior */
206       ladoDer = util.mod(ladoDer, maxSR) + util.mod(salidaCr, maxSR);
207       ladoDer = util.mod(ladoDer, maxSR);
208     }
209   }
210
211   /* Descomposición del lado izquierdo y derecho para concatenarlas y
212   obtener la cadena cifrada */
213   string mensajeCifrado;
214   mensajeCifrado += descomponer(ladoIzq,l);
215   mensajeCifrado += descomponer(ladoDer,r);
216   return mensajeCifrado;
217 }
218
```

Código fuente 9: Cifrador interno BC de BPS (parte 2).

```
41 /**
42  * Proceso para generar un token a partir de un número de tarjeta. Primero
43  * se busca en la base de datos, si ya hay un token para el número dado, es
44  * este el que se regresa; sino, se crea uno (y se inserta en la base) con la
45  * función pseudoaleatoria.
46  *
47  * \return Token asociado al PAN dado.
48  */
49
50 ArregloDeDigitos TKR::tokenizar(
51   const ArregloDeDigitos& pan                   /**< Número de tarjeta. */
52 )
53 {
54   Registro informacion = mBaseDeDatos->buscarPorPan(pan);
55   if (informacion.obtenerToken() == Arreglo<int>{})
56   {
57     auto division = pan
58       / Arreglo<unsigned int>{6, pan.obtenerNumeroDeElementos() - 1};
59     ArregloDeDigitos temporal =
60       mFuncionPseudoaleatoria->operar(
61         {static_cast<ArregloDeDigitos>(division[1]).obtenerNumeroDeElementos()});
62     temporal = static_cast<ArregloDeDigitos>(division[0]) || temporal;
63     informacion.colocarToken(temporal
64       || ArregloDeDigitos{modulo(algoritmoDeLuhn(temporal) + 1, 10)});
65     informacion.colocarPAN(pan);
66     mBaseDeDatos->guardar(informacion);
67   }
68   return informacion.obtenerToken();
69 }
```

Código fuente 10: Proceso de cifrado de TKR.

```
71 /**
72  * Proceso de detokenización. Simplemente busca en la base de datos el
73  * PAN asociado al token dado.
74  *
75  * \return PAN asociado al token dado.
76  */
77
78 ArregloDeDigitos TKR::detokenizar(
79   const ArregloDeDigitos& token             /** Token (generado previamente). */
80 )
81 {
82   Registro informacion = mBaseDeDatos->buscarPorToken(token);
83   if (informacion.obtenerPAN() == Arreglo<int>{})
84     throw TokenInexistente{"El token no está en la base de datos."};
85   return informacion.obtenerPAN();
86 }
```

Código fuente 11: Proceso de descifrado de TKR.

```
53 /**
54  * Utiliza la función interna para generar una cadena de bits pseudoaleatoria;
55  * después interpreta esa cadena con los datos miembro (la cardinalidad y
56  * la longitud de la cadena) para regresar la cadena con el formato adecuado.
57  *
58  * \return Arreglo pseudoaleatorio con \ref mLongitudDeBits de longitud.
59  */
60
61 ArregloDeDigitos FuncionRN::operar(const vector<unsigned int>& entrada)
62 {
63   Arreglo<unsigned char> binarioAleatorio = mFuncionInterna->operar({
64     static_cast<entero>(4 * mLongitudDeCadena * mLongitudDeBits), mContador});
65   Arreglo<unsigned char> binarioDistribuido = redistribuir(binarioAleatorio);
66   ArregloDeDigitos resultado (mLongitudDeCadena);
67   for (unsigned int i = 0, j = 0; i < mLongitudDeCadena; j++)
68   {
69     unsigned int numero = static_cast<unsigned int>(binarioDistribuido[j]);
70     if (numero < mCardinalidadDeAlfabeto)
71       resultado[i++] = numero;
72   }
73   mContador++;
74   return resultado;
75 }
```

Código fuente 12: Función RN.

```
51 /**
52  * Operación de función: recibe en el vector de los argumentos un
53  * contador (el estado el algoritmo en RN) y la longitud de bits que el
54  * resultado debe tener. La longitud real regresada es el múltiplo del
55  * tamaño de bloque de AES (128 bits) más cercano hacia arriba.
56  *
57  * Este es el algoritmo propuesto en el artículo de TKR para instanciar f
58  * con un cifrador por bloques.
59  *
60  * \return Arreglo de bytes con contenido del cifrado.
61  */
62
63 Arreglo<unsigned char> PseudoaleatorioAES::operar(
64   const std::vector<entero>& entrada    /**< Contador y longitud de resultado. */
65 )
66 {
67   unsigned int numeroDeBloques = static_cast<unsigned int>(ceil(entrada[0] / 128.0));
68   Arreglo<unsigned char> resultado (numeroDeBloques * 16);
69   unsigned char bufferUno[16];
70   memset(bufferUno, 0, 16);
71   for (int i = 0; i < 8; i++)
72     bufferUno[i] = static_cast<unsigned char>(entrada[1] >> (i * 8));
73   unsigned char bufferDos[16];
74   cifrar(bufferUno, bufferDos, 0);
75   for (unsigned int i = 0; i < numeroDeBloques; i++)
76   {
77     cifrar(bufferDos, bufferUno, i);
78     for (int j = 0; j < 16; j++)
79       resultado[(i * 16) + j] = bufferUno[j];
80   }
81   return resultado;
82 }
```

Código fuente 13: Generación de bytes pseudoaleatorios basado en AES.

Partiendo de lo general hacia lo particular, en la sección de código 14 se ve el método *principal*; primero, dependiendo de la longitud del Personal Account Number (PAN) calcula los límites entre los cuales el token será válido. Posteriormente, mediante el método *crearBloqueT*, descrito en 15, se concatena a la salida de la función SHA256 (que recibe como entrada el Issuer Identification Number (INN)), la representación binaria de la entrada X, que es el número de cuenta que se va a tokenizar. Una vez que se tiene el bloque T, se cifra mediante Advanced Encryption Standard (AES) con una llave de 256 bits y se obtiene la representación decimal de los últimos bits del bloque cifrado; el resultado de esta operación es el token candidato, pues aún falta revisar si el token se encuentra dentro de los límites obtenidos al principio; en caso de que no cumpla, se usa cifrado de caminata cíclica hasta obtener un token válido. Finalmente, se revisa en la base de datos si el token creado existe (este algoritmo permite generar varios tokens para un mismo PAN): si no existe, guarda la relación PAN-token en la base de datos; si sí, aumenta en uno el INN y vuelve a ejectuar el algoritmo desde el inicio. Respecto a la detokenización, como se observa en 16, se realiza directamente una búsqueda en la base de datos, en caso de no encontrar el token ingresado, lanza un error indicando que no existe en la base de datos.

5.2.5. Módulo de DRBG

En la sección 3.3.1 se resume el estándar del National Institute of Standards and Technology (NIST) para la creación de un Deterministic Random Bit Generator (DRBG). En 3.4.5 se muestran dos posibles implementaciones concretas: una basada en una función hash y la otra basada en un cifrado por bloques. En 4.1.2 se explica la distribución de las clases creada para este módulo. En esta sección se muestran los apsectos más significativos de las implementaciones.

Como se muestra en el diagrama 4.6, un generador solamente expone una función pública: `operar` para la generación de bytes; el nombre viene dado por el contrato con la interfaz de una función que recibe enteros y regresa arreglos de bytes. El código fuente 17 muestra esta implementación, la cuál depende de `generarBytes`, que dentro del contexto de un DRBG genérico, es abstracta.

En los códigos fuente 18 y 19 se muestran las dos operaciones más importantes del generador basado en funciones hash: la generación de bytes y una función de generación interna. Como se puede ver en el primero de los códigos, la función de generación de bytes depende de la función interna: esta genera los bytes pedidos y después, la función de generación de bytes, se encarga de actualizar el estado interno.

En los códigos fuente 20 y 21 se muestran las dos operaciones más importantes del generador basado en un cifrado por bloques. La primera se trata de la función de generación de bytes; como se puede observar, su operación es idéntica al modo de operación de contador (sección 2.2.6). La segunda función actualiza el estado cada vez que hay un cambio de semilla.

```
284 /**
285  * Método principal que se encarga de crear el token:
286  * 1. Crea el mBloqueT al concatenar la salida truncada del SHA256 (alimentada
287  *    con mEntradaU) y la representación binaria de mEntradaX.
288  * 2. Realiza al menos una vez los siguientes pasos:
289  *  2.1 Cifrar el mBloqueT mediante AES256 y lo guarda en el mBloqueC.
290  *  2.2 Obtiene la representación decimal de los últimos mLongitudBytesEntradaX
291  *      bits del mBloqueC y los guarda en token.
292  *  2.3 Comprueba que el valor en token sea menor a 10^mNumeroDigitosEntradaX:
293  *      en otras palabras, verifica que tenga mNumeroDigitosEntradaX dígitos.
294  *      En caso de que no lo tenga, usa el método de la caminata cíclica al
295  *      regresar al paso 2.1, pero cifrando el mBloqueC actual.
296  * 3. Verifica si existe el token en la base de datos. Si existe, regresa al
297  *    paso 1, con la misma llave y la misma entrada mEntradaX, pero aumenta
298  *    en uno la entrada mEntradaU.
299  */
300 void AHR::tokenizarHibridamente()
301 {
302   entero limiteS = potencia<entero>(10, mNumeroDigitosEntradaX);
303   entero limiteI = potencia<entero>(10, (mNumeroDigitosEntradaX-1)) - 1;
304
305   /* Primer paso del algoritmo. */
306   crearBloqueT();
307
308   /* Caminata cíclica para obtener un token dentro del dominio del mensaje. */
309   do{
310     /* Segundo paso del algoritmo: cifrar el mBloqueT y ponerlo en mBloqueC. */
311     mCifrador.cifrarBloque(mBloqueT);
312     mBloqueC = mCifrador.obtenerBloqueTCifrado();
313
314     /* Tercer y cuarto paso del algoritmo: verificar que esté dentro del dominio
315      * del mensaje y pasarlo a su forma decimal.
316      */
317     obtenerNumeroC();
318     memcpy(mBloqueT, mBloqueC, M);
319   }while(mToken >= limiteS || mToken <= limiteI);
320
321   /* Quinto paso del algoritmo: revisar si existe el token generado.*/
322   if(existeToken())
323   {
324     /* Aumentar en uno la mEntradaU y volver a correr el algoritmo.*/
325     mEntradaU += 1;
326     tokenizarHibridamente();
327   }
328 }
```

Código fuente 14: Tokenización mediante AHR.

```
193 /**
194   * Este método se encarga del primer paso del algoritmo: obtener el mBloqueT
195   * que será utilizado como entrada para el cifrador por bloques. Primero
196   * obtiene la salida de la función pública (SHA256 en este caso) de la entrada
197   * adicional mEntradaU. Luego llama a la función obtenerBitsX para llenar los
198   * últimos mLongitudBytesEntradaX bloques del mBloqueT y, finalmente, le
199   * concatena al inicio los bytes más significativos de la salida del SHA para
200   * completar el tamaño de bloque del cifrador.
201   */
202 void AHR::crearBloqueT()
203 {
204   string cadenaU  = to_string(mEntradaU);
205   SHA256 funcionF;
206   byte salidaFCompleta[SHA256::DIGESTSIZE];
207 
208   /* Obtener valor SHA256 de la entrada adicional**/
209   funcionF.Update(
210     reinterpret_cast<const unsigned char*>(cadenaU.c_str()),
211     cadenaU.length()
212   );
213   funcionF.Final(salidaFCompleta);
214 
215   obtenerBitsX();
216 
217   int j = 0;
218   for(int i=0; i < M - mLongitudBytesEntradaX; i++)
219   {
220     mBloqueT[i] = salidaFCompleta[j];
221     j++;
222   }
223 }
```

Código fuente 15: Primer paso para la tokenización con AHR.

```
360 /**
361  * Este método se encarga de regresar el PAN que corresponde al
362  * token dado en el argumento. Como este es un algoritmo irreversible,
363  * busca en la base de datos el token dado y regresa el PAN asociado.
364  * En caso de no existir, lanza una excepción.
365  */
366 ArregloDeDigitos AHR::detokenizar(const ArregloDeDigitos& token)
367 {
368   Registro informacion = mAccesoADatos->buscarPorToken(token);
369   if (informacion.obtenerPAN() == Arreglo<int>{})
370     throw TokenInexistente{"El token no está en la base de datos."};
371   return informacion.obtenerPAN();
372 }
```

Código fuente 16: Primer paso para la detokenización con AHR.

```
66 /**
67  * La fuerza de seguridad de la salida es el mínimo de entre la longitud
68  * de la salida y el nivel de la instanciación.
69  *
70  * \param entrada El primer elemento indica la longitud deseada para la salida.
71  *                El segundo elemento (opcional) indica la fuera de seguridad
72  *                deseada. Si no se da, se supone la de la instancia.
73  *
74  * \throw FuerzaNoSoportada Si la fuerza solicitada es mayor que el nivel de
75  *                          seguridad provisto por la instanciación.
76  */
77
78 Arreglo<unsigned char> DRBG::operar(const vector<entero>& entrada)
79 {
80   if (entrada.size() == 2)
81     if (entrada[1] > static_cast<unsigned int>(mNivelDeSeguridad))
82       throw FuerzaNoSoportada{
83         "La instanciación no soporta ese nivel de seguridad."};
84
85   mContadorDePeticiones++;
86   if (mContadorDePeticiones > mMaximoDePeticiones)
87     cambiarSemilla();
88
89   return generarBytes(entrada[0]);
90 }
```

Código fuente 17: Función pública de generadores pseudoaleatorios

```
117 /**
118  * Define el proceso de generación de bytes pseudoaleatorios. Corresponde a la
119  * sección 10.1.1.4 del estándar. El trabajo para generar los nuevos bytes se
120  * encuentra en funcionDeGeneracion (es la misma división del documento, pero
121  * con nombres distintios), también se genera un nuevo estado para la semilla.
122  *
123  * \return Arreglo con bytes aleatorios.
124  */
125
126 Arreglo<unsigned char> HashDRBG::generarBytes(
127   entero longitud           /**< Número de bytes deseados. */
128 )
129 {
130   auto resultado = funcionDeGeneracion(static_cast<unsigned int>(longitud));
131   auto temporal = hash(Arreglo<unsigned char>{3} || mSemilla);
132   mSemilla = mSemilla + temporal
133     + mConstanteSemilla + Arreglo<unsigned char>(mContadorDePeticiones);
134   return resultado;
135 }
```

Código fuente 18: Función de generación de bytes de hash DRBG

```
166 /**
167  * Función ocupada por generarBytes para generar el número de bytes aleatorios
168  * usando la función hash interna y el valor de la semilla.
169  */
170
171 Arreglo<unsigned char> HashDRBG::funcionDeGeneracion(
172   unsigned int longitudDeSalida     /**< Longitud de la salida. */
173 )
174 {
175   Arreglo<unsigned char> resultado;
176   unsigned int longitud = mFuncionHash->DigestSize();
177   unsigned int numeroDeBloques = ceil(
178     static_cast<double>(longitudDeSalida) / longitud);
179   Arreglo<unsigned char> datos {mSemilla};
180   for (unsigned int i = 0; i < numeroDeBloques; i++)
181   {
182     resultado = resultado || hash(datos);
183     datos = move(datos || Arreglo<unsigned char>{1});
184   }
185   return (resultado / Arreglo<unsigned int>{longitudDeSalida})[0];
186 }
```

Código fuente 19: Función de generación interna de hash DRBG

```
82 /**
83  * Función generadora de bytes aleatorios (llamada desde clase
84  * abstracta).
85  *
86  * \return Arreglo con el número de bytes solicitados.
87  */
88
89 Arreglo<unsigned char> CTRDRBG::generarBytes(
90   entero longitud         /**< Longitud de salida. */
91 )
92 {
93   unsigned int longitudLocal = static_cast<unsigned int>(longitud);
94   Arreglo<unsigned char> resultado;
95   while (resultado.obtenerNumeroDeElementos() < longitudLocal)
96   {
97     mSemilla = ((mSemilla + 1ull) / Arreglo<unsigned int>{mLongitudBloque})[0];
98     resultado = resultado || cifrarBloque(mSemilla);
99   }
100  return (resultado / Arreglo<unsigned int>{longitudLocal})[0];
101 }
```

Código fuente 20: Función de generación de bytes de CTR DRBG

```
103 /**
104  * Operación de actualización de estado. En el estándar corresponde a
105  * CTR_DRBG_Update (sección 10.2.1.2). Calcula un nuevo valor para
106  * la semilla y para la llave del cifrador por bloques.
107  */
108
109 void CTRDRBG::actualizarEstado(
110   const Arreglo<unsigned char>& entrada   /**< Arreglo de entrada (entropía). */
111 )
112 {
113   Arreglo<unsigned char> temporal;
114   while (temporal.obtenerNumeroDeElementos() < mLongitudSemilla)
115   {
116     mSemilla = ((mSemilla + 1ull) / Arreglo<unsigned int>{mLongitudBloque})[0];
117     temporal = temporal || cifrarBloque(mSemilla);
118   }
119   temporal = (temporal / Arreglo<unsigned int>{mLongitudSemilla})[0];
120   temporal = temporal ^ entrada;
121   mLlave = (temporal / Arreglo<unsigned int>{mLongitudLlave})[0];
122   mSemilla = (temporal / Arreglo<unsigned int>{mLongitudSemilla})[0];
123 }
```

Código fuente 21: Función de actualización de estado de CTR DRBG

5.3. Resultados

5.3.1. Resultados de las pruebas estadísticas a los DRGB

En esta sección se pretende demostrar la aleatoriedad de los Deterministic Random Bit Generator (DRBG) que fueron implementados, esto por medio del conjunto de pruebas del National Institute of Standards and Technology (NIST), pruebas de las que se habla más a fondo en la sección B.

En las siguientes 6 tablas (5.1, 5.2, 5.3, 5.4, 5.5 y 5.6) se observan los resultados obtenidos para las 15 pruebas estadísticas descritas en [38] para 3 tipos distintos de DRBG (uno basado en Advanced Encryption Standard (AES) y otros dos basados en Secure Hash Algorithm (SHA)) en cada uno de los niveles de seguridad que se tienen (112, 128, 192 y 256).

Para poder comprender la información más claramente, es necesario establecer los siguientes puntos:

- Las pruebas se realizaron con 10 millones de bits obtenidos por cada tipo de DRBG.
- Cada prueba realizada consta de 20 ejecuciones, y cada ejecución usa medio millón de bits.
- Se puede resumir que el valor de P va de 0 a 1, y que mientras más grande sea este valor, el DRBG probado es más aleatorio.
- Las pruebas de sumas acumulativas, de coincidencia sin superposición, de entropía aproximada, de excursiones aleatorias, y su variante constan de varias subpruebas.
- En las tablas, en las columnas de *proporción*, se pone el número de pruebas pasadas con respecto al

número de pruebas (subpruebas si existen) que se hicieron en total.

- Posterior a las columnas de *proporción* se pone un ✓ en caso de que el DRBG aprobara de forma general la prueba o un ✕ en el caso contrario.

Pruebas estadísticas para el DRBG basado en cifrados por bloque (AES)						
Nivel de seguridad	**112**			**128**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.991468	19/20	✓	0.437274	20/20	✓
De frecuencia en un bloque	0.017912	20/20	✓	0.048716	20/20	✓
De sumas acumulativas	0.872861	38/40	✓	0.356491	40/40	✓
De carreras	0.964295	20/20	✓	0.350485	20/20	✓
De la carrera más larga en un bloque	0.637119	20/20	✓	0.213309	20/20	✓
Del rango de matriz binaria	0.964295	20/20	✓	0.911413	20/20	✓
Espectral	0.637119	20/20	✓	0.213309	20/20	✓
De coincidencia sin superposición	0.475566	2925/2960	✓	0.513657	2930/2960	✓
De coincidencia con superposición	0.964295	20/20	✓	0.637119	20/20	✓
Estadística universal de Maurer	0.066882	20/20	✓	0.834308	19/20	✓
De entropía aproximada	0.739918	20/20	✓	0.834308	20/20	✓
De excursiones aleatorias	———	70/72	✓	0.485432	96/96	✓
Variante de excursiones aleatorias	———	159/162	✓	0.283322	215/216	✓
Serial	0.588596	40/40	✓	0.577037	40/40	✓
De complejidad lineal	0.122325	20/20	✓	0.090936	20/20	✓

Tabla 5.1: Resultado de las pruebas estadísticas del DRBG basado en cifrados por bloque (AES) para los niveles de seguridad de 112 y 128.

Pruebas estadísticas para el DRBG basado en cifrados por bloque (AES)						
Nivel de seguridad	**192**			**256**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.964295	20/20	✓	0.162606	20/20	✓
De frecuencia en un bloque	0.739918	20/20	✓	0.162606	20/20	✓
De sumas acumulativas	0.637119	40/40	✓	0.325292	40/40	✓
De carreras	0.834308	19/20	✓	0.534146	20/20	✓
De la carrera más larga en un bloque	0.534146	19/20	✓	0.739918	20/20	✓
Del rango de matriz binaria	0.350485	20/20	✓	0.122325	20/20	✓
Continúa en siguiente página						

Continuación						
Nivel de seguridad	**192**			**256**		
Prueba	Valor P	Proporción		Valor P	Proporción	
Espectral	0.637119	20/20	✓	0.637119	19/20	✓
De coincidencia sin superposición	0.496882	2925/2960	✓	0.495414	2923/2960	✓
De coincidencia con superposición	0.122325	20/20	✓	0.437274	19/20	✓
Estadística universal de Maurer	0.350485	20/20	✓	0.437274	20/20	✓
De entropía aproximada	0.834308	20/20	✓	0.534146	20/20	✓
De excursiones aleatorias	0.413551	87/88	✓	0.393867	88/88	✓
Variante de excursiones aleatorias	0.416464	198/198	✓	0.399044	197/198	✓
Serial	0.404927	39/40	✓	0.506506	40/40	✓
De complejidad lineal	0.350485	20/20	✓	0.534146	20/20	✓

Tabla 5.2: Resultado de las pruebas estadísticas del DRBG basado en cifrados por bloque (AES) para los niveles de seguridad de 192 y 256.

Pruebas estadísticas para el DRBG basado en funciones hash (SHA256)						
Nivel de seguridad	**112**			**128**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.437274	20/20	✓	0.739918	20/20	✓
De frecuencia en un bloque	0.637119	20/20	✓	0.637119	19/20	✓
De sumas acumulativas	0.328236	40/40	✓	0.264105	40/40	✓
De carreras	0.006196	19/20	✓	0.739918	20/20	✓
De la carrera más larga en un bloque	0.275709	20/20	✓	0.739918	20/20	✓
Del rango de matriz binaria	0.350485	20/20	✓	0.637119	19/20	✓
Espectral	0.350485	19/20	✓	0.008879	20/20	✓
De coincidencia sin superposición	0.521362	2921/2960	✗	0.480227	2937/2960	✓
De coincidencia con superposición	0.637119	20/20	✓	0.911413	19/20	✓
Estadística universal de Maurer	0.637119	20/20	✓	0.048716	20/20	✓
De entropía aproximada	0.275709	20/20	✓	0.437274	20/20	✓
De excursiones aleatorias	0.407729	102/104	✓	———	62/64	✓
Variante de excursiones aleatorias	0.376856	232/234	✓	———	142/144	✓
Serial	0.404927	40/40	✓	0.637032	40/40	✓
De complejidad lineal	0.637119	20/20	✓	0.637119	20/20	✓
Continúa en siguiente página						

Continuación						
Nivel de seguridad	**112**			**128**		
Prueba	Valor P	Proporción		Valor P	Proporción	

Tabla 5.3: Resultado de las pruebas estadísticas del DRBG basado en funciones hash (SHA256) para los niveles de seguridad de 112 y 128.

En la tabla 5.3 se observa que la única prueba que no pasó el DRBG basado en SHA256 con nivel de seguridad de 112, fue la de coincidencias sin superposición, situación que viene de no haberse aprobado más del 90 % de una de las subpruebas.

Pruebas estadísticas para el DRBG basado en funciones hash (SHA256)						
Nivel de seguridad	**192**			**256**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.739918	20/20	✓	0.006196	20/20	✓
De frecuencia en un bloque	0.437274	20/20	✓	0.162606	20/20	✓
De sumas acumulativas	0.899301	40/40	✓	0.688519	40/40	✓
De carreras	0.911413	17/20	×	0.350485	20/20	✓
De la carrera más larga en un bloque	0.834308	20/20	✓	0.911413	20/20	✓
Del rango de matriz binaria	0.911413	20/20	✓	0.122325	20/20	✓
Espectral	0.035174	20/20	✓	0.437274	19/20	✓
De coincidencia sin superposición	0.504889	2937/2960	✓	0.507982	2928/2960	✓
De coincidencia con superposición	0.739918	20/20	✓	0.834308	20/20	✓
Estadística universal de Maurer	0.090936	20/20	✓	0.637119	20/20	✓
De entropía aproximada	0.739918	20/20	✓	0.637119	20/20	✓
De excursiones aleatorias	0.667135	78/80	✓	———	72/72	✓
Variante de excursiones aleatorias	0.498556	174/180	✓	———	162/162	✓
Serial	0.637032	39/40	✓	0.534146	38/40	✓
De complejidad lineal	0.911413	20/20	✓	0.350485	19/20	✓

Tabla 5.4: Resultado de las pruebas estadísticas del DRBG basado en funciones hash (SHA256) para los niveles de seguridad de 192 y 256.

En la tabla 5.4 se observa que el DRBG basado en SHA256 con nivel de seguridad de 192, no aprobó de forma general la prueba de carreras, esto por no llegar a las 18 ejecuciones aprobadas de 20.

Pruebas estadísticas para el DRBG basado en funciones hash (SHA512)						
Nivel de seguridad	**112**			**128**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.213309	20/20	✓	0.637119	20/20	✓
De frecuencia en un bloque	0.834308	20/20	✓	0.437274	20/20	✓
De sumas acumulativas	0.911413	40/40	✓	0.637032	40/40	✓
De carreras	0.534146	19/20	✓	0.739918	20/20	✓
De la carrera más larga en un bloque	0.911413	20/20	✓	0.122325	19/20	✓
Del rango de matriz binaria	0.834308	20/20	✓	0.834308	19/20	✓
Espectral	0.637119	20/20	✓	0.739918	19/20	✓
De coincidencia sin superposición	0.512787	2934/2960	✓	0.475790	2937/2960	✓
De coincidencia con superposición	0.437274	20/20	✓	0.275709	20/20	✓
Estadística universal de Maurer	0.739918	20/20	✓	0.275709	20/20	✓
De entropía aproximada	0.834308	20/20	✓	0.739918	20/20	✓
De excursiones aleatorias	———	72/72	✓	0.442340	77/80	✓
Variante de excursiones aleatorias	———	161/162	✓	0.523098	180/180	✓
Serial	0.493802	40/40	✓	0.585633	40/40	✓
De complejidad lineal	0.534146	20/20	✓	0.350485	20/20	✓

Tabla 5.5: Resultado de las pruebas estadísticas del DRBG basado en funciones hash (SHA512) para los niveles de seguridad de 112 y 128.

Pruebas estadísticas para el DRBG basado en funciones hash (SHA512)						
Nivel de seguridad	**192**			**256**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De frecuencia	0.162606	20/20	✓	0.534146	20/20	✓
De frecuencia en un bloque	0.739918	20/20	✓	0.739918	19/20	✓
De sumas acumulativas	0.336147	40/40	✓	0.637032	40/40	✓
De carreras	0.534146	20/20	✓	0.637119	20/20	✓
De la carrera más larga en un bloque	0.739918	20/20	✓	0.834308	20/20	✓
Del rango de matriz binaria	0.534146	20/20	✓	0.739918	20/20	✓
Espectral	0.066882	19/20	✓	0.275709	19/20	✓
De coincidencia sin superposición	0.489630	2934/2960	✓	0.484244	2929/2960	✓
De coincidencia con superposición	0.350485	20/20	✓	0.437274	20/20	✓
Estadística universal de Maurer	0.834308	20/20	✓	0.739918	19/20	✓
Continúa en siguiente página						

Continuación						
Nivel de seguridad	**192**			**256**		
Prueba	Valor P	Proporción		Valor P	Proporción	
De entropía aproximada	0.275709	20/20	✓	0.964295	20/20	✓
De excursiones aleatorias	0.503591	104/104	✓	0.481113	78/80	✓
Variante de excursiones aleatorias	0.235885	234/234	✓	0.572188	178/180	✓
Serial	0.688519	39/40	✓	0.555009	37/40	✓
De complejidad lineal	0.637119	19/20	✓	0.739918	20/20	✓

Tabla 5.6: Resultado de las pruebas estadísticas del DRBG basado en funciones hash (SHA512) para los niveles de seguridad de 192 y 256.

Con base en los resultados de las pruebas estadísticas, se puede decir que los DRBG implementados son aleatorios. Aunque hay que tomar en cuenta que el número de ejecuciones que se realizaron para las pruebas (20) es aun un número pequeño, pero hacer un mayor número de pruebas consumiría mas recursos de los que se tienen.

5.3.2. Comparación de desempeño

Como se indicó en la introducción de este trabajo, la tokenización digital es un proceso relativamente nuevo y la mayoría de los proveedores de este servicio aprovechan la confusión y desinformación existente para atraer a clientes potenciales, pues pocos comparten la manera en que crean sus TOKENS. Esta sección busca resarcir un poco el daño, presentado una comparación entre los tiempos de respuesta de tokenización y detokenización de los algoritmos implementados.

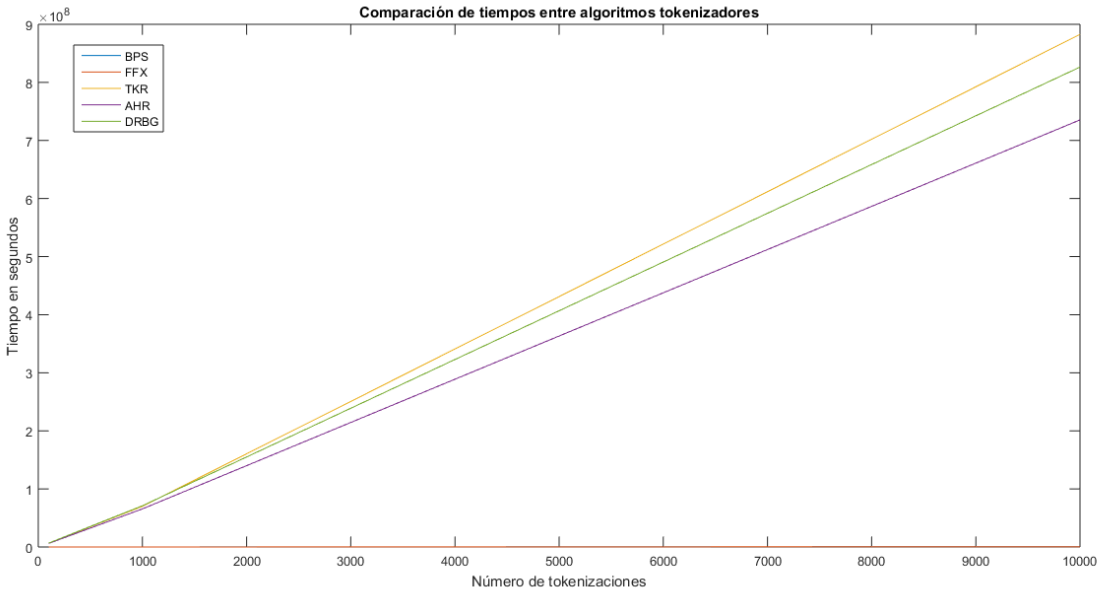
En la tabla 5.7 se puede observar el tiempo promedio que le toma a cada algoritmo hacer el número de tokenizaciones o detokenizaciones señaladas en el encabezado. Los algoritmos están divididos en reversibles y no reversibles, pues, dadas sus características, no sería muy parejo compararlos entre sí; por ejemplo, basta hacer una consulta en la base de datos para hacer la detokenización en los algoritmos irreversibles; mientras que en los reversibles hay que hacer el proceso inverso, o uno equivalente, para poder obtener de nuevo el PERSONAL ACCOUNT NUMBER (PAN).

Los datos de la tabla 5.7 fueron utilizados para crear una serie de gráficas: en la figura 5.1 se comparan los tiempos de tokenización y detokenización de todos los algoritmos, tanto reversibles como irreversibles; la velocidad de los algoritmos reversibles es notoria, en especial al momento de la operación de tokenización. Posteriormente, en la figura 5.2, se compararon solo los algoritmos reversibles (BPS y FFX), siendo el segundo el más rápido de los dos; especialmente cuando se realizan operaciones de detokenización. Finalmente, se compararon los tiempos para los algoritmos irreversibles en las gráficas de la figura 5.3:

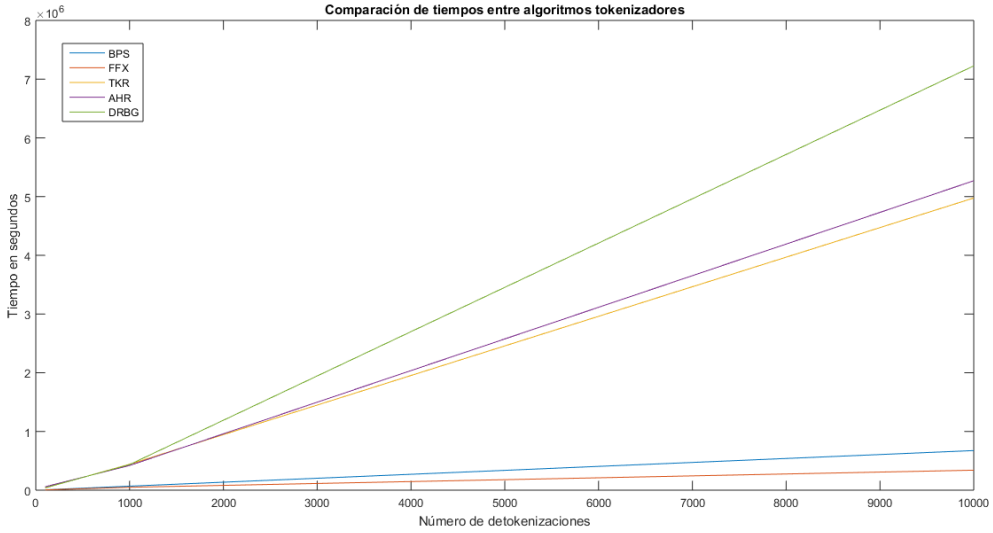
		100 oper.		1K oper.		10K oper.	
		Tok.	Detok.	Tok.	Detok.	Tok.	Detok.
Reversibles	BPS	7.247 ms	6.990 ms	68.514 ms	68.566 ms	714.336 ms	674.571 ms
	FFX	5.627 ms	5.516 ms	49.738 ms	49.550 ms	487.529 ms	340.749 ms
Irreversibles	TKR	6.573 s	37.623 ms	70.116 s	441.815ms	882.695 s	4.976 s
	AHR	6.053 s	58.814 ms	65.631 s	420.729ms	735.405 s	5.269 s
	DRBG	6.718 s	40.265 ms	71.082 s	436.753ms	826.403 s	7.226 s

Tabla 5.7: Comparación de tiempos de tokenización de los algoritmos implementados.

durante las operaciones de tokenización, aunque están relativamente cerrados los tiempos, AHR queda como el más rápido de los tres, mientras que DRBG queda en segundo lugar; es menester recordar porqué estos algoritmos toman más tiempo: AHR utiliza CIFRADO DE CAMINATA CÍCLICA, y, al generar un TOKEN, consulta la base de datos, pues permite tener más de un TOKEN por PAN; además, tanto TKR2 como DRBG necesitan bits pseudoaleatorios, y si el generador de bits pseudoaleatorios tarda en obtenerlos, los algoritmos se ven afectados; finalmente, en las operaciones de detokenización, TKR2 es el más rápido de todos, mientras que DRBG queda como el más lento.

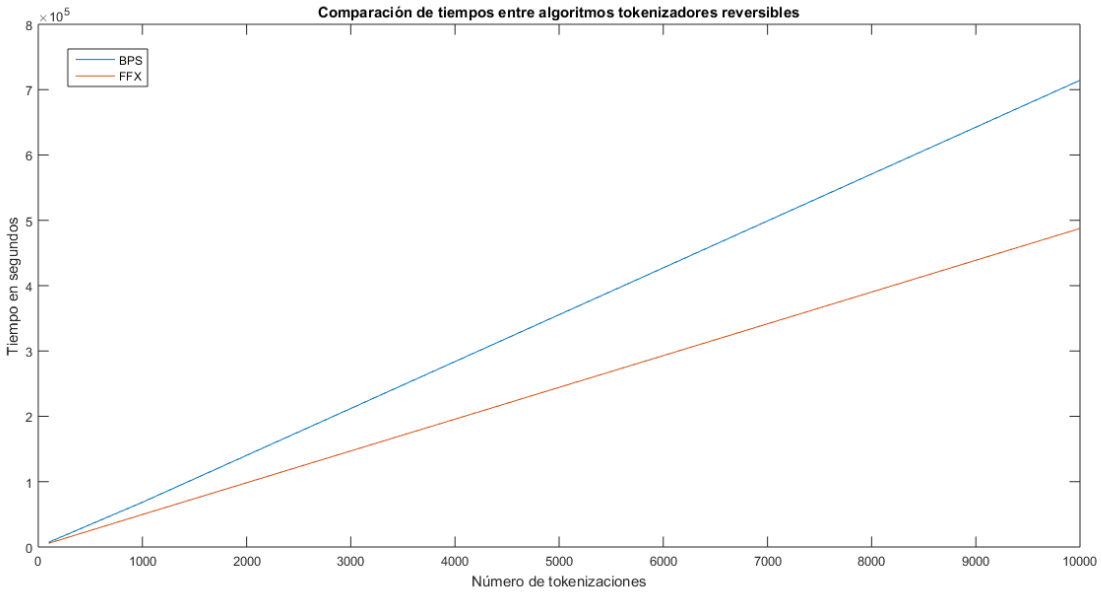


(a) Tiempos de tokenización.

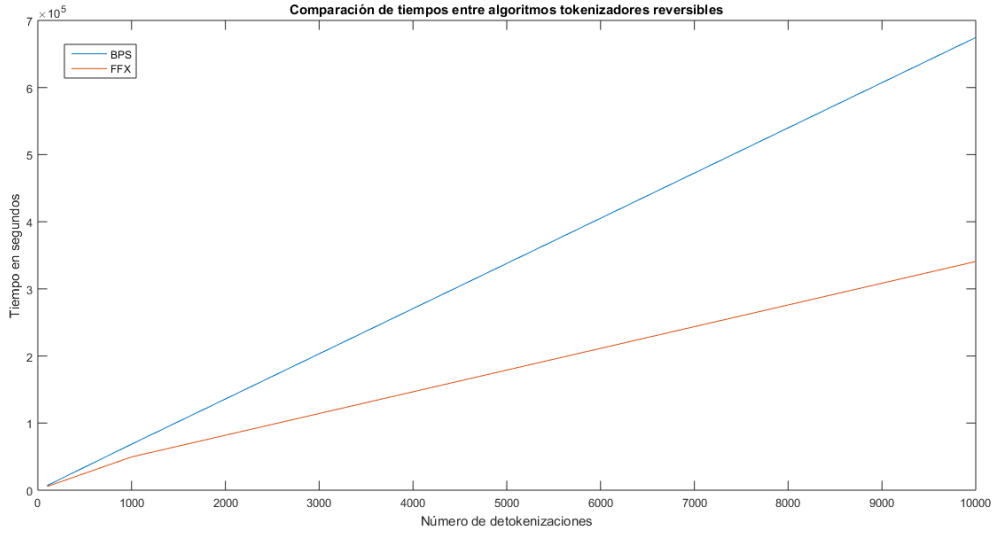


(b) Tiempos de detokenización.

Figura 5.1: Comparación de tiempos entre algoritmos tokenizadores.

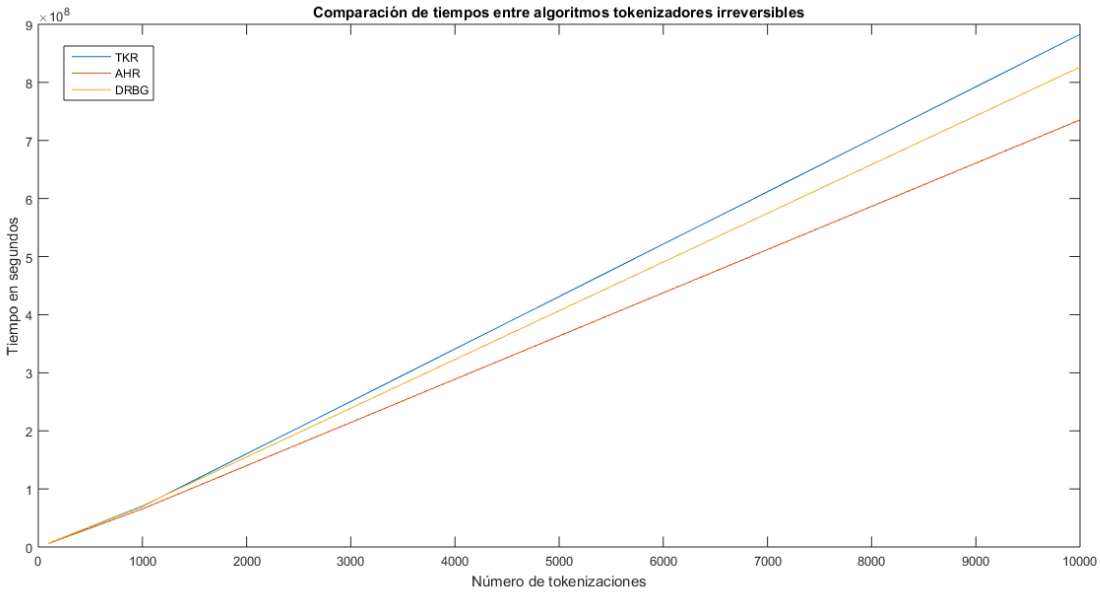


(a) Tiempos de tokenización.

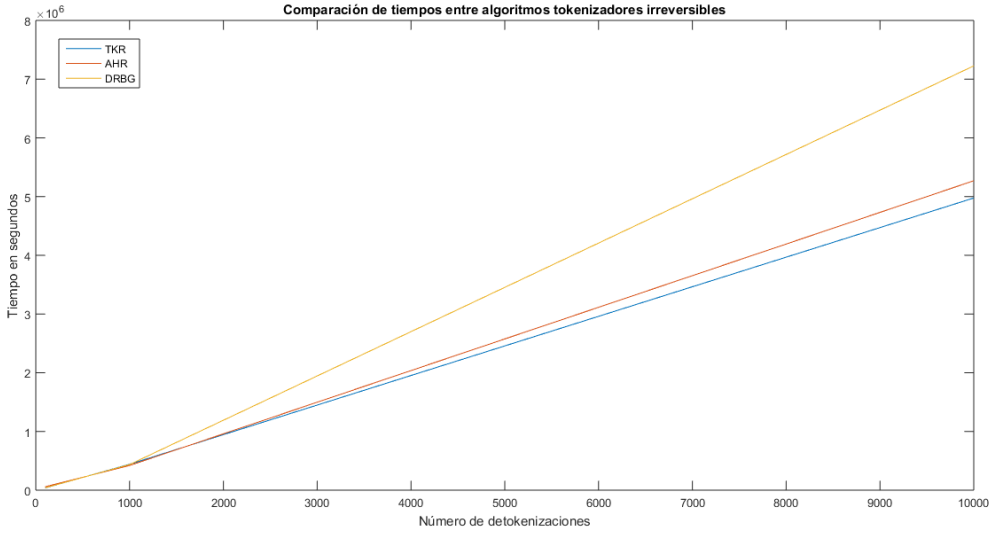


(b) Tiempos de detokenización.

Figura 5.2: Comparación de tiempos entre algoritmos tokenizadores reversibles.



(a) Tiempos de tokenización.



(b) Tiempos de detokenización.

Figura 5.3: Comparación de tiempos entre algoritmos tokenizadores irreversibles.

Capítulo 6

Conclusiones

«*Cada día, Sancho, te vas haciendo menos simple y más discreto.*»
Don Quijote de la Mancha
Miguel de Cervantes.

«*Those who believe in telekinetics, raise my hand.*»
Kurt Vonnegut.

En este curso se da por terminado el primero de tres módulos que componen al trabajo: un programa generador de *tokens*. Aunque de enunciado fácil, el desarrollo de este programa encierra toda la variedad de temas que se ha visto hasta el momento: estudio de los aspectos de criptografía involucrados, investigación de los algoritmos generadores de *tokens* existentes, análisis de los estándares y recomendaciones oficiales relacionados, comparaciones de desempeño, pruebas de aleatoriedad, etcétera. Con esto se cumple, en gran medida, el objetivo general del trabajo: la implementación de algoritmos generadores de *tokens* con el propósito de proveer confidencialidad a los datos de tarjetas bancarias. También se cubre en su totalidad el primero de los objetivos específicos: la revisión de diversos algoritmos para la generación de *tokens*.

Los planes a futuro, para trabajo terminal II, incluyen el desarrollo de los dos módulos restantes: la interfaz en red para utilizar el programa tokenizador y la tienda en línea como cliente de la interfaz en red. Esto permitirá que lo desarrollado en este semestre llegue hasta los usuarios finales, esto es, hasta los clientes de las tiendas en línea. De esta forma se cubren los otros dos objetivos específicos del trabajo.

Apéndices

Apéndice A

Administración de llaves

«—No está mal, no está mal —decía Diotallevi—. Encontrar la verdad reconstruyendo exactamente un texto mendaz.»
El péndulo de Foucault
Umberto Eco.

A.1. Tipos de llaves

En [36], el National Institute of Standards and Technology (NIST) divide a las llaves criptográficas en 19 categorías, las cuales surgen de la combinación de la naturaleza de la llave (simétrica, pública o privada) con la funcionalidad que se le da (firmas, autenticación, cifrado, entre otras). En la tabla A.1 se muestran dichas categorías: se marca con ✓ las combinaciones que tienen sentido práctico; por ejemplo, para firmar (y posteriormente validar) solamente se ocupan las llaves de un esquema asimétrico (i. e. públicas y privadas).

Para firmas: llaves que son usadas en algoritmos asimétricos para generar firmas digitales con posibles implicaciones a largo plazo. Cuando son manejadas correctamente, este tipo de llaves está diseñado para proveer autenticación de origen, autenticación de integridad y soporte para el no repudio.

Para autenticación: son usadas para proporcionar garantías sobre la identidad de un fuente generadora (i. e. autenticación de origen).

Para cifrado de datos: usadas en esquemas simétricos para proveer de confidencialidad a la información (esto es, para cifrar la información). Como se tratan de algoritmos simétricos (o de llave secreta), la misma llave se usa para quitar la confidencialidad (para descifrar).

Como envoltura de otras llaves: también llamadas llaves cifradoras de llaves (*key-encrypting keys*), son usadas para proteger otras llaves usando algoritmos simétricos. Dependiendo del algoritmo con el que se use la llave, esta también puede ser usada para proveer de validaciones de integridad.

Para generadores aleatorios: llaves usadas para generar números o bits aleatorios.

Como llaves maestra: una llave maestra simétrica se usa para derivar otras llaves simétricas (p. ej. llaves de cifrado, o de envoltura). También se conoce como llave de derivación de llaves (*key-derivation key*).

Para el transporte de llaves: llaves utilizadas en esquemas asimétricos para proteger la comunicación en un proceso de acuerdo de llaves. Se usan para proteger otras llaves (p. ej. de cifrado, de envoltura) o información relacionada (p. ej. vectores de inicialización).

Para el acuerdo de llaves: utilizadas en algoritmos de acuerdo de llaves para establecer otras llaves. Generalmente funcionan en plazos muy largos.

Efímeras para el acuerdo de llaves: llaves de muy corto plazo que son usadas una sola vez en un proceso de establecimiento de otras llaves. En algunos casos la llave efímera puede ser usada más de una vez, pero dentro de la misma transacción.

De autorización: usadas para proveer y verificar los privilegios de una entidad. En un esquema simétrico, la llave es conocida tanto por la entidad que provee acceso, como por la que desea acceder.

		Naturaleza		
		Simétrica	Pública	Privada
Funcionalidad	Para firmas		✓	✓
	Para autenticación	✓	✓	✓
	Para cifrado de datos	✓		
	Como envoltura de otras llaves	✓		
	Para generadores aleatorios	✓		
	Como llave maestra	✓		
	Para el transporte de llaves		✓	✓
	Para el acuerdo de llaves	✓	✓	✓
	Efímeras para el acuerdo de llaves		✓	✓
	De autorización	✓	✓	✓

Tabla A.1: Clasificación de llaves criptográficas

A.2. Usos de llaves

En general, las llaves se deben de utilizar para un solo propósito: el uso de la misma llave para dos operaciones criptográficas puede debilitar la seguridad provista por ambas; al limitar el uso de una llave se limita el nivel de riesgo de que esta se vea comprometida; algunos usos de llaves interfieren unos con otros.

La regla anterior no se extiende a situaciones en donde el mismo proceso provea de múltiples servicios. Por ejemplo, cuando una sola firma digital provee de autenticación de integridad y de autenticación de origen, o cuando una sola llave simétrica es usada para cifrar y para autenticar en una sola operación.

A.3. Criptoperiodos

Un criptoperiodo es el rango de tiempo durante el cual una llave es válida. Algunas de las funciones de los criptoperiodos son:

- Limitar el total de daños si una sola llave se ve comprometida.
- Limitar el uso de un algoritmo en particular (*particular* en el sentido de la instancia misma del algoritmo).
- Limitar el tiempo disponible para intentos de ataques sobre los mecanismos que protegen a la llave del acceso sin autorización.
- Limitar el periodo en el cual la información puede verse comprometida por revelaciones inadvertidas de llaves a entidades sin autorización.
- Limitar el tiempo disponible para ataque criptoanalíticos.

A continuación se enlistan los principales factores a tomar en cuenta al momento de establecer un criptoperiodo:

- La fuerza del mecanismo criptográfico (el algoritmo, la FUERZA EFECTIVA de la llave, el tamaño de bloque, el modo de operación, etc.).
- El entorno de operacion (p. ej. un lugar de acceso restringido, una terminal pública).
- El volumen de información transmitida, o el número de transacciones.
- La función de seguridad (cifrado de datos, firma digital, derivación de llaves, etc.).
- El método de entrada de la llave (por teclado, a nivel de sistema).
- El número de nodos en una red que comparten la misma llave.
- El número de copias de la llave distribuidas.
- Rotaciones de personal.
- La amenaza de los adversarios (¿de quién se está protegiendo la información?, ¿cuáles son sus capacidades?)
- La amenza de avances tecnológicos (nuevos límites para el problema del logaritmo discreto, computadoras cuánticas).

La duración de los criptoperiodos también debe tomar en cuenta cuáles son los riesgos de las actualizaciones de llaves. En general, entre más cortos sean los criptoperiodos, mejor; sin embargo, si los mecanismos de distribución de llaves son manuales y propensos a errores humanos, entonces el riesgo se invierte. En estos casos (en especial cuando se utiliza CRIPTOGRAFÍA FUERTE) resulta mejor tener pocas y bien controladas distribuciones manuales, a que estas sean frecuentes y mal controladas.

Las consecuencias de una exposición se miden de acuerdo a qué tan sensible es la información, qué tan críticos son los procesos protegidos, y qué tan alto es el costo de recuperación en caso de exposición. La sensibilidad se refiere al periodo de tiempo durante el cual la información debe estar protegida (5 segundos, 5 minutos, 5 horas o 5 años) y a las consecuencias potenciales en caso de pérdida de protección. En general, entre más sensible sea la información protegida, el criptoperiodo debe ser menor (sin llegar a que sea contraproducente).

Algunos otros factores a tomar en cuenta incluyen el uso de las llaves y el costo de actualizaciones. En cuanto al uso de las llaves, generalmente se hace distinción en cuanto a si la información protegida solamente se está transfiriendo, o si se va a almacenar; en general los criptoperiodos son más largos en caso de almacenamiento, dado el costo que puede representar el recifrado de toda una base de datos. Esto está

Tipo de llave	**Criptoperiodo**	
	Periodo de uso de emisor (PE)	**Periodo de uso de receptor (PR)**
1.- Llave privada para firma	1 a 3 años	-
2.- Llave pública para verificación de firma	Depende del tamaño de la llave	
3.- Llave simétrica para autenticación	≤ 2 años	$\leq PE + 3$ años
4.- Llave privada para autenticación	1 a 2 años	
5.- Llave pública para autenticación	1 a 2 años	
6.- Llave simétrica para cifrado de datos	≤ 2 años	$\leq PE + 3$ años
7.- Llave simétrica como envoltura	≤ 2 años	$\leq PE + 3$ años
8.- Llave simétrica para generadores aleatorios	ver SP800-90	-
9.- Llave simétrica maestra	alrededor de 1 año	-
10.- Llave privada para transporte	≤ 2 años	
11.- Llave pública para transporte	1 a 2 años	
12.- Llave simétrica para acuerdo	1 a 2 años	
13.- Llave privada para acuerdo	1 a 2 años	
14.- Llave pública para acuerdo	1 a 2 años	
15.- Llave privada efímera para acuerdo	Solo 1 transacción	
16.- Llave pública efímera para acuerdo	Solo 1 transacción	
17.- Llave simétrica para autorización	≤ 2 años	
18.- Llave privada para autorización	≤ 2 años	
19.- Llave pública para autorización	≤ 2 años	

Tabla A.2: Criptoperiodos sugeridos por tipo de llave

relacionado con el costo de las actualizaciones: cuando el volumen de información protegida es demasiado grande o cuando esta se encuentra distribuida en distintos puntos geográficos, el costo de un cambio de llaves puede ser demasiado elevado.

La tabla A.2 muestra las sugerencias del National Institute of Standards and Technology (NIST) en [36] en cuanto a la duración de los criptoperiodos según cada tipo de llave.

A.4. Estados de llaves y transiciones

En la figura A.1 se muestra un diagrama de estados con los posibles comportamientos de una llave criptográfica. Un criptoperiodo inicia al entrar en el estado «activo» (transición 4) y termina al llegar al estado «destruido». En el caso de esquemas asimétricos, las transiciones se aplican a ambos pares de llaves. Se debe llevar un registro de la fecha y hora (y en algunos casos de las razones) de cualquier cambio de estado.

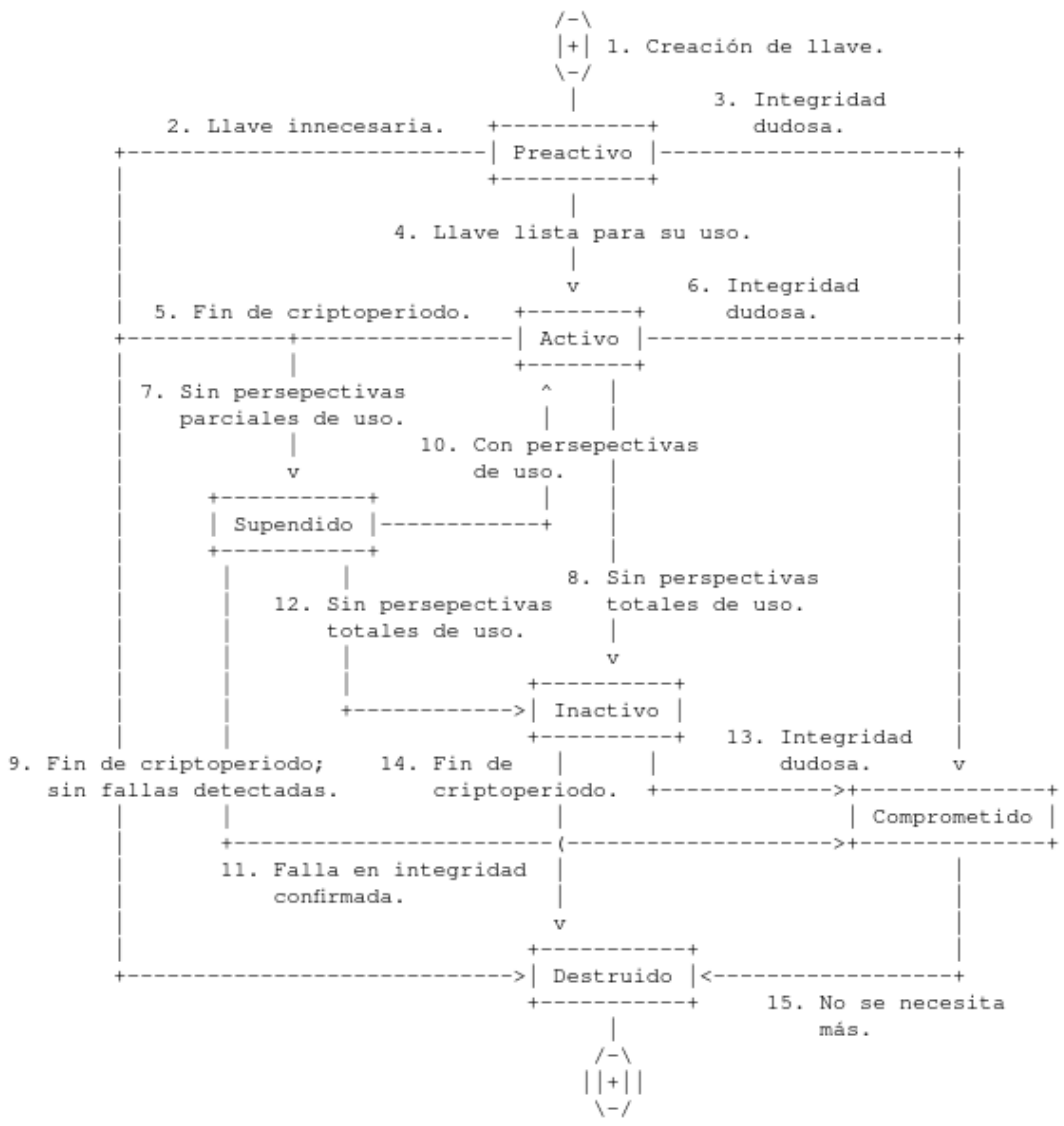


Figura A.1: Diagrama de estado de llaves criptográficas.

Hay varias posibles razones por las que se puede llegar a un estado «suspendido»: la entidad dueña de una firma digital no se encuentra disponible; se sospecha que la integridad de la llave está comprometida, por lo que se pasa a este estado en lo que se investiga más en profundidad; entre otros. Una llave que está en este estado («suspendido») no debe ser usada bajo ninguna circunstancia.

Las llaves que se encuentran en el estado «inactivo» no deben ser usadas para aplicar protección criptográfica, pero en algunos casos, pueden ser usadas para procesar información protegida con criptografía. Por ejemplo, las llaves simétricas usadas para autenticación, cifrado de datos o envoltura de llaves, pueden ser usadas para procesar información hasta que acabe el periodo del emisor de la llave emisora.

El estado «comprometido» representa a las llaves a las que una entidad sin autorización tiene o ha tenido acceso. Las llaves comprometidas no deben ser usadas para aplicar protección criptográfica, sin embargo, en algunos casos es posible que sean usadas para procesar información (en condiciones sumamente controladas); por ejemplo, si la cierta información fue firmada antes de que se produjera la brecha de seguridad, las llaves pueden ser usadas para valida esta firma.

Apéndice B

Pruebas estadísticas para generadores de números aleatorios y pseudoaleatorios

«*If you don't know who I am, then maybe your best course would be to tread lightly.*»
Walter White.

B.1. Definiciones

Existen dos tipos básicos de generadores, los Random Number Generator (RNG) y los Pseudorandom Number Generator (PRNG).

B.1.1. Generadores aleatorios

Este tipo de generador usa fuentes de entropía no determinísticas para producir números aleatorios, dichas fuentes de entropía normalmente consisten en la medición de eventos físicos, como el ruido en un circuito eléctrico.

La salidas de los Random Number Generator (RNG) puede usarse directamente para obtener números aleatorios, o como una entrada de un Pseudorandom Number Generator (PRNG). En el primer caso, es necesario que la salida satisfaga pruebas estadísticas para determinar que dicha salida parece aleatoria.

Los RNG que se usan con fines criptográficos tienen que ser impredecibles (ver sección B.1.4), por lo cual es recomendado combinar varias fuentes de entropía, dado que algunas (como los vectores de tiempo) son demasiado predecibles. De cualquier forma, en la mayoría de los casos es mejor usar los PRNG para obtener de manera directa números aleatorios.

B.1.2. Generadores pseudoaleatorios

Estos son generadores de múltiples números aleatorios a partir de una o varias entradas llamadas semilla. Cuando es necesario que la salida sea impredecible, la semilla debe ser aleatoria y también impredecible por sí sola, razón por lo cual esta se debe de obtener por medio de un Random Number Generator (RNG).

La salida de los Pseudorandom Number Generator (PRNG) normalmente son funciones determinísticas de la semilla, por lo que la verdadera aleatoriedad queda en la generación de esta misma, motivo por el que se usa el término de *pseudoaleatorio* y lo que causa la necesidad de mantener en secreto la semilla.

Las mejores cualidades de los PRNG son que, al contrario de lo que se creería por el nombre, suelen parecer más aleatorias que las RNG, así como que producen números aleatorios más rápido.

B.1.3. Aleatoriedad

Una secuencia de bits aleatoria puede interpretarse como una secuencia en la que el valor de cada bit es independiente, y los valores que lo conforman están uniformemente distribuidos (ver distribución

UNIFORME).

B.1.4. Impredecibilidad

Los generadores aleatorios y pseudoaleatorios deben de ser impredecibles. La impredecibilidad es la incapacidad de conocer la salida de un bit $n + 1$ aun conociendo los valores del bit 0 al n si la SEMILLA es desconocida. Esta también es la incapacidad de determinar la SEMILLA usando el conocimiento de los valores generados por la misma.

Hay que resaltar que conociendo tanto la SEMILLA como el algoritmo generador, el PSEUDORANDOM NUMBER GENERATOR (PRNG) es completamente predecible, por lo cual en la mayoría de los casos el algoritmo es público y la SEMILLA se mantiene secreta.

B.1.5. Pruebas estadísticas

Estas sirven para determinar si la secuencia a evaluar puede ser considerada una secuencia aleatoria. Existe una gran diversidad de pruebas estadísticas para comparar y evaluar una secuencia generada por un RANDOM NUMBER GENERATOR (RNG) o un PSEUDORANDOM NUMBER GENERATOR (PRNG) contra una secuencia verdaderamente aleatoria, cada una de estas buscando la existencia o ausencia de patrones para poder indicar si la secuencia parece o no aleatoria.

Es necesario aclarar que ningún conjunto de pruebas puede ser considerado completo, y que el resultado de determinada prueba es complementado por otros, motivo por el cual los resultados obtenidos deben ser interpretados cuidadosamente para no llegar a conclusiones incorrectas.

En [38] se determina que las pruebas estadísticas están formuladas para determinar si una secuencia binaria cumple con la hipótesis nula *(H0)*, o la hipótesis alternativa *(Ha)*, las cuales dicen que la secuencia probada es aleatoria y no aleatoria respectivamente. Cada prueba descrita en la sección B.2 consiste en aceptar alguna de las dos hipótesis mencionadas.

Para cada prueba, es necesario elegir una medida estadística relevante de aleatoriedad, para así establecer una DISTRIBUCIÓN DE PROBABILIDAD y un valor crítico que sirvan de referencia, y durante la prueba, se pueda comparar el valor estadístico de la secuencia probada con el valor elegido; aceptando la hipótesis nula en caso de que no se exceda del valor crítico y rechazándola en caso contrario. De manera práctica, las pruebas estadísticas funcionan dado que la DISTRIBUCIÓN DE PROBABILIDAD de referencia y el valor crítico son seleccionados asumiendo que la secuencia es aleatoria.

Hay que mencionar que estas pruebas estadísticas pueden tener dos tipos de errores, que la secuencia sea aleatoria y se concluya que no lo es, *(error tipo I)*, y que la secuencia no sea aleatoria y se concluya que sí, *(error tipo II)*. La probabilidad de que ocurra el error de tipo I es normalmente llamada *nivel de*

significancia de la prueba, y se denota como α, este valor dentro de la criptografía es común que ronde cerca de 0.01. La probabilidad de que ocurra el error de tipo II se denota con β y a diferencia de α, no es un valor fijo y es más difícil de calcular.

Uno de los principales objetivos de las pruebas es minimizar la probabilidad de β, por lo cual se selecciona un tamaño de muestra n y un valor para α, para después establecer un valor crítico que produzca el valor β más pequeño. Esto es posible gracias a que las probabilidades α y β están relacionadas entre sí y con el tamaño de muestra n.

Cada prueba se basa en un valor estadístico de prueba calculado, que es una función de los mismos datos, por lo cual si el valor estadístico de prueba es S y el valor crítico es t, entonces:

$$\alpha = P(S > t \,||\, H0\, es\, verdadera) = P(rechazar\, H0 \,||\, H0\, es\, verdadero) \tag{B.1}$$

$$\beta = P(S \leq t \,||\, H0\, es\, falso) = P(aceptar\, H0 \,||\, H0 es\, falso) \tag{B.2}$$

Las pruebas estadísticas calculan un valor P, que indica la probabilidad de que un RNG perfecto haya producido una secuencia menos aleatoria que la secuencia que se probó, dado el tipo de aleatoriedad evaluada por la prueba. En caso de que el valor de P sea de 1 en una prueba, se considera que la secuencia parece tener una aleatoriedad perfecta y en caso de que el valor sea 0, la secuencia parece ser por completo no aleatoria.

Eligiendo un nivel de significancia α para las pruebas, si $P \geq \alpha$, entonces la hipótesis nula es aceptada, es decir, la secuencia parece ser aleatoria; y si $P < \alpha$, entonces la hipótesis nula es rechazada, es decir, la secuencia parece ser no aleatoria. Normalmente el valor de α se elige en el rango de [0.001, 0.01].

B.2. Pruebas

El paquete de pruebas descrito en el estándar del National Institute of Standards and Technology (NIST) *800-22R1A: Statistical Suite for Random Number Generators*, disponible en [38], es un conjunto de 15 algoritmos que se encargan de probar la aleatoriedad de secuencias binarias de longitudes arbitrariamente largas producidas por un Random Number Generator (RNG) o un Pseudorandom Number Generator (PRNG).

B.2.1. Prueba de frecuencia

(The frequency (Monobit) Test)

Esta prueba se encarga de medir la proporción de ceros y unos en toda la secuencia binaria. Su propósito es determinar si la cantidad de ceros y de unos es parecida a la que se esperaría en una secuencia

verdaderamente aleatoria. Algo a tener en cuenta es que todos las pruebas subsecuentes dependen de pasar esta prueba.

B.2.2. Prueba de frecuencia en un bloque

(Frequency Test within a Block)

Esta prueba se centra en medir la proporción de unos dentro de N bloques de M-bits de la secuencia a probar. El propósito de la prueba es saber si la frecuencia de unos se aproxima a $M/2$ como sería de esperar de una secuencia aleatoria.

B.2.3. Prueba de carreras

(The Runs Test)

Esta prueba se encarga de calcular el número de carreras en la secuencia que se está probando, donde una carrera es una subsecuencia ininterrumpida de bits del mismo valor encerrada entre bits del valor opuesto. El objetivo de la prueba es determinar si el número de carreras en la secuencia es parecido al que se esperaría de una secuencia aleatoria, particularmente, establece si la oscilación entre ceros y unos es muy rápida o muy lenta. Cabe resaltar que la prueba tiene como requisito pasar la prueba de frecuencia.

B.2.4. Prueba de la carrera más larga en un bloque

(Tests for the Longest-Run-of-Ones in a Block)

El propósito de esta prueba es determinar si la longitud de la carrera más larga en la secuencia es consistente con la longitud de la carrera más larga de una secuencia aleatoria. Se tiene que resaltar que encontrar una irregularidad en la carrera más larga de unos implica una irregularidad en la carrera más larga de ceros, por lo cual solo es necesario hacer la prueba para la carrera más larga de unos.

B.2.5. Prueba del rango de matriz binaria

(The Binary Matrix Rank Test)

La prueba tiene como finalidad revisar si existen DEPENDENCIAS LINEALES entre subcadenas de longitud fija de la secuencia a probar.

B.2.6. Prueba Espectral

(The Discrete Fourier Transform (Spectral) Test)

Esta prueba se encarga de obtener los picos más altos de la transformada discreta de Fourier de la secuencia. Su propósito es detectar comportamientos periódicos en la secuencia que indiquen que esta no puede asumirse como aleatoria. De manera más precisa, la prueba detecta si el número de picos que exceden un umbral del 95 % son significativamente diferentes al 5 %.

B.2.7. Prueba de coincidencia sin superposición

(The Non-overlapping Template Matching Test)

La prueba tiene como finalidad detectar generadores que producen una gran cantidad de ocurrencias de un patrón no periódico en sus secuencias de salida. Para esta prueba se utiliza una ventana de m bits que recorre la secuencia, para buscar patrones específicos de m bits. Dado que esta prueba es sin traslape, en caso de que se encuentre el patrón se recorre la ventana m bits y de lo contrario se recorre uno.

B.2.8. Prueba de coincidencia con superposición

(The Overlapping Template Matching Test)

La prueba tiene como finalidad detectar generadores que producen una gran cantidad de ocurrencias de un patrón no periódico en sus secuencias de salida. Para esta prueba se utiliza una ventana de m bits que recorre la secuencia, para buscar patrones específicos de m bits. Dado que esta prueba es con traslape, independientemente de si se encontra o no un patrón, la ventana se recorre un bit.

B.2.9. Prueba estadística universal de Maurer

(Maurer's "Universal Statistical" Test)

Esta prueba se centra en medir el número de bits existentes entre patrones que coinciden, medida que se relaciona con el tamaño mínimo para comprimir una secuencia. La prueba tiene como finalidad determinar si una secuencia puede comprimirse de manera significativa sin perder información, lo cual significa que no es aleatoria.

B.2.10. Prueba de complejidad lineal

(The Linear Complexity Test)

El objetivo de la prueba es determinar si la secuencia a probar es lo bastante compleja como para considerarse aleatoria, esto por medio de medir el tamaño de su REGISTRO DE DESPLAZAMIENTO CON RETROALIMENTACIÓN LINEAL correspondiente, ya que las secuencias aleatorias se caracterizan por tener valores altos en este parámetro.

B.2.11. Prueba serial

(The Serial Test)

Esta prueba tiene como objetivo determinar si el número de ocurrencias de 2^m patrones de m bits son aproximadamente las mismas que se esperarían de una cadena aleatoria, considerando que al ser una secuencia aleatoria, es una secuencia uniforme, y cada patrón de m bits tiene la misma probabilidad de aparecer que los demás.

B.2.12. Prueba de entropía aproximada

(The Approximate Entropy Test)

La prueba se encarga de comparar la frecuencia de sobreponer dos bloques de longitudes consecutivas con el resultado esperado de una secuencia que es aleatoria.

B.2.13. Prueba de sumas acumulativas

(The Cumulative Sums (Cusums) Test)

Su objetivo es determinar si las sumas acumulativas de una secuencia parcial de la secuencia probada son muy grandes o pequeñas con relación a las sumas acumulativas de una secuencia aleatoria. Las sumas aculumativas son consideradas como caminatas aleatorias, y para las secuencias aleatorias la realización de una caminata aleatoria debe tener un valor cercano a cero.

B.2.14. Prueba de excursiones aleatorias

(The Random Excursions Test)

Esta prueba tiene como finalidad determinar si el número de apariciones de un estado x en un ciclo es parecido al número que se esperaría de una secuencia aleatoria. Esta prueba está conformada de 8 subpruebas, cada una para los valores de estado x de -4, -3, -2, -1, 1, 2, 3, 4.

B.2.15. Prueba variante de excursiones aleatorias

(The Random Excursions Variant Test)

La prueba se centra en calcular el número de veces que un estado en particular ocurre en una suma acumulativa de una camita aleatoria. Su propósito, es detectar desviaciones del número esperado de ocurrencias de varios estados en una caminata aleatoria. Esta prueba está conformada de 18 subpruebas, cada un para los valores de estado x de -9 a de -1 y 1 a 9.

Glosario

Glosario de términos criptográficos y matemáticos. Las principales fuentes bibliográficas usadas son [16] y [46]; en caso de tratarse de una fuente distinta, se indica en la entrada en particular.

1. Acoplamiento

(*Coupling*) Medida de la fuerza de asociación establecida por la conexión de dos módulos dentro de un sistema. Un acoplamiento fuerte complica a un sistema, dado que es más difícil de entender, cambiar o corregir. La complejidad de un programa se puede reducir buscando el menor acoplamiento posible entre módulos [47]. 1, 79, 95

2. Autenticación de origen

Tipo de autenticación donde se corrobora que una entidad es la fuente original de la creación de un conjunto de datos en un tiempo específico. Por definición, la *autenticación de origen* incluye la integridad de datos, pues cuando se modifican los datos, se tiene una nueva fuente. 1, 30, 128, *véase también* INTEGRIDAD DE DATOS.

3. Autenticación multifactor

(*Multi-factor authentication*) Método de autenticación que requiere al menos dos métodos (independientes entre sí) para identificar al usuario. 1, 71, 77

4. Autenticación mutua

(*Mutual authentication*) Autenticación en la cual cada una de las partes identifica a la otra. 1, 72

5. Biyección

Dicho de las funciones que son inyectivas y suprayectivas al mismo tiempo; en otras palabras, que todos los elementos del conjunto de salida tengan una imagen distinta en el conjunto de llegada y a cada elemento del conjunto de llegada le corresponde un elemento del conjunto de salida. 1, 17, *véase también* INYECTIVA, SUPRAYECTIVA, FUNCIÓN y IMAGEN.

6. Cifrado con prefijo

Técnica de ordenamiento pseudoaleatorio que consite en seguir el orden dado por el texto cifrado de todos los elementos a ordenar. 1, 35

7. Cifrado de caminata cíclica

(*Cycle-walking cipher*) Método diseñado para cifrar mensajes de un espacio M utilizando un algoritmo de cifrado por bloques que actúa en un espacio $M' \supset M$ y obtener textos cifrados que están en M al cifrar iterativamente hasta que el mensaje cifrado se encuentra en el dominio deseado. 1, 64, 108, 119

8. Cifrado iterativo

(*Iterated block cipher*) Cifrado de bloque que involucra la repetición secuencial de una función interna llamada función de ronda. Los parámetros incluyen el número de rondas, el tamaño de bloque y el tamaño de llave. 1, 15, *véase también* RONDA.

9. Circuito booleano

(*Boolean circuit*) Modelo matemático definido en términos de compuertas lógicas digitales (AND, OR, NOT, etc.). 1

10. Codominio

Una función mapea a los elementos de un conjunto A con elementos de un conjunto B; A es el dominio y B es el *codominio*. 1, *véase también* FUNCIÓN y DOMINIO.

11. Combinación lineal

En álgebra lineal, una combinación lineal es la expresión resultate de la suma de n vectores v_i multiplicados por n escalares c_i de la siguiente forma: $\sum_{i=1}^{n} c_i v_i$. 1

12. Computacionalmente indistinguible

(*Computational indistinguishability*) Para un A tomado de algún conjunto de distribución y un circuito booleano C (con las suficientes entradas), p_C^A es la probabilidad de que la salida del circuito booleano C sea 1 para una entrada de A. Se dice que los conjuntos de distribución $\{A_k\}$ y $\{B_k\}$ son *computacionalmente indistinguibles* si para cualquier familia de circuitos de tamaño polinomial $C = \{C_k\}$, la función $e(k) = |p_{A_k}^{C_k} - p_{B_k}^{C_k}|$ es despreciable [48].
Otra forma de expresarlo, en un contexto más criptográfico, es como la incapacidad de un adversario de distingir si la salida de una primitiva criptográfica es una permutación aleatoria o una permutación pseudoaleatoria: una permutación P es segura (en términos de un ataque de texto cifrado conocido) cuando es *computacionalmente indistingible* de una permutación aleatoria. 1, *véase también* FUNCIÓN, FUNCIÓN DESPRECIABLE, CONJUNTO DE DISTRIBUCIÓN y CIRCUITO BOOLEANO.

13. Computacionalmente no factible

(*Computationally infeasible*) Se dice que una tarea es *computacionalmente no factible* si su costo (medido en términos de espacio o de tiempo) es finito pero ridículamente grande. 1, 71

14. Conjunto de distribución

(*Distribution ensemble*) Un *conjunto de distribución* $\{A_k\}$ es una familia de medidas de probabilidad en $\{0,1\}^*$ para la cual hay un polinomio q tal que las únicas cadenas de longitud mayor a $q(k)$ tienen una probabilidad distinta de cero en $\{A_k\}$ [48]. 1

15. Criptografía fuerte

(*Strong cryptography*) De acuerdo al Payment Card Industry (PCI) Security Standard Council (SSC) (en [49]), es la criptografía basada en algorítmos probados y aceptados en la industria, junto con longitudes de llaves fuertes y buenas prácticas de administración de llaves. 1, 39, 72, 130

16. Criptología

Estudio de los sistemas, claves y lenguajes secretos u ocultos. 1, 11

17. Criptomoneda

(*Cryptocurrency*) Moneda digital que funciona con herramientas criptográficas modernas. La primera y más exitosa a la fecha, llamada *bitcoin*, fue propuesta en [15] por el epónimo *Satoshi Nakamoto*. La característica más importante de las *criptomonedas* es que no están respaldadas por alguna autoridad central [50]. 1, 7

18. Dependencia lineal

Un conjunto de vectores es linealmente independiente sin ninguno de ellos puede ser representado por una combinación lineal de los vectores restantes. Si los vectores no son linealmente independientes, se dice que existen dependencias lineales. 1, 139, *véase también* combinación lineal.

19. Distribución de probabilidad

Una distribución de probabilidad P en el conjunto de eventos S es una secuencia de números positivos p_1, p_2, $\ldots$ p_n que sumados dan 1. Donde p_i se interpreta como la probabilidad de que el evento s_i ocurra. 1, 137

20. Distribución uniforme

Distribución de probabilidad en la que todos los elementos tienen la misma probabilidad ocurrencia. 1, 136, *véase también* distribución de probabilidad.

21. Dominio

El *dominio* de una función $f(x)$ es el conjunto de valores para los cuales la función está definida. 1, *véase también* función y codominio.

22. Entropía

Definida para una función de probabilidad de distribución discreta, mide cuánta información en promedio es requerida para identificar muestras aleatorias de esa distribución. 1, 15, 43–45, 47, 50, 51, 73, 136, *véase también* función y distribución de probabilidad.

23. Equiprobable

Se dice que un conjunto de eventos es equiprobable cuando cada uno tiene la misma probabilidad de ocurrencia. 1, 63, 75

24. Estadísticamente independiente

Dicho de la ocurrencia de dos eventos E_1 y E_2: si $P(E_1 \cap E_2) = P(E_1)P(E_2)$ entonces E_1 y E_2 son *estadísticamente independientes* entre sí. Es importante notar que si esto ocurre, entonces $P(E_1|E_2) = P(E_1)$ y $P(E_2|E_1) = P(E_2)$, es decir, la ocurrencia de uno no tiene ninguna influencia en las probabilidades de ocurrencia del otro. 1, 74, 76, *véase también* PROBABILIDAD CONDICIONAL.

25. Fuerza efectiva

Para un espacio de llaves K, su *fuerza efectiva* es $\log_2 |K|$ (el logaritmo base dos de su cardinalidad). 1, 74, 76, 130

26. Función

Regla entre dos conjuntos A y B de manera que a cada elemento del conjunto A le corresponda un único elemento del conjunto B. 1, *véase también* CODOMINIO y DOMINIO.

27. Función booleana

Son las funciones que mapean f a un valor del conjunto booleano $0, 1$ o *verdadero* y *falso*. Formalmente, se define como $f : B^n \to B$, donde $B = 0, 1$ y n un entero no negativo que corresponde al número de argumentos, o variables, que necesita la función. 1, 64, *véase también* FUNCIÓN.

28. Función despreciable

(*Negligible function*) Una función $e : N \to R$ es *despreciable* si, para todos los enteros positivos c, existe un entero N_c tal que para todo $x \geq N_c$, $|e(x)| \leq \frac{1}{x^c}$. Esto significa que $e(x)$ se desvanece más rápido que el inverso de cualquier polinomio [48]. 1, *véase también* FUNCIÓN.

29. Imagen

Suponga que se tiene $x \in X$ y $y \in Y$ tal que $f(x) = y$; se dice entonces que y es la *imagen* de x bajo f, o que x es preimagen de y. 1, *véase también* PREIMAGEN, FUNCIÓN, DOMINIO y CODOMINIO.

30. Integridad de datos

Propiedad en la que los datos no han sido alterados sin autorización desde que fueron creados, transmitidos o almacenados por una fuente autorizada. Operaciones que insertan, eliminan, modifican o reordenan bits invalidan la *integridad de los datos*. La *integridad de los datos* incluye que los datos estén completos y, cuando los datos son divididos en bloques, cada bloque debe cumplir con lo mencionado anteriormente. 1, 30

31. Inyectiva

Una función $f : D_f \to C_f$ es *inyectiva* (o uno a uno) si a diferentes elementos del dominio le corresponden diferentes elementos del codominio; se cumple para dos valores cualesquiera $x_1, x_2 \in D_f$ que $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$. 1, *véase también* FUNCIÓN, DOMINIO, CODOMINIO, BIYECCIÓN y SUPRAYECTIVA.

32. Libreta de un solo uso

(*One-time pad*) Algorimo de cifrado donde el texto en claro se combina con una llave secreta que es, al menos, de la longitud del mensaje. Si es utillizado e implementado correctamente, este algoritmo es indescifrable. 1

33. Material de llaves

(*keying material*) Conjunto de llaves criptográficas sin un formato específico. 1

34. Modo de operación

Construcción que permite extender la funcionalidad de un cifrado a bloques para operar sobre tamaños de información arbitrarios. 1, 14, 24–27, 34, 40, 41, 159, 161, 162

35. Máquina de Turing

(*Turing machine*) Se considera como una cinta infinita dividida en casillas, cada una de las cuales contiene un símbolo. Sobre dicha cinta actúa un dispositivo que puede adoptar distintos estados y que, en cada instante, lee un símbolo de la casilla sobre la que está situado; dependiendo del símbolo leído y del estado en el que se encuentra, la máquina realiza las siguientes tres acciones: primero, pasa a un nuevo estado; segundo, imprime un símbolo en el lugar del que acaba de leer; y, tercero, se desplaza hacia la derecha, hacia la izquierda o se detiene. 1

36. Nonce

Valor que varía con el tiempo y es improbable que se repita. Por ejemplo, puede ser un valor aleatorio generado para cada uso, una etiqueta de tiempo, un número de secuencia o una combinación de los tres. 1, 43–46

37. Oráculo

Oracle machine Se refiere a una máquina abstracta utilizada para estudiar problemas de decisión. Puede verse como una máquina de Turing con una caja negra (llamada *oráculo*) que puede resolver ciertos problemas de decisión u obtener el valor de una función en una sola operación. 1, 73, *véase también* MÁQUINA DE TURING.

38. Permutación

Sea S un conjunto finito de elementos. Una *permutación* p en S es una biyección de S a sí misma (i. e. $p : S \to S$). 1, 12, 74, 75, *véase también* BIYECCIÓN.

39. Preimagen

Suponga que se tiene $x \in X$ y $y \in Y$ tal que $f(x) = y$; se dice entonces que x es *preimagen* de y, o que y es la imagen de x bajo f. 1, 28, 48, *véase también* IMAGEN, FUNCIÓN, DOMINIO y CODOMINIO.

40. Primitiva criptográfica

(*Cryptographic primitive*) Algoritmos criptográficos que son usados con frecuencia para la construcción de protocolos de seguridad. En [16] se clasifican en tres categorías principales: de llave simétrica, de llave pública y sin llave. 1, 40, 52, 53, 62, 63, 65, 73

41. Probabilidad condicional

Sean E_1 y E_2 dos eventos, con $P(E_2) \geq 0$. La *probabilidad condicional* se denota por $P(E_1|E_2)$, y es igual a

$$P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)}$$

Esto mide la probabilidad de que ocurra E_1 sabiendo que ya ocurrió E_2 1

42. Prueba de componente

(*Component testing*) Pruebas de componentes hechos a partir de objetos que interactúan entre sí [51]. 1, 86

43. Prueba de unidad

(*Unit testing*) Proceso para probar componentes más simples de un programa de forma individual (funciones u objetos aislados) [51]. 1, 86

44. Registro de desplazamiento

Es un arreglo de *flip-flops* que se encarga de desplazar la información contenida en su estado actual, teniendo una entrada de un bit, para reemplazar el valor del primer *flip-flop* del arreglo. 1

45. Registro de desplazamiento con retroalimentación lineal

(*Linear-feedback shift register (LFSR)*) Es un registro de desplazamiento en el que su bit de entrada es una función lineal de su estado anterior. 1, 141, *véase también* REGISTRO DE DESPLAZAMIENTO.

46. Ronda

(*Round*) Bloque compuesto por un conjunto de operaciones que es ejecutado múltiples veces. Las *rondas* son definidas por el algoritmo de cifrado. 1, 15, 17, 19, 20

47. Semilla

(*Seed*) Cadena de bits que es utilizada como entrada para los los mecanismos Deterministic Random Bit Generator (DRBG). Determina una parte del estado interno del DRBG. 1, 42–48, 50, 136, 137

48. Suprayectiva

Una función $f : D_f \to C_f$ es *suprayectiva* si todo elemento de su codominio C_f es imagen de por lo menos un elemento de su dominio D_f: $\forall b \in C_f \; \exists a \in D_f$ tal que $f(a) = b$. 1, *véase también* función, dominio, codominio, imagen, biyección y inyectiva.

49. Tiempo polinomial

Se dice que una función computable o algoritmo es de *tiempo polinomial* cuando su complejidad está, en el peor de los casos, acotada por arriba por un polinomio sobre el tamaño de sus variables de entrada. Si se toma a f como una función computable, esta es de *tiempo polinomial* si $f \in O(n^k)$ donde $k \geq 1$. 1

50. Token

Valor representativo que se usa en lugar de información valiosa. 1, 3–7, 10, 39, 41, 51–53, 62–64, 67, 70–77, 79, 80, 85, 103, 108, 118, 119, 159, 163

51. Vector de inicialización

Cadena de bits de tamaño fijo que sirve como entrada a muchas primitivas criptográficas (p. ej. algunos modos de operación). Generalmente se requiere que sea generado de forma aleatoria. 1, 25, 32, 34, 59, 128, *véase también* modo de operación y primitiva criptográfica.

Siglas y acrónimos

Criptográficos

ABL Arbitrary Block Length Mode. 34

AES Advanced Encryption Standard. 10, 13, 15, 20, 30, 34, 41, 54–56, 72, 85, 95, 103, 107, 108, 113–115, 161, 163

CAVP Cryptographic Algorithm Validation Program. 40

CBC Cipher-block Chaining. 10, 24–27, 30, 31, 33, 41, 55, 95, 149, 159, 162

CFB Cipher Feedback. 24, 41

CMC Cipher-block Chaining (CBC)-Mask-CBC. 33

CRHF Collision-Resistant Hash Function. 28

CSP Critical Security Parameter. 44, 45

CTR Counter Mode. 10, 24, 27, 34, 41, 103, 150, 162

DES Data Encryption Standard. 13, 15, 17, 19, 20, 30, 41, 56, 85, 103, 151

DH Diffie-Hellman. 41, 150

DHE Diffie-Hellman (DH) Ephemeral. 41

DRBG Deterministic Random Bit Generator. 42–46, 48, 50–53, 66, 68, 85, 86, 108, 111–118, 149, 161, 163

DSA Digital Signature Algorithm. 41, 150

ECB Electronic Codebook. 10, 24, 25, 34, 150, 159, 162

ECC Elliptic Curve Cryptosystem. 41

ECDH Elliptic Curve DH. 41

ECDSA Elliptic Curve Digital Signature Algorithm (DSA). 41

ECIES Elliptic Curve Integrated Encryption Scheme. 41

ECMQV Elliptic Curve Menezes-Qu-Vanstone. 41

EME Electronic Codebook (ECB)-Mask-ECB. 34

FEAL Fast Data Encipherment Algorithm. 15

FFX Format-preserving Feistel-based Encryption. 53, 54, 82, 95, 96, 161, 163

FPE Format Preserving Encryption. 34, 35, 62

HCTR Hash Counter Mode (CTR). 34

HMAC Keyed-Hashed Message Authentication Code. IV, 32, 33, 159

IDEA International Data Encryption Algorithm. 15

MAA Message Authenticator Algorithm. 30

MAC Message Authentication Code. 10, 29–32, 55, 95, 151, 162

MD4 Message Digest-4. 29, 30, 41

MD5 Message Digest-5. 28, 29, 41

MDC Message Digest Cipher. 30

NMAC Nested Message Authentication Code (MAC). 32

NRBG Non-deterministic Random Bit Generator. 42, 52, 53

OAEP Optimal Asymmetric Encryption Padding. 41

OCB Offset Codebook. 41

OFB Output Feedback. 24, 41

OMAC One-key MAC. 31

OWHF One-Way Hash Function. 28, 32

PRF Pseudorandom Function. IV, 30, 31, 162

PRNG Pseudorandom Number Generator. 76, 136–138

RBG Random Bit Generator. 42

RNG Random Number Generator. 136–138

RSA Ron Rivest, Adi Shamir, Leonard Adleman. 41, 151

RSAES Ron Rivest, Adi Shamir, Leonard Adleman (RSA) Encryption Scheme. 41

SAFER Secure And Fast Encryption Routine. 15

SHA Secure Hash Algorithm. 28–30, 42, 56, 113, 115–118, 161

TBC Tweakable Block Chaining. 33, 34, 159

TBC Tweakable Block Cipher. 32–34

TDES Triple Data Encryption Standard (DES). 41, 56

TES Tweakable Encyphering Scheme. 10, 32–34

TRNG True Random Number Generator. 63

XCB Extended Codebook. 34

Computacionales

API Application Program Interface. 71

ASCII American Standard Code for Information Interchange. 35, 94

BSL Boost Software License. 93

DAO Data Access Object. 85

GCC GNU Compilers Collection. 92

GNU GNU is Not Unix. 92, 93, 152

GPL GNU General Public Licence. 92–94

HTML Hypertext Markup Language. 94

PDF Portable Document Format. 94

PHP PHP: Hypertext Processor. 92

SDK Software Development Kit. 5

SDL Security Development Lifecycle. 70

SSH Secure Shell. 32

SSL Secure Sockets Layer. 29, 32

UML Unified Modeling Language. 79

VCS Version Control System. 93, 94

Bancarios

BIN Bank Identifier Number. 64

CDV Card Data Vault. 62, 70, 75, 76, 80, 85, 86

CISP Cardholder Information Security Program. 2

DSS Data Security Standard. 2, 3, 5, 40, 70, 71

INN Issuer Identification Number. 38, 108

MDES MasterCard Digital Enablement Service. 6

MII Major Industry Identifier. 38, 39, 161

PA Payment Application. 71

PAN Personal Account Number. 3, 5, 6, 35, 38, 39, 52, 53, 62–64, 70–78, 80, 85, 103, 108, 118, 119

PCI Payment Card Industry. V, 2–5, 38–41, 51–53, 62, 70, 71, 74, 76–79, 145, 159, 161

VTS Visa Token Services. 6

De instituciones y asociaciones

FIPS Federal Information Processing Standard. 17, 70, 74, 76

IEC International Electrotechnical Commission. 72

ISO International Organization for Standaridization. 72, 74, 76

NIST National Institute of Standards and Technology. 20, 29, 30, 38, 40, 41, 53, 65, 72, 74–76, 93, 108, 113, 128, 131, 138

NSA National Security Agency. 29

NVLAP National Voluntary Laboratory Accreditation Program. 50

RAE Real Academia Española. 10

SSC Security Standard Council. V, 4, 5, 38–41, 51–53, 74, 76, 79, 145, 159, 161

Bibliografía

[1] John S. Kiernan. *Credit Card And Debit Card Fraud Statistics*. 2017. URL: https://wallethub.com/edu/credit-debit-card-fraud-statistics/25725/ (vid. pág. 2).

[2] UK Cards Association. *What is PCI DSS?* 2018. URL: http://www.theukcardsassociation.org.uk/security/What_is_PCI%20DSS.asp (vid. pág. 2).

[3] SearchSecurity Staff. *The history of the PCI DSS standard: A visual timeline*. 2013. URL: https://searchsecurity.techtarget.com/feature/The-history-of-the-PCI-DSS-standard-A-visual-timeline (vid. pág. 2).

[4] David Khan. *The Codebreakers: A Comprehensive History of Secret Communication from Ancient Times to the Internet*. Nueva York: Scribner, 1996 (vid. pág. 3).

[5] Simon Singh. *The Code Book: How to Make It, Break It, Hack It, Crack it*. Nueva York: Delacorte Press, 2001. ISBN: 0-375-89012-2 (vid. pág. 3).

[6] Shift4 Payments. *The History of TrueTokenization*. 2018. URL: https://www.shift4.com/dotn/4tify/trueTokenization.cfm (vid. pág. 5).

[7] Shift4 Payments. *Tried and True Tokenization: The Original Tokenization Solution for Card Data Security*. 2018. URL: https://www.shift4.com/pdf/Shift4-TrueTokenization.pdf (vid. pág. 5).

[8] BluePay Processing, LLC. *What you need to know about tokenization technology and BluePay's TokenShield*. 2018. URL: https://www.bluepay.com/infographic/tokenization/ (vid. pág. 5).

[9] Braintree. *Tokenization Secures CC Data and Meet PCI Compliance Requirements*. 2007. URL: https://www.braintreepayments.com/blog/using-tokenization-to-secure-credit-card-data-and-meet-pci-compliance-requirements/ (vid. pág. 5).

[10] MerchantLink. *TransactionVault Tokenization*. 2018. URL: http://www.merchantlink.com/solutions/products/transactionvault/ (vid. pág. 6).

[11] Jack Henry & Associates Inc. *Card Processing Solutions For Jack Henry Banks*. 2018. URL: https://www.jackhenry.com/jha-payment-solutions/cps-jack-henry-banks/pages/tokenization.aspx (vid. pág. 6).

[12] MasterCard. *Digital Commerce Solutions*. 2018. URL: https://www.mastercard.us/en-us/issuers/products-and-solutions/grow-manage-your-business/digital-commerce-solutions.html (vid. pág. 6).

[13] MasterCard. *Frequently Asked Questions*. 2018. URL: https://www.mastercard.us/en-us/frequently-asked-questions.html#tokenization (vid. pág. 6).

[14] Securosis. *Understanding and Selecting a Tokenization Solution*. 2018. URL: https://securosis.com/assets/library/reports/Securosis_Understanding_Tokenization_V.1_.0_.pdf (vid. pág. 6).

[15] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009. URL: HTTPS://BITCOIN.ORG/BITCOIN.PDF (vid. págs. 7, 145).

[16] Alfred Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7 (vid. págs. 10, 14, 23, 27, 30, 143, 148).

[17] Hans Delfs y Helmut Knebl. *Introduction to Cryptography - Principles and Applications*. Information Security and Cryptography. Springer, 2007. ISBN: 978-3-540-49243-6. DOI: 10.1007/3-540-49244-5. URL: HTTPS://DOI.ORG/10.1007/3-540-49244-5 (vid. págs. 10, 14, 27, 30).

[18] Claude E. Shannon. «Communication theory - Exposition of fundamentals». En: *Trans. of the IRE Professional Group on Information Theory (TIT)* 1 (1953), págs. 44-47. DOI: 10.1109/TIT.1953.1188568. URL: HTTPS://DOI.ORG/10.1109/TIT.1953.1188568 (vid. pág. 11).

[19] Phillip Rogaway. *A Synopsis of Format-Preserving Encryption*. 2010. URL: HTTP://WEB.CS.UCDAVIS.EDU/~ROGAWAY/PAPERS/SYNOPSIS.PDF (vid. págs. 17, 35).

[20] Bruce Schneier y John Kelsey. «Unbalanced Feistel Networks and Block Cipher Design». En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 121-144. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_49. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_49 (vid. pág. 17).

[21] Ross J. Anderson y Eli Biham. «Two Practical and Provably Secure Block Ciphers: BEARS and LION». En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 113-120. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_48. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_48 (vid. pág. 17).

[22] Stefan Lucks. «Faster Luby-Rackoff Ciphers». En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 189-203. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_53. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_53 (vid. pág. 17).

[23] Debrup Chakraborty y Francisco Rodríguez-Henríquez. «Block Cipher Modes of Operation from a Hardware Implementation Perspective». En: *Cryptographic Engineering*. Ed. por Çetin Kaya Koç. Springer, 2009, págs. 321-363. ISBN: 978-0-387-71816-3. DOI: 10.1007/978-0-387-71817-0_12. URL: HTTPS://DOI.ORG/10.1007/978-0-387-71817-0_12 (vid. pág. 23).

[24] Alaa Hussein Al-Hamami y Ghossoon M. Waleed al-Saadoo. *Handbook of research on threat detection and countermeasures in network security*. 1.ª ed. IGI Global, 2015 (vid. pág. 27).

[25] Prakash C. Gupta. *Cryptography and Network Security*. PHI Learning, 2015 (vid. pág. 27).

[26] Dhiren R. Patel. *Information security - Theory and Practice*. Prentice-Hall of India, 2008 (vid. pág. 30).

[27] Cuauhtémoc Mancillas López. «Studies on Disk Encryption». Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional, 2013 (vid. pág. 32).

[28] Moses Liskov, Ronald L. Rivest y David A. Wagner. «Tweakable Block Ciphers». En: *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*. Ed. por Moti Yung. Vol. 2442. Lecture Notes in Computer Science. Springer, 2002, págs. 31-46. ISBN: 3-540-44050-X. DOI: 10.1007/3-540-45708-9_3. URL: HTTPS://DOI.ORG/10.1007/3-540-45708-9_3 (vid. pág. 32).

[29] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969. ISBN: 0201038021. URL: HTTP://WWW.WORLDCAT.ORG/OCLC/310551264 (vid. págs. 36, 162).

[30] International Organization for Standarization. *ISO/IEC 7812*. 5.ª ed. 2017, pág. 7. URL: HTTPS://WWW.ISO.ORG/STANDARD/70484.HTML (vid. pág. 38).

[31] International Organization for Standardization. *ISO 9362*. 4.ª ed. 2014, pág. 6. URL: HTTPS://WWW.ISO.ORG/STANDARD/60390.HTML (vid. pág. 38).

[32] Abhay Bhargav. *PCI compliance: The Definitive Guide*. CRC Press, 2015 (vid. pág. 38).

[33] Payment Card Industry Security Standards Council. *Tokenization Product Security Guidelines – Irreversible and Reversible Tokens*. 2015. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/DOCUMENTS/TOKENIZATION_PRODUCT_SECURITY_GUIDELINES.PDF (vid. págs. 39, 40, 74-76).

[34] Payment Card Industry Security Standards Council. *Data Security Standard - Version 3.2*. 2016. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/DOCUMENTS/PCI_DSS_V3-2.PDF (vid. págs. 40, 71).

[35] Elaine Barker y John Kelsey. *NIST Special Publication 800-90A - Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. 2015. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-90AR1 (vid. págs. 41-43, 65, 74, 75).

[36] Elaine Barker. *NIST Special Publication 800-57 - Recommendation for Key Management*. 2016. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-57PT1R4 (vid. págs. 42, 72, 128, 131).

[37] Elaine Barker, Miles Smid, Dennis Branstad y col. *NIST Special Publication 800-130 - A Framework for Designing Cryptographic Key Management Systems*. 2013. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-130 (vid. págs. 42, 72).

[38] Andrew Rukhin, Juan Soto, James Nechvatal y col. *NIST Special Publication 800-22 Revision 1a - A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. 2010. URL: HTTPS://NVLPUBS.NIST.GOV/NISTPUBS/LEGACY/SP/NISTSPECIALPUBLICATION800-22R1A.PDF (vid. págs. 42, 113, 137, 138).

[39] Mihir Bellare, Phillip Rogaway y Terence Spies. «The FFX Mode of Operation for Format-Preserving Encryption». Ver. 1.0. En: (2009) (vid. pág. 53).

[40] Mihir Bellare, Phillip Rogaway y Terence Spies. «The FFX Mode of Operation for Format-Preserving Encryption». Ver. 1.1. En: (2010) (vid. pág. 54).

[41] Eric Brier, Thomas Peyrin y Jacques Stern. «BPS: a Format-Preserving Encryption Proposal». En: (2010) (vid. pág. 55).

[42] Sandra Diaz-Santiago, Lil María Rodríguez-Henríquez y Debrup Chakraborty. «A cryptographic study of tokenization systems». En: *Int. J. Inf. Sec.* 15.4 (2016), págs. 413-432. DOI: 10.1007/s10207-015-0313-x. URL: HTTPS://DOI.ORG/10.1007/S10207-015-0313-X (vid. págs. 62, 85).

[43] Riccardo Aragona, Riccardo Longo y Massimiliano Sala. «Several proofs of security for a tokenization algorithm». En: *Appl. Algebra Eng. Commun. Comput.* 28.5 (2017), págs. 425-436. DOI: 10.1007/s00200-017-0313-3. URL: HTTPS://DOI.ORG/10.1007/S00200-017-0313-3 (vid. pág. 63).

[44] Information Technology Laboratory National Institute of Standards y Technology. *FIPS PUB 140-2 - Security Requirements for cryotographic modules*. 2001. URL: HTTPS://CSRC.NIST.GOV/CSRC/MEDIA/PUBLICATIONS/FIPS/140/2/FINAL/DOCUMENTS/FIPS1402.PDF (vid. págs. 70, 74, 76).

[45] Payment Card Industry Security Standards Council. *Payment Application Data Security Standard - Requirements and Security Assessment Procedures - Version 3.0*. 2013. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/MINISITE/EN/DOCS/PA-DSS_V3.PDF (vid. págs. 71, 72).

[46] William Stallings. *Cryptography and network security - principles and practice (6. ed.)* Pearson, 2014. ISBN: 978-0-13-335469-0 (vid. pág. 143).

[47] Grady Booch, Robert A. Maksimchuk, Michael W. Engle y col. *Object-oriented analysis and design with applications, Third Edition*. Addison Wesley object technology series. Addison-Wesley, 2007. ISBN: 978-0-201-89551-3 (vid. pág. 143).

[48] Donald Beaver, Silvio Micali y Phillip Rogaway. «The Round Complexity of Secure Protocols (Extended Abstract)». En: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. Ed. por Harriet Ortiz. ACM, 1990, págs. 503-513. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100287. URL: HTTP://DOI.ACM.ORG/10.1145/100216.100287 (vid. págs. 144, 146).

[49] Payment Card Industry Security Standards Council. *Data Security Standard (DSS) and Payment Application Data Security Standard (PA-DSS) - Glossary of Terms, Abbreviations, and Acronyms - Version 1.2*. 2008. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/PDFS/PCI_DSS_GLOSSARY.PDF (vid. pág. 145).

[50] Sanjay Bhattacherjee y Palash Sarkar. «Cryptocurrency Voting Games». En: *IACR Cryptology ePrint Archive* 2017 (2017), pág. 1167. URL: HTTP://EPRINT.IACR.ORG/2017/1167 (vid. pág. 145).

[51] Ian Sommerville. *Software engineering, 8th Edition*. International computer science series. Addison-Wesley, 2007. ISBN: 9780321313799. URL: HTTP://WWW.WORLDCAT.ORG/OCLC/65978675 (vid. pág. 148).

[52] Dieter Gollmann, ed. *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6.

Lista de figuras

Lista de tablas

Lista de pseudocódigos