UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

El algoritmo de cifrado AES: su implantación y optimización de bajo nivel para mejorar los mecanismos de seguridad de una tarjeta inteligente

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Alberto Nicolás Escalante Bañuelos

DIRECTOR DE TESIS

Dr. Francisco J. García Ugalde

Ciudad Universitaria, Cd. Mx., 2003

Dedicatoria

A mis padres:
Rosario Bañuelos Castañeda y
Evodio Escalante Betancourt,
quienes me han enseñado la
maravilla de la existencia.

A mis grandes maestros de toda la vida,
a quienes siempre tengo presentes en
mi memoria por su tan noble labor.

A mis hermanos:
Alejandro, Sebastián y Ana Laura,
quienes me han aceptado y
amado tal como soy.

Agradecimientos

A la UNAM,
nuestra máxima casa de estudios,
por la valiosa formación que me ha
proporcionado en todos los sentidos.

A mi director de tesis,
el Dr. Francisco J. García Ugalde,
por sus acertados consejos y su apreciada
dedicación al dirigirme para realizar este trabajo.

A la DGSCA y a la Fundación UNAM,
por su colaboración al desarrollo de
este proyecto mediante la donación de los
equipos lectores de tarjetas que utilizamos.

A mis sinodales,
por sus valiosas sugerencias
para mejorar este trabajo,
y a quienes por fortuna en su mayoría pude
tener además como maestros.

A todos mis familiares,
por su incesante cariño y apoyo.

A mis amigos,
por todos los momentos gratos
que hemos compartido y por haberme
motivado para llevar este trabajo a su conclusión.

A todo aquel que de alguna manera haya
ayudado al desarrollo de esta tesis.

Índice general

Índice de figuras

Índice de cuadros

Resumen

La característica más importante de las tarjetas inteligentes es su capacidad de efectuar procedimientos de autenticación y de controlar el acceso a los datos que contienen. Esto es muy útil puesto que nos permite concentrar los mecanismos de seguridad en un solo punto. Los protocolos criptográficos de autenticación de entidades de las tarjetas inteligentes están basados en un algoritmo de cifrado de bloques. Tradicionalmente se ha utilizado el algoritmo *DES* (*Data Encryption Standard*), sin embargo es necesario emplear algoritmos con mayores longitudes de clave y diseñados específicamente para resistir los ataques mediante algunas técnicas criptoanalíticas que han surgido en los últimos años. Como respuesta a este problema, el algoritmo *Rijndael* fue seleccionado en Octubre de 2002 para convertirse en el *AES* (*Advanced Encryption Standard*) y reemplazar al *DES* y al triple *DES*.

En esta tesis analizamos la factibilidad de emplear el *AES* para mejorar los mecanismos de seguridad de una tarjeta inteligente en particular que no fue evaluada durante el periodo de selección del *AES* y de utilizarlo para construir otros algoritmos o protocolos criptográficos que funcionen dentro de la tarjeta. En primer lugar exploramos diversos aspectos de la tecnología de las tarjetas inteligentes y mostramos entre otras cosas: su funcionamiento, sus aplicaciones, los comandos que soportan y sus mecanismos de seguridad fundamentados en técnicas de cifrado simétricas.

Después estudiamos la teoría básica de la criptografía de clave simétrica y algunos tipos de protocolos que emplean este tipo de cifrado. También analizamos concienzudamente el *AES* haciendo énfasis en las operaciones matemáticas de su especificación y pensando en su implantación y optimización para la arquitectura de tarjetas inteligentes que escogimos: la basada en el microcontrolador RISC de 8 bits *AT90S8515* de la familia *AVR* de *Atmel*.

Implantamos y optimizamos el algoritmo en bajo nivel para dicha arquitectura, y posteriormente mejoramos y ampliamos los mecanismos de seguridad de la tarjeta. Puesto que su estado interno queda oculto por su sistema operativo, con el propósito de probar y medir el desempeño de nuestra implantación, creamos un ambiente de simulación de la tarjeta basado en nuestras propias herramientas y en algunas herramientas libres que utilizamos o modificamos.

Desarrollamos una herramienta de control que nos permite interactuar y ejecutar las nuevas funciones disponibles en las tarjetas. También creamos una biblioteca con la cual podemos trabajar fácilmente con las estructuras algebraicas del *AES*, y con ella obtuvimos las transformaciones inversas del algoritmo, realizamos las transformaciones de ronda (directas e inversas) y generamos una representación polinomial de la caja S y su inversa.

Nuestros resultados indican que nuestra implantación del *AES* tiene un mucho mejor desempeño que el reportado como típico utilizando el *DES* en arquitecturas de tarjetas inteligentes comparables, por lo que al menos desde el punto de vista de los recursos computacionales que ocupan es conveniente reemplazar al anterior *DES* por el nuevo *AES*.

Logramos disminuir significativamente el tiempo de procesamiento de los comandos que previamente empleaban el algoritmo *TEA* (*Tiny Encryption Algorithm*) y tenemos un mejor desempeño que otras implantaciones del *AES* para arquitecturas de 8 bits. Esto quiere decir que mediante nuestra implantación del algoritmo tenemos una mayor flexibilidad en cuanto a los mecanismos de seguridad que podemos efectuar en las tarjetas inteligentes. Un ejemplo de esto es el nuevo comando que creamos, que permite la lectura de archivos asegurando la confidencialidad y autenticación del origen de los datos (y por consiguiente su integridad) mediante el modo de operación *CCM*.

Introducción

La tecnología de las *Tarjetas Inteligentes* (TIs) se originó hace unos 50 años y tiene su primer precedente en las primitivas *tarjetas plásticas* empleadas en los Estados Unidos de América. Inicialmente las tarjetas plásticas eran utilizadas para la identificación personal y para realizar pagos, pero han sufrido un proceso de evolución que ha ocasionado su mejora constante, su proliferación y su diversificación. Entre las tarjetas plásticas actualmente encontramos a las *tarjetas con cinta magnética*, a las *tarjetas ópticas* y a las *tarjetas inteligentes*.

Las *tarjetas inteligentes* son aquellas tarjetas plásticas que tienen un circuito integrado incrustado y algunos contactos metálicos que sirven como interfaz entre dicho circuito y un dispositivo externo, comúnmente un *lector de tarjetas*.

Existen varios tipos de *tarjetas inteligentes*, pero nosotros nos interesamos particularmente por las *tarjetas con microcontrolador*, las cuales, como su nombre lo indica, contienen un microcontrolador y pueden concebirse como una computadora diminuta conectada con el exterior exclusivamente mediante un puerto serial.

Las *tarjetas inteligentes* no solamente pueden almacenar información importante, sino que tienen la capacidad de procesarla. Las operaciones que las tarjetas pueden realizar varían de acuerdo a cada aplicación. Como ejemplo tenemos a *SCQL*[1], estandarizado en el *ISO 7816-7*, que proporciona comandos para realizar operaciones sobre una base de datos contenida dentro de la tarjeta. Las *tarjetas inteligentes* tienen la capacidad de restringir el acceso a la información almacenada y proteger dicha información mediante mecanismos de *software* y de *hardware*. Los mecanismos de *software* son posibles gracias al sistema operativo de las tarjetas con microcontrolador, el cual tiene control sobre cada una de las operaciones que se efectúan en su interior y sirve como intermediario entre una aplicación externa y los datos internos. Los mecanismos de *hardware* están encaminados a evitar o dificultar los ataques a la implantación, por ejemplo, con mecanismos que detecten intentos de intrusión al circuito integrado mismo o condiciones de operación inadecuadas, como frecuencias de reloj o voltajes fuera del rango permitido.

Las *tarjetas inteligentes* son utilizadas frecuentemente como unidades en las que podemos concentrar los mecanismos de seguridad del sistema, de forma que, por ejemplo, tenemos mayor control sobre donde se encuentran las claves, donde se ejecutan los mecanismos criptográficos y en caso de haber algún problema podríamos actualizar o reemplazar exclusivamente la tarjeta y dejar al resto del sistema intacto.

En los últimos años las *tarjetas inteligentes* han ganando terreno a un ritmo sorprendente. Por ejemplo, de acuerdo a [1] se estima que el número de tarjetas enviadas a Estados Unidos y Canadá aumentó en más del 100 % en la primera mitad del año 2002 comparado con el mismo periodo del 2001. En Taiwán [25], un proyecto de salud proporcionará a 24 millones de Taiwaneses una *tarjeta inteligente*

[1] *Structured Card Query Language.*

para el seguro médico. En Europa las *tarjetas inteligentes* han tenido muy buena aceptación desde hace más de 15 años. Otras de las aplicaciones de las *tarjetas inteligentes* incluyen al control de acceso a sistemas de cómputo, al dinero electrónico y a la telefonía fija y móvil. El número de aplicaciones está en crecimiento y así seguirá debido a que la tecnología de las *tarjetas inteligentes* está transformándose continuamente y al incrementarse su poder computacional se abre la posibilidad de implantar nuevas aplicaciones que demanden mayores recursos.

Como es conocido, la tecnología y cada vez más la información tienen un impacto muy fuerte en la vida diaria de las personas ya que afecta la forma en que la sociedad, las empresas y los gobiernos se desarrollan. Una empresa que no cuente con los medios para manejar correctamente su información tendrá menor capacidad de decisión y corre el riesgo de ser dejada atrás por empresas competidoras. Si no se puede asegurar la veracidad de la información de la empresa, particularmente de la financiera, podrían tomarse decisiones equivocadas y perjudiciales. Para los individuos es de especial importancia que su información sea privada o que permanezca anónima cuando así lo deseen. Debido a que la información es un bien tan valioso, es necesaria su protección.

Las personas desde el surgimiento de la escritura han tenido la necesidad de comunicarse y transmitir información que debe permanecer oculta o secreta. Algunas muestras de esto las encontramos en las civilizaciones: Egipcia, Griega y Romana.

La *criptografía* es el estudio de las técnicas matemáticas relacionadas a aspectos de seguridad de la información y es una herramienta que nos puede ayudar a proporcionar los servicios de confidencialidad, integridad, autenticación y no repudio. De acuerdo a la definición anterior, la criptografía no solamente se ocupa del cifrado y descifrado de la información sino que también nos ofrece una amplia gama de algoritmos y protocolos. Algunos ejemplos prácticos de lo que podemos hacer con la criptografía son: asegurar que la información no ha sido alterada de forma no autorizada, que verdaderamente fue originada por la entidad adecuada, que un cierto mensaje fue generado en un cierto momento y no en otro, y probar a un tercero que alguna entidad efectivamente generó o efectuó cierta operación, por nombrar solo algunos ejemplos.

Históricamente es clara la importancia de la criptografía, especialmente en el siglo pasado durante la primera y segunda guerra mundial.

En la segunda guerra mundial los Ingleses y Polacos pudieron "quebrar" diferentes máquinas *Enigma* utilizadas por los Alemanes. Los Estadounidenses pudieron "quebrar" varios de los cifradores empleados por los Japoneses como: *Red*, *Orange* y *Purple*. La necesidad de romper estos cifradores dio un impulso muy importante al desarrollo de las primeras computadoras, por ello incluso se ha llamado a la criptografía como la madre de las ciencias computacionales [11].

Algunos ejemplos de usos de la criptografía que involucran directamente a México los tenemos en la experiencia de *Nicholas Trist* [14, pp. 21-36] y en la nota de *Zimmerman* [41].

Una de las primitivas criptográficas más importantes son los cifradores de bloques, los cuales toman un bloque de texto en claro y lo cifran mediante una clave produciendo un mensaje cifrado o *criptograma*. Los cifradores de bloque no solamente sirven para cifrar información. Pueden ser utilizados para construir otras primitivas criptográficas como las funciones *hash* y los *códigos de autenticación de mensajes*, esto gracias al empleo de alguno de los muchos modos de operación existentes.

En octubre del 2000, el *NIST* anunció que *Rijndael* se convertiría en el *AES* o *Estándar de Cifrado Avanzado*, que reemplazará al conocido y veterano *DES*. Esta decisión fue precedida por un proceso de selección abierto, en el que empresas e investigadores de todo el mundo pudieron proponer sus cifradores de bloque siempre que éstos cumplieran con algunos requerimientos generales. La evaluación y el criptoanálisis de estos algoritmos no fueron realizados por el *NIST* sino que corrieron por cuenta de la

comunidad criptoanalista abierta. El *NIST* con base en la información recolectada y con el apoyo de la opinión pública eligió a *Rijndael*.

Hipótesis

La hipótesis central de esta tesis es que el *Estándar Avanzado de Cifrado* (*AES*) es factible de ser implantado en una arquitectura real para *tarjetas inteligentes* que no fue evaluada durante el periodo de selección del *AES*, y de ser utilizado como primitiva básica para construir otros algoritmos o protocolos criptográficos que funcionen dentro de la tarjeta.

Conviene puntualizar un poco más nuestra hipótesis. La factibilidad de la implantación involucra varios aspectos. El sistema final ofrecerá tiempos de respuesta adecuados de acuerdo a los tiempos típicos documentados utilizando al *DES* y al *AES* en arquitecturas comparables.

Nuestra implantación deberá utilizar los recursos de la arquitectura eficientemente, de forma que pueda servir como componente de una aplicación real, en la que el *AES* será el núcleo de algunos algoritmos y protocolos criptográficos.

Cuando hablamos de una arquitectura real para *tarjetas inteligentes* nos referimos al sistema completo, que incluye, además del microcontrolador de las tarjetas a todos los componentes de *software* y *hardware* involucrados, tales como el sistema operativo de las tarjetas, los lectores de tarjetas y el sistema host.

Objetivos

Objetivo General

Estudiar y comprender la teoría básica de la criptografía, particularmente la basada en métodos simétricos y algunos tipos de protocolos o algoritmos que emplean este tipo de cifrado. Explorar la tecnología de las *tarjetas inteligentes*, conocer los distintos protocolos utilizados y sus aplicaciones más comunes. Estudiar y comprender a fondo al *AES* haciendo énfasis en las operaciones matemáticas involucradas pensando en su implantación y en el diseño de alguna aplicación.

Objetivos Particulares

- Conocer los mecanismos de seguridad de las tarjetas inteligentes fundamentados en técnicas de cifrado de clave simétrica.
- Estudiar el funcionamiento interno de las tarjetas, conocer sus sistemas operativos, los comandos que soportan, los protocolos utilizados, cuáles son sus principales aplicaciones y qué tecnologías alternativas existen.
- Estudiar las primitivas criptográficas más importantes desde el punto de vista de la criptografía simétrica.
- Conocer los modos de operación más comunes de los cifradores de bloques y sus propiedades.
- Estudiar la estructura y funcionamiento del algoritmo de cifrado *AES* poniendo atención especial en las operaciones matemáticas que lo constituyen y conocer de manera básica los principios que llevaron a sus creadores a desarrollar el algoritmo de esta manera.

- Explorar una arquitectura existente para tarjetas inteligentes poniendo atención en sus características, ventajas y limitaciones.
- Implantar al *AES* optimizándolo para alguna arquitectura, realizar simulaciones y evaluar el desempeño obtenido para lograr una comprensión profunda del algoritmo y de la arquitectura.
- Crear alguna aplicación básica que haga uso de nuestra implantación del algoritmo y que pueda servir como base para una aplicación más completa.

Justificación

Algunas formas de comunicación electrónica, como la *Internet*, han ganado en los últimos años mayor presencia en la vida de las personas y en las empresas. Hoy en día una buena proporción de la información importante es procesada por sistemas de cómputo. Esto trae con sigo nuevas oportunidades, como la posibilidad de acceder nuestra información desde cualquier lugar, pero también involucra riesgos.

Para un adecuado desarrollo de la sociedad de la información debemos ser capaces de proporcionar los medios que nos den confianza en las transacciones financieras, en la integridad de nuestros registros, en la confidencialidad de la información de las empresas y en la privacidad personal. Cada vez es más evidente la importancia de la información y es común que sea el activo más valioso para una empresa o para la persona que la posea. La criptografía da los cimientos para construir los mecanismos que evitarán que nuestra información sea alterada, observada o manipulada de forma no autorizada. Por ello es importante el estudio de los algoritmos criptográficos desde el punto de vista teórico y por otro lado "aterrizar" y poner en práctica estos conocimientos.

Las tarjetas inteligentes aparecieron originalmente como una solución para la identificación de personas pero posteriormente se les han dado otros usos. Es común utilizarlas para almacenar información confidencial, como mecanismo para firmar documentos o para guardar claves públicas de forma segura. Están ganando terreno a gran velocidad y es una tecnología muy prometedora que vale la pena conocer y estudiar.

Gracias a las características de seguridad de las tarjetas inteligentes es común que se les emplee como medios para contener dinero electrónico, tal es el caso de *Mondex*. El dinero electrónico implica una transacción electrónica de fondos de una parte a otra. Debido a su naturaleza, el dinero electrónico puede ser fácilmente objeto de intentos de robo y fraude. De acuerdo al *principio de Kerchoffs* la criptografía moderna debe fundamentar su seguridad no en un proceso de cifrado obscuro, sino en una clave. Solamente los protocolos criptográficos abiertos y cuidadosamente estudiados por la comunidad de criptoanalistas pueden ganar nuestra confianza.

El *AES* es un cifrador de bloques que reemplazará al *DES* y en muchos casos al tripe *DES*. El *AES* ofrece tamaños de clave que pueden hacerle frente a los recursos computacionales actuales de un posible adversario, además está diseñado para resistir los ataques generales más potentes del criptoanálisis que son el lineal y el diferencial. Es necesario comprender los algoritmos criptográficos que se utilizan en el presente y se utilizarán en los próximos años, especialmente cuando se espera que tengan tanta popularidad.

Por otro lado, la implantación de un algoritmo criptográfico de tanta importancia como lo es el *AES* en una tarjeta inteligente es altamente estimulante ya que siempre es un reto poder hacer un uso eficiente de los recursos tan limitados de este tipo de arquitecturas.

Alcance y Límites

El área general en que recae esta tesis puede considerarse que es la tecnología de la información y abarca principalmente temas de estudio como la seguridad informática, las tarjetas inteligentes y la criptografía. Nos interesa en particular el estudio del *AES* y las primitivas criptográficas que puedan derivarse del mismo con miras a su aplicación en las tarjetas inteligentes para mejorar sus esquemas de seguridad mediante un algoritmo que ha sido tan estudiado por la comunidad de criptógrafos y criptólogos.

La criptografía puede dividirse en simétrica y asimétrica (aunque algunos protocolos utilizan técnicas híbridas). Para el presente trabajo prácticamente se excluye a la criptografía asimétrica porque los diferentes problemas y la teoría matemática en que se fundamenta no son necesarios para la comprensión del *AES* y su estudio nos alejaría del tema central de la tesis. También evitaremos hablar de las primitivas que normalmente están basadas en este tipo de criptografía, tales como: las firmas digitales y las pruebas de cero conocimiento. Existe una gran cantidad de cifradores de bloque de clave simétrica, los cuales emplean operaciones de muy diversos tipos. Aquí trataremos de concentrarnos en el *AES* y solamente debido a su importancia y con fines comparativos haremos referencia al *DES* (*Data Encryption Standard*). En cuanto al estudio de *Rijndael*, abordaremos tanto al cifrador directo y al inverso, favoreceremos la comprensión de las transformaciones desde una representación matemática, sin embargo nuestra implantación solamente se ocupará del cifrado directo. El cifrado inverso se omitirá debido a que es más costoso computacionalmente (especialmente en arquitecturas de 8 bits) y es posible evitar su uso en muchas ocasiones mediante un modo de operación adecuado.

Con el propósito de dar una explicación más completa, trataremos de abordar múltiples aspectos de las tarjetas inteligentes. Los puntos que deben quedar bien establecidos son los protocolos, los comandos, el sistema de archivos de las tarjetas y los mecanismos de autenticación. Además de esto trataremos otros aspectos de las tarjetas y tocaremos algunas de las aplicaciones y estándares más importantes entre los que tenemos a *EMV*, *JavaCard* y *GSM*.

Respecto a la criptografía el tema central son los cifradores de bloques y para complementar su desarrollo hablaremos de los modos de operación más importantes. No entraremos en detalles de como deben implantarse estos modos de operación sino los trataremos de un modo más bien cualitativo y descriptivo.

Uno de los objetivos implica una implantación del *AES* en bajo nivel. Mientras podamos evaluar el desempeño del algoritmo y realizar pruebas consideramos suficiente con llegar a una simulación, pero debemos intentar llegar a una implantación física para corroborar con mayor precisión los tiempos de respuesta y comprobar en la práctica el funcionamiento de nuestros sistemas desarrollados.

Contexto y Conexión con Otros Problemas

La criptografía es un área multidisciplinaria en la que confluyen las matemáticas aplicadas, las ciencias de la computación y la ingeniería en computación. Dentro de cada una de estas disciplinas podemos distinguir en particular a: la estadística, la teoría de números, la teoría de grupos y estructuras algebraicas, la teoría de la complejidad, la teoría de la información, la seguridad en redes de datos, las comunicaciones digitales en general, la teoría de algoritmos, las técnicas para el diseño de *hardware* y la teoría de códigos por nombrar algunas.

La criptología abarca tanto a la criptografía como al criptoanálisis. A su vez la criptografía puede dividirse de forma general en dos grandes ramas: la criptografía simétrica (de clave secreta, simétrica, etc.) y la asimétrica (de clave pública). Los algoritmos de clave simétrica utilizan una misma clave (o bien la clave para descifrar puede derivarse fácilmente de la clave para cifrar) y el mismo tipo de operaciones

para cifrar y descifrar. Los algoritmos de clave pública utilizan claves diferentes para el cifrado y el descifrado, además normalmente las operaciones matemáticas involucradas son de distinta naturaleza. La posesión de la clave pública no debe significar que podamos derivar la otra clave.

Los algoritmos simétricos de cifrado pueden clasificarse a su vez en cifradores de flujos y cifradores de bloques. Los cifradores de flujos toman una pequeña cantidad de datos (por ejemplo un bit o un byte) de una fuente de datos y la cifran, mientras que los cifradores de bloques necesitan un bloque completo de datos (por ejemplo 128 bits) para poder obtener su criptograma.

Los cifradores de bloques son una primitiva muy importante debido a que con base en ellos pueden generarse otras primitivas como: códigos de detección de modificación, funciones *hash*, códigos de autenticación de mensajes y generadores de números pseudoaleatorios. En la práctica normalmente no se utiliza un cifrador de bloques directamente sino que se utiliza algún modo de operación con el propósito de obtener diferentes propiedades.

Si bien las técnicas criptográficas más formales hasta hace tiempo se ocupaban fundamentalmente para usos militares y diplomáticos, cada vez es más necesario su uso en el ámbito comercial y personal. Gran cantidad de información importante se encuentra almacenada o es procesada mediante sistemas de cómputo. Entre las necesidades que la criptografía ayuda a satisfacer se encuentra la necesidad de privacidad, la de restringir el acceso a recursos, realizar pagos electrónicos y poder tener confianza en las transacciones.

El manejo de claves ha sido un problema muy importante ya que para la criptografía simétrica es un requisito compartir una clave secreta con la otra parte de antemano. Desgraciadamente la criptografía asimétrica requiere frecuentemente procesos de cómputo más laboriosos y complejos. Además es común que el nivel de seguridad que ofrece un esquema de clave pública con una longitud de clave dada sea menor al nivel que podría ofrecer mediante un esquema de clave secreta. Es por ello que la criptografía simétrica en ocasiones se ejecuta en conjunción con la criptografía asimétrica, la cual por su naturaleza se presta para el intercambio de claves.

Otra área que está muy ligada con la criptografía y las tarjetas inteligentes es el comercio electrónico y en particular el dinero electrónico. Debido a las diferencias del "mundo real" con el "mundo digital" es necesario utilizar protocolos especialmente diseñados para evitar el fraude.

Las firmas digitales están relacionadas con los códigos de autenticación de mensajes, pero además permiten probar el origen de los datos a un tercero (no repudio). Regularmente las firmas digitales no se aplican sobre el mensaje original que deseamos firmar, sino sobre su extracto obtenido mediante una función *hash*.

Una aplicación importante de los cifradores de bloques es la generación de números pseudoaleatorios. Aquí cabe hacer la diferencia entre aquellos números que estadísticamente presentan propiedades de aleatoriedad y aquellos que están diseñados con el propósito de resistir técnicas criptoanalíticas.

Método

Trataremos de seguir un esquema en el que abordaremos los temas en orden de complejidad y especialización creciente. Proponemos comenzar el estudio con los conceptos generales de la criptografía con miras en los algoritmos simétricos y sus protocolos. Comprenderemos la teoría matemática que sustenta la especificación del *AES* haciendo énfasis en la representación de los datos, las operaciones y las estructuras algebraicas utilizadas. Estudiaremos de lleno al *AES* transformación por transformación abarcando al cifrador directo y al inverso. Debido a que nos podría desviar del tema central de la tesis solamente estudiaremos marginalmente las técnicas asimétricas.

En cuanto a las tarjetas inteligentes, se consultará una gran variedad de fuentes de información para obtener una comprensión extensiva de esta tecnología enfocándonos en los protocolos de comunicación, los sistemas operativos de las tarjetas, el sistema de archivos, los comandos, estándares y aplicaciones.

Seleccionaremos una arquitectura para tarjetas inteligentes, la estudiaremos y trabajaremos con las herramientas de desarrollo disponibles.

Materializaremos los conocimientos adquiridos implantando el algoritmo en dicha arquitectura y mejoraremos los mecanismos de seguridad de la tarjeta. También desarrollaremos una aplicación en el sistema anfitrión para experimentar con estos mecanismos.

Contenido

Este trabajo de tesis está dividido en 6 capítulos. En los capítulos 1 y 2 introduciremos información general con la que daremos al lector una base de conocimientos y antecedentes que facilitarán la comprensión de los capítulos posteriores.

Con el capítulo 1 nos familiarizaremos con diversos aspectos teóricos y prácticos de las tarjetas inteligentes. Describimos los tipos de tarjetas que existen, sus características, los protocolos utilizados y sus sistemas operativos. Hablamos sobre los componentes que integran un sistema basado en tarjetas inteligentes, sobre algunos estándares, algunos de los comandos que soportan, la comunicación con el sistema *host* y finalmente comentaremos sobre algunas de sus aplicaciones prácticas.

En el capítulo 2 abordaremos los antecedentes de criptografía del *AES* y de algunas primitivas. Desarrollaremos los conceptos fundamentales de la criptografía y explicaremos la teoría básica de la de clave simétrica. Haremos énfasis en los cifradores de bloques y algunos de sus modos de operación. Describiremos la teoría básica de las funciones *hash*, los códigos de detección de modificaciones y los códigos de autenticación de mensajes.

En cuanto a la autenticación describiremos los tipos existentes, poniendo atención especial en la autenticación débil y la fuerte por ser esenciales como mecanismos de seguridad para nuestras tarjetas. Explicaremos la teoría de las estructuras algebraicas utilizadas en *Rijndael* y que es la base para comprender la especificación original del algoritmo.

En el capítulo 3 examinaremos exclusivamente al *AES*. Primero hablamos un poco sobre su proceso de selección y luego explicamos detalladamente la especificación del algoritmo paso por paso. Haremos hincapié en el cifrador inverso y en la planificación de clave del cifrador, tanto directo como inverso, debido a que son las transformaciones menos documentadas y en las que más podemos aportar algo sin ser redundantes con la especificación del algoritmo. Si bien el algoritmo puede describirse de manera práctica desde la perspectiva de la programación, nosotros daremos preferencia a una descripción matemática empleando la teoría del capítulo anterior.

En el capítulo 4 trataremos nuestra experiencia al implantar el algoritmo en las tarjetas inteligentes. Hablaremos de las herramientas para desarrollar sobre la arquitectura, cómo las hemos utilizado y en ocasiones creado o adecuado. Explicaremos las características básicas de la arquitectura seleccionada y de que forma podemos trabajar con esta arquitectura tanto en simulación como en el sistema físico. Describiremos brevemente una primera implantación del algoritmo en lenguaje C, orientada a familiarizarnos aún más con el algoritmo, y después explicaremos como realizamos nuestra optimización en

ensamblador y adaptamos el código resultante al sistema operativo de la tarjeta. Adicionalmente a enfocarnos al desarrollo por el lado de la tarjeta, también mostraremos como realizar una aplicación del lado del *host* que nos permita administrar y manipular la información almacenada en nuestra tarjeta utilizando protocolos basados en nuestra implantación del algoritmo.

En cuanto a las operaciones matemáticas de *Rijndael*, mostraremos cómo creamos una librería destinada específicamente a trabajar con las estructuras algebraicas involucradas y cómo la utilizamos para derivar las transformaciones inversas de *Rijndael*, efectuar cada uno de los pasos del algoritmo y obtener una representación alternativa de la caja S y su inversa.

Presentaremos los resultados generales a los que llegamos en el capíulo 5. Evaluaremos los sistemas de *software* que desarrollamos, prestando atención especial en el desempeño de nuestras implantaciones del algoritmo. Compararemos el sistema resultante con el anterior y haremos referencia a la información documentada como típica para otras tarjetas y algoritmos de cifrado. Equipararemos los resultados finales con los objetivos de la tesis.

Por último, en el capítulo 6 expondremos las conclusiones a las que este trabajo de tesis nos llevó. También propondremos una serie de actividades de trabajo futuro que podrían enriquecer y dar continuidad a nuestro trabajo.

Capítulo 1

Tarjetas Inteligentes

Antes de abordar temas más complejos como el diseño de una aplicación completa basada en tarjetas inteligentes, es necesario que presentemos los conceptos fundamentales de esta tecnología. En este capítulo se tratarán una gran diversidad de temas relacionados directamente con las tarjetas inteligentes para sentar las bases teóricas y prácticas para la comprensión de los capítulos posteriores y poner en contexto a esta tecnología con respecto a otras.

Primero que nada se introducirá la definición de tarjeta inteligente, también conocida como *Smart Card* (SC), y se aclararán algunos términos que en ocasiones provocan cierta confusión. Se expondrán las clasificaciones más comunes de las tarjetas inteligentes y los conceptos técnicos generales más importantes que nos serán de utilidad para familiarizarnos con la forma en que operan en una aplicación real.

Se desarrollarán con mayor detalle los aspectos técnicos de más bajo nivel del funcionamiento de las tarjetas inteligentes de acuerdo a los estándares internacionales (especialmente de acuerdo al *ISO 7816*), abarcando desde las especificaciones de la conexión física y de la transmisión de datos hasta las de nivel de aplicación.

También se mencionarán algunas aplicaciones en las que las tarjetas inteligentes han tenido mayor éxito, desde un punto de vista no tanto tecnológico sino más bien en la industria.

En el apéndice A se proporciona información complementaria como algunos protocolos adicionales, los componentes de una aplicación real, estándares y cuál es el panorama en lo referente al futuro de las tarjetas inteligentes.

1.1. ¿Qué es una Tarjeta Inteligente?

El término *Tarjeta Inteligente* (TI) ha sido empleado para referirse a una gran cantidad de tarjetas plásticas que tienen contactos metálicos[1] y un circuito integrado incrustado.

En realidad a pesar de que externamente varios tipos de tarjetas se ven exactamente iguales y muchas veces es imposible distinguir, al menos fácilmente, que tipo de tarjeta estamos utilizando, en su interior pueden ser muy diferentes. Están integradas por componentes muy diversos, entre los que encontramos a las memorias *EEPROM*, *ROM*, *RAM* y *Flash*, microcontroladores, puertos seriales, lógica específica, relojes, coprocesadores criptográficos y muchos otros.

[1] Las *Tarjetas Inteligentes sin Contactos* serán discutidas posteriormente en la sección 1.2.

Aparte de la diversidad de componentes físicos, las tarjetas inteligentes pueden seguir diferentes protocolos y realizar una gran variedad de funciones. Por ello, para entender exactamente de que estamos hablando, es necesario acotar el significado preciso del término *tarjeta inteligente* o *smart card*.

Cuando la tecnología de las TIs comenzó a establecerse, el nombre con el que se les designaba en los estándares más importantes era el de *Tarjetas con Circuito Integrado* o *Integrated Circuit Cards (ICC)*[2]. Estas tarjetas se distinguen por contener un circuito integrado, el cual almacena la información de la tarjeta y en él se encuentra la lógica que permite acceder a esta información. El circuito integrado se encuentra incrustado en una tarjeta de plástico, la cual comúnmente es de 85.6 por 54 mm (ID–1), pero puede ser de distintos tamaños. Toda la comunicación con la tarjeta se realiza a través de un puerto serial, el cual generalmente es unidireccional (*half–duplex*[3]), o bien mediante señales electromagnéticas (*tarjetas sin contactos*).

El adjetivo calificativo *"inteligente"* no es igualmente adecuado para todas las TIs. Las *tarjetas de memoria*[4] son un tipo de tarjetas con circuito integrado que consisten a grandes rasgos en un conjunto de celdas donde se almacenan datos y una lógica que permite condicionar el acceso a los mismos. En ocasiones esta lógica es sumamente sencilla y permite obtener y alterar indiscriminadamente la información. En otros casos se requiere la autenticación del usuario mediante un PIN[5].

Hay tarjetas de memoria que implantan mecanismos de autenticación más complejos, pero estas tarjetas son en realidad menos utilizadas. El problema principal que presentan es que el control de acceso a los datos es sumamente simple, por lo que no son adecuadas para aplicaciones en que se requiera un buen nivel de seguridad, además la funcionalidad es muy limitada y no puede ampliarse mediante programas adicionales.

Por otro lado las *Tarjetas con Microcontrolador* (*Microcontroller Cards*) o *Tarjetas con Microprocesador* (*Microprocessor Cards*) permiten establecer esquemas más estrictos para el control sobre los datos. Debido al poder computacional de estas tarjetas, es posible implantar mecanismos criptográficos mucho más complejos que los empleados en las tarjetas de memoria. También ofrecen una mayor flexibilidad ya que en muchas ocasiones es posible cargar programas o *scripts* con lo que podemos desarrollar aplicaciones específicas. Debido a la mayor flexibilidad y seguridad que ofrecen, éstas son las tarjetas que más se merecen el adjetivo de *"inteligentes"*. A causa de esto algunos puristas prefieren reservar el término TI para las tarjetas con microcontrolador.

El uso de tarjetas de memoria o de microcontrolador no es indistinto ya que cada una ofrece diferentes ventajas y desventajas. Se discutirán las características que las distinguen con mayor detalle en la sección 1.2. En particular la razón más importante por la que las tarjetas de memoria son más utilizadas es por su bajo costo, por lo que son muy comunes en aplicaciones donde la tarjeta es desechada como en los sistemas de telefonía pública (no móvil). En sistemas que requieren una mayor flexibilidad y seguridad son más utilizadas las tarjetas con microcontrolador. Un ejemplo de ello es en la telefonía celular, donde las TIs son un componente esencial del estándar *GSM*.

A lo largo de todo el trabajo utilizaremos el término *tarjeta inteligente* para referirnos tanto a las tarjetas de memoria como a las de microprocesador, aunque en algunas ocasiones, por el contexto, será claro

[2] Otros nombres por los que son conocidas las TIs son *Chip Card, Carte à puce y Chipkarte*.

[3] Se están desarrollando también tarjetas con comunicación bidireccional simultanea (*full–duplex*).

[4] También conocidas como *Memory Cards* (MC).

[5] *Personal Identification Number, Card Holder Verification (CHV)* o bien, en Español *Número de Identificación Personal (NIP).*

que nos referimos exclusivamente a las tarjetas con microprocesador.

1.2. Clasificaciones de las Tarjetas Inteligentes

Como ya se mencionó en la sección 1.1, existen múltiples componentes que pueden constituir una TI, y por lo mismo pueden presentar características muy variadas. Esta diversidad de componentes significa que también hay una gran diferencia entre la funcionalidad que nos ofrecen.

Para poder escoger el tipo de tarjeta que más nos conviene para nuestra aplicación debemos conocer cuáles son los distintos tipos de tarjetas y cuáles son sus atributos principales. Después hay que seleccionar la tarjeta que se adapte mejor a nuestras necesidades y que tenga un costo razonable.

En esta sección se hablará de las principales clasificaciones de las tarjetas inteligentes, las cuales agrupan a las TI sde acuerdo a varios criterios[6]:

- De Acuerdo al Tipo de Circuito Integrado o Función.
- De Acuerdo al Método de Transferencia de Datos.
- De Acuerdo a la Forma y el Tamaño de las Tarjetas.

A continuación explicaremos con mayor detalle cada una de estas clasificaciones.

1.2.1. De Acuerdo al Tipo de Circuito Integrado o Función

Esta es la clasificación más elemental y divide a las tarjetas inteligentes en dos grandes grupos, los cuales a su vez serán subdivididos de acuerdo a otros criterios:

- Tarjetas de Memoria.
- Tarjetas de Microcontrolador.

Tarjetas de Memoria

Todas las tarjetas inteligentes almacenan datos. En el caso de las tarjetas de memoria, ésta es su función principal y la realizan sin implantar protocolos de seguridad avanzados. Típicamente la información es guardada en celdas EEPROM y existe una lógica sencilla que se encarga del control de acceso a los datos.

A este tipo de tarjetas también se les llama tarjetas síncronas. Se comunican con el dispositivo lector a través de protocolos poco flexibles y el lector tiene control preciso de cuando realizar la transferencia de datos.

Uno de los problemas más importantes de este tipo de tarjetas es que no hay una especificación que todos los fabricantes sigan para definir los aspectos técnicos necesarios para que las tarjetas sean compatibles entre sí. Esto causa que no sea sencilla la portabilidad de las tarjetas de un sistema a otro, además es común encontrar problemas con los dispositivos lectores que están diseñados para funcionar con un solo tipo de tarjetas[7].

A continuación presentaremos una subclasificación de acuerdo al tipo de protección de los datos:

[6]Los criterios para clasificación de Tarjetas Inteligentes no son del todo consistentes dentro de la literatura. En esta sección se trata de dar un enfoque general de estos criterios desde un punto de vista de la seguridad.

[7]Éste es el caso de las tarjetas telefónicas Francesas y Alemanas que no son compatibles a pesar de que ambas cobran en Euros.

- *Tarjetas sin Protección.*
  Todos los datos contenidos en la memoria pueden leerse y modificarse directamente, solamente es necesario mandar los comandos adecuados de acuerdo al protocolo de comunicación. Afortunadamente estas tarjetas no son muy utilizadas a causa de su evidente limitación.

- *Tarjetas Protegidas*
  Parte de la información almacenada en estas tarjetas es accesible únicamente después de que el usuario se ha autenticado. La autenticación normalmente es muy simple ya que solamente requiere conocer un PIN o algún código de seguridad sencillo. Esto da a pie a que posibles agresores obtengan acceso no autorizado a información protegida con solo conocer el PIN.

  La información es separada de acuerdo al nivel de protección deseado. Por ejemplo, una parte permanece sin protección[8], otra parte es accesible por el usuario y una tercera solamente es accesible por la compañía que manufacturó la tarjeta.

- *Tarjetas con Lógica Segura*
  La característica fundamental es que la tarjeta inteligente controla la forma en que la memoria es accedida y modificada[9].

  Estas tarjetas pueden contener otros mecanismos de seguridad como escrituras irreversibles (solamente disminuir el saldo disponible, nunca aumentarlo), uso de códigos de transporte (evitar modificaciones en la información de la tarjeta hasta que ésta llegue al usuario final) y protección de la información en caso de sustracción prematura de la tarjeta, entre otros.

  Por sus características, son las tarjetas más utilizadas en aplicaciones de telefonía pública fija, y como tarjetas prepagadas.

De acuerdo a los objetivos del presente trabajo de tesis, las tarjetas de memoria no son adecuadas ya que la lógica que gobierna su funcionamiento está preestablecida y no hay forma de cargar programas nuevos o aplicaciones. A continuación se presenta otro tipo de tarjetas mucho más interesante desde el punto de vista de la ingeniería en computación y que resuelve estas dificultades.

Tarjetas con Microcontrolador

Cuando es necesaria una mayor flexibilidad y poseer los mecanismos de seguridad más avanzados es más conveniente recurrir a las tarjetas con microcontrolador. Estas propiedades se derivan de su elevado poder computacional[10].

La característica principal de éstas tarjetas es que el acceso a la información se hace a través de la lógica de los programas que el microprocesador ejecuta. Gracias a esta lógica es posible implantar protocolos criptográficos que nos ofrezcan un mayor nivel de seguridad.

Los componentes de *hardware* y *software* que hacen lo anterior posible son:

[8] Un ejemplo de esto es la respuesta a reset o ATR que se discutirá en la sección 1.3.

[9] Se han detectado casos en los que esta lógica falla o no es utilizada adecuadamente. En Alemania un grupo de Holandeses logró recargar las tarjetas telefónicas agotadas y las vendió para ser reutilizadas; las pérdidas fueron de millones de Dólares [33].

[10] Todo depende de la perspectiva con la que se les mire. El poder computacional que poseen actualmente las tarjetas inteligentes es comparable con el de una PC de mediados de los 80. Claro que los circuitos integrados de aquellas computadoras ocupaban mucho más de los 25 mm^2 o menos de área que ocupa el circuito de las *tarjetas de microcontrolador.*

- *Microprocesador*
  Este componente permite que la tarjeta realice todo tipo de operaciones aritméticas y lógicas. Los microcontroladores más empleados son el *6805* de *Motorola* y el *8051* de *Intel*, los cuales tienen un microprocesador con longitud de palabra de 8 bits. Normalmente no se desarrolla un nuevo microprocesador, sino que se emplea uno que ya ha sido probado ampliamente, esto con el fin de reducir costos de desarrollo y pruebas. La mayoría de los microprocesadores empleados han sido de 8 bits, pero los de 32 bits están comenzando a emerger[11].

  En el caso del microcontrolador utilizado para este trabajo de tesis, se escogió uno de la familia *AVR* de *Atmel*, su microprocesador es de 8 bits, pero el *set* de instrucciones fue planeado con cuidado para facilitar la ejecución de las operaciones aritméticas básicas eficientemente. El tema de la arquitctura utilizada se tratará con más detalle en el capítulo 4.

  La arquitectura preferida hasta el momento es la CISC[12], aunque se está popularizando la arquitectura RISC[13] por permitir velocidades de operación más elevadas (normalmente no requieren dividir la frecuencia del reloj externo y el número de ciclos que ocupa cada operación es menor que en los CISC).

- *Memorias: RAM, ROM y de Almacenamiento Masivo*
  Las tarjetas inteligentes para hacer un mejor uso del área del circuito integrado, típicamente poseen varios tipos de memoria. A continuación se habla un poco de las memorias más comunes, claro que no todas las memorias se encuentran necesariamente en todas las tarjetas:

  - *Memoria de acceso aleatorio (RAM)*. Es la memoria de trabajo que emplean los programas o el sistema operativo para almacenar información durante la operación de la tarjeta. Esta memoria es normalmente volátil y su costo por bit es el más elevado, lo que ocasiona que la cantidad de este recurso sea muy pequeña. Por lo regular, la cantidad total de memoria *RAM* de las tarjetas está entre 128 y 256 bytes.
  - *Memoria ROM*. Contiene programas (el sistema operativo) y los datos estáticos asociados a éstos. Esta información se graba directamente en el semiconductor al momento de manufacturar el *chip*. El proceso de desarrollar una aplicación y fabricar la máscara es costoso y consume mucho tiempo. Por otro lado, si ya tenemos una máscara confiable y que satisface nuestras necesidades, el costo de fabricar grandes volúmenes de microcontroladores es más bajo.
  - *Memoria PROM*. Es similar a la *ROM*, pero permite que cierta información sea almacenada después de la manufactura de la tarjeta. Ejemplos de esta información pueden ser números de serie, fechas de expiración y claves, los cuales queremos escribir una sola vez.
  - *Memoria EEPROM*. Memoria no volátil empleada como medio de almacenamiento masivo. Tiene tres desventajas importantes: el tiempo de acceso que está en el orden de los ms, la gran cantidad de área que ocupa por bit y el alto requerimiento de potencia para efectuar las escrituras.

[11] En 1993, el proyecto CASCADE (*Chip Architecture for Smart Cards and portable intelligent Devices*) introdujo la primer arquitecutra de 32 bits para una TI.

[12] *Complex Instruction Set Computing (Computer).*

[13] *Reduced Instruction Set Computing (Computer).*

- *Memoria Flash*. Es una alternativa a las memorias *ROM*. Es similar a la memoria *EEPROM* pero tiene la ventaja de poder ser reescrita mucho más rápido y ocupa menos área por bit. Una desventaja de las memorias *Flash* es que su borrado se hace por bloques. Como podemos darnos cuenta el uso de las memorias Flash puede ser particularmente útil para depurar nuestras aplicaciones ya que podemos cargar nuevas versiones de nuestros programas en la tarjeta cada vez que lo deseemos con las señales eléctricas correspondientes.
- *Memorias FRAM*[14]. Son una tecnología prometedora como sustituto de las *EEPROM*. Se trata de memorias no volátiles con tiempos de acceso mucho menores que las *EEPROM* (del orden de los ns), el tiempo de escritura y lectura es el mismo (característica de las memorias *RAM*), su consumo de potencia es más bajo y el área por bit es menor.

El conocimiento de los distintos tipos de memoria que podemos encontrar en las tarjetas inteligentes es sumamente importante para poder desarrollar una aplicación exitosamente. En el capítulo 4, se tratará el tema de cómo seleccionamos las memorias que utiliza nuestra implantación de un algoritmo de cifrado.

- *Unidad de Comunicación Serial de Entrada y Salida*
  Puede pensarse en una tarjeta inteligente como en una computadora de capacidad limitada que se comunica con el exterior solamente a través de una interfaz serial. Existen dos tipos de comunicación que se discutirán más adelante cuando se hable de la clasificación de acuerdo al *Método de Transferencia de Datos*. A manera de adelanto, señalamos que se trata de la comunicación por contacto eléctrico y la comunicación por radiofrecuencias.

- *Sistema Operativo*
  Para que las tarjetas inteligentes puedan realizar las operaciones básicas, se requiere primero que nada de un sistema que se haga cargo de las funciones de bajo nivel directamente ligadas con el *hardware*. Este sistema es llamado *Sistema Operativo de la Tarjeta* o bien *COS*[15]. A diferencia de otros sistemas operativos para otras plataformas, el *COS* es sumamente compacto dado que está sujeto a restricciones muy importantes de memoria. Las funciones que realiza están de acuerdo con el *hardware* que tiene asociado.

  El sistema operativo provee una capa de abstracción sobre el *hardware* que permite que las aplicaciones tengan una mayor portabilidad. También hace más sencillo y rápido su desarrollo, además de reducir el número de errores.

  Los primeros *COS* eran monolíticos, difíciles de modificar y estaban enfocados a una sola aplicación. Posteriormente se desarrollaron los *COS* estructurados que están divididos en capas.

  Debido a que diversas aplicaciones tienen requerimientos distintos de funcionalidad y seguridad, es natural que se encuentren múltiples *COS* en el mercado, cada uno satisfaciendo cierto sector.

  Algunos ejemplos de los *COS* existentes son: *MULTOS*, *STARCOS*, *MFC*, *Cyberflex*, *PCOS*, *Windows for Smart Cards*, etc. por nombrar solo unos pocos.

  Una clasificación de los *COS* de acuerdo al tipo de aplicaciones que soportan distingue dos tipos:

[14] Memorias *RAM* Ferroeléctricas, *FRAM* o *FERAM*.
[15] *Card Operating System*.

- *COS de propósito general*. Implanta un conjunto de comandos estándar.
- *COS para aplicaciones específicas*. Tiene comandos especialmente diseñados para trabajar con un tipo de aplicación eficientemente.

Otra clasificación, con base en la funcionalidad que proporciona el *COS* en cuanto a aplicaciones, es la siguiente:

- *COS multifuncionales.* Es posible reducir el número de tarjetas inteligentes que cargamos con nosotros si permitimos que la misma tarjeta sirva para varios propósitos. Se trata de una sola aplicación capaz de realizar diferentes funciones. Por consiguiente hay una disminución en el costos si lo comparamos con el costo asociado a emplear una tarjeta para cada función.
- *COS con soporte para múltiples aplicaciones.* En los *COS* multifuncionales la aplicación la desarrolla y prueba una misma entidad. En caso de querer modificar alguna de las funciones es necesario alterar todo el contenido de la tarjeta. No hay mecanismos que permitan separar los datos que son usados por cada una de las funciones, de forma que si llegara a ejecutarse una función insegura se comprometerían los datos de toda la tarjeta. Por ello surgieron los *COS* para múltiples aplicaciones. Éstos aumentan la flexibilidad de la tarjeta ya que varios desarrolladores pueden crear aplicaciones que conviven en una misma tarjeta sujetos a restricciones que evitan que una aplicación acceda a los datos de otra. Si se desea eliminar o actualizar alguna de las aplicaciones, las demás no sufren ningún daño.

  Muchos sistemas operativos están optando por esta solución pero se presentan dificultades ya que los microcontroladores tradicionales no proporcionan mecanismos de hardware que permitan la separación de las aplicaciones en memoria ni para validar el código de las aplicaciones que se cargan. Por ello los *COS* para aplicaciones múltiples en el mercado se ven forzados a interpretar el código de las aplicaciones mediante una máquina virtual que realiza las comprobaciones de seguridad pertinentes.

El diseño del *COS* no es un proceso trivial. Corregir un error puede ser extremadamente difícil y costoso. Debe cuidarse la seguridad para evitar las vulnerabilidades al máximo, en particular deben evitarse las puertas traseras[16] ya que en cualquier momento pueden ser descubiertas y comprometer la seguridad de todo el sistema.

Las tareas principales de las que se encarga el *COS* son:

- Implantar las funciones de bajo nivel para comunicarse con el exterior mediante protocolos como *T=0*, *T=1*, etc.[17]
- Interpretar, ejecutar y responder a los comandos recibidos.
- Mantener el estado interno y el nivel de autenticación actual.
- Realizar la comprobación de la seguridad para restringir el acceso a los datos.
- Proporcionar librerías de algoritmos criptográficos.
- Detectar y recuperarse de condiciones de error.
- Administrar el sistema de archivos

[16] Funciones o accesos no documentados dejados por los desarrolladores principalmente para tareas administrativas.
[17] Más información sobre los protocolos de comunicación serial en la sección 1.3.

- Ejecutar comandos administrativos como deshabilitar tarjeta o desbloquear el NIP.

Debido a su importancia y cualidades vamos a mencionar brevemente dos de los *COS* más sobresalientes:

MULTOS
Se trata de un *COS* para aplicaciones múltiples. Su característica principal es que las aplicaciones, las cuales deben estar codificadas en *MEL*[18], son interpretadas por una máquina virtual, alcanzando independencia de *hardware*. Las aplicaciones pueden programarse directamente en *MEL*, o en diversos lenguajes como *C*, *Java*, *Visual Basic*, etc. los cuales se compilan con herramientas especiales que producen código *MEL* a la salida. Un programa cargador transfiere las aplicaciones a la tarjeta realizando comprobaciones de seguridad que involucran certificados de las aplicaciones, los cuales son manejados por una autoridad certificadora central.

Como puede observarse, MULTOS no es solamente un sistema operativo sino todo un ambiente de desarrollo. La seguridad alcanzable mediante MULTOS es de las mejores, soporta una gran variedad de algoritmos criptográficos incluyendo al *DES*, *triple DES*, *RSA* y basados en *ECC*[19].

MULTOS fue desarrollado por la única compañía que se ha atrevido a crear una aplicación de dinero electrónico 100 % anónima y no auditable, *Mondex*. Una aplicación con estas características requiere de niveles muy altos de seguridad ya que cualquier vulnerabilidad se presta para ocasionar fraudes multimillonarios.

Java Card
Primero que nada hay que enfatizar que Java Card no se trata de un *COS* sino de una API[20], pero hay diversas implantaciones. La idea principal es lograr independencia de la plataforma programando código que pueda ser ejecutado en diversas arquitecturas. Se emplea un subconjunto del lenguaje *Java* que reduce la complejidad del lenguaje, por ejemplo eliminando el polimorfismo y los arreglos multidimensionales.

El interprete no ejecuta los *bytecodes* del archivo *class* directamente sino que se le proporciona otro formato llamado *Card Applet* (archivo *.cab*) que es específico para cada Java Card y está formado por los *bytecodes* optimizados para una arquitectura específica y por las librerías ligadas estáticamente.

- *Otros Componentes*
  Además de los componentes anteriores, existen otros que no siempre están presentes pero que realizan operaciones muy útiles. Un ejemplo importante son los *coprocesadores*, los cuales al ser llamados mediante instrucciones especiales realizan alguna operación con los datos de la memoria y al finalizar devuelven el control al microprocesador. Las operaciones típicamente están relacionadas con la criptografía asimétrica y son muy intensivas en términos computacionales. Si se intentara ejecutar los mismos algoritmos pero solamente con las instrucciones del microprocesador, los tiempos de procesamiento serían inaceptables. Las tarjetas con coprocesador criptográfico permiten realizar autenticación de forma rápida y también permiten realizar algunas operaciones con tamaños de palabra más grandes que los manejados por el microprocesador.

[18] *MULTOS Excecutable Language*.
[19] Ver capítulo 2 para mayor información sobre estos algoritmos.
[20] Interfaz de Programación de Aplicaciones (*Application Programming Interface*).

Otro ejemplo de un componente que podemos encontrar en las tarjetas inteligentes, es un *generador de números aleatorios*. La generación de números aleatorios no es sencilla y en ocasiones requiere *hardware* especial. Es más sencillo generar números pseudoaleatorios. Los números aleatorios son importantes en aplicaciones criptográficas porque son empleados para generar claves. En ocasiones se utilizan registros de corrimiento retroalimentados[21] o semillas, las cuales son alteradas por factores externos como tiempos de retardo o información de las entradas y salidas.

Hay otros componentes que pueden ser muy útiles como lo es el *hardware* para realizar la comunicación serial, la cual muchas veces tiene que controlarse con rutinas de *software* implantadas en el sistema operativo. El *software* tiende a hacerse más complejo y esto puede propiciar errores de programación.

1.2.2. De Acuerdo al Método de Transferencia de Datos

Las tarjetas inteligentes, tanto las de memoria como las de microprocesador, requieren de alguna interfaz de comunicación con el lector[22]. Los tipos de tarjeta de acuerdo al método de transferencia empleado son:

- *Tarjetas con Contactos*
- *Tarjetas sin Contactos*
- *Tarjetas Combi o Híbridas*

Tarjetas con Contactos

El estándar *ISO-7816* especifica las características de los contactos eléctricos por donde se transmite la información y se alimenta a la tarjeta. En lugar de discutir los aspectos físicos de los contactos tales como su ubicación y la asignación de los contactos, estudiaremos los aspectos de los contactos relacionados con la seguridad.

Desde el punto de vista de la seguridad, los contactos pueden ser nocivos ya que al estar expuestos, son uno de los primeros lugares donde un atacante tratará de encontrar una vulnerabilidad. Es posible interceptar toda la comunicación entre la tarjeta y el lector mediante dispositivos especiales que se colocan físicamente en medio del lector y de la tarjeta.

Por otro lado, los circuitos internos pueden ser dañados si un voltaje externo demasiado alto es aplicado. En este caso, es posible que el microcontrolador opere de formas que los diseñadores no tenían pensadas, posiblemente abriendo una brecha de seguridad.

Tarjetas sin Contactos

Las tarjetas sin contactos reciben la información, señal del reloj y potencia de operación mediante señales electromagnéticas, no disponen de la fuente de alimentación de forma tan sencilla como las tarjetas con contactos. Obtener la potencia necesaria para su funcionamiento es la principal dificultad que deben superar, pero no es la única.

Las técnicas más frecuentes para la transmisión de datos son el acoplamiento inductivo y el capacitivo. Para el acoplamiento capacitivo, unas placas conductoras en la tarjeta funcionan como una de las

[21] *Feedback Shift Registers*.

[22] Otros nombres con los que se conoce al lector en Inglés son *Card Acceptance Device* (CAD) e *Interface Device* (IFD).

placas de varios capacitores. Este tipo de acoplamiento es útil para la transmisión de los datos pero no proporciona la energía necesaria para la operación de la tarjeta por lo que ésta se debe obtener de otro medio.

El acoplamiento inductivo es el más empleado. El cuerpo de la tarjeta contiene una bobina. El sistema puede concebirse como un par de bobinas con cierto acoplamiento, de forma que cuando se produce un cambio en el campo magnético producido por la bobina del lector, se puede detectar en la bobina de la tarjeta. La potencia es transmitida mediante una señal de alta frecuencia típicamente de 125 kHz o 13.56 MHz. La información es transmitida mediante técnicas como ASK (*Amplitude Shift Keying*), FSK (*Frequency Shift Keying*) y PSK (*Phase Shift Keying*).

Dependiendo del tipo de aplicación, los requerimientos de potencia varían. Las aplicaciones que escriben datos en la memoria necesitan mucha más potencia que las que solamente la leen, típicamente superior a los 100 μW para las tarjetas de memoria y 100 mW para las tarjetas de microcontrolador. Si la aplicación se limita a leer datos y transmitirlos, la potencia consumida es mucho menor y la distancia de operación entre la tarjeta y el lector se incrementa.

A continuación se describen los tipos de tarjetas sin contactos de acuerdo a la distancia a la que pueden estar de los lectores y las ventajas y desventajas que presentan.

- *De Proximidad Inmediata.* Estas tarjetas requieren que el dispositivo lector y las tarjetas se encuentren extremadamente cerca (menos de 1 mm de separación) y muy bien alineados.

- *De Proximidad o Acoplamiento Cercano.* La separación máxima entre el lector y la tarjeta es de hasta un centímetro. Se requiere que el consumo de potencia sea menor a los 150 mW. Una misma tarjeta puede emplear acoplamiento capacitivo e inductivo.

- *De Acoplamiento Remoto.* Las tarjetas con acoplamiento remoto operan a una distancia desde un par de centímetros hasta aproximadamente un metro del lector, la orientación de la tarjeta no es tan importante siempre que sea aproximadamente perpendicular al campo del lector. La ventaja principal que ofrecen estas tarjetas es que no es necesario que el usuario las lleve en la mano ni que la tarjeta sea insertada y sacada para funcionar. Esto es muy útil en aplicaciones como el control de acceso, y el pago de tarifas (como las del transporte público) ya que permite agilizar el proceso. La mayor flexibilidad de este tipo de tarjetas trae con sigo también desventajas. En los tipos anteriores de tarjetas sin contactos, la relación entre tarjetas y lectores era de uno a uno, pero con las tarjetas de acoplamiento remoto es posible que varias tarjetas se encuentren en el rango de operación del lector. Por ello es necesario establecer mecanismos para evitar colisiones.

  También hay que considerar que puede haber más de un lector escuchando pasivamente la comunicación, por lo que hay que tener cuidado de no proporcionar información útil sin cifrar como claves de sesión.

  Por otro lado, al usuario de la tarjeta por lo regular le gusta tener el control de las operaciones que se realizan con ella. Por ejemplo, no sería conveniente que se hiciera una transacción financiera sin el consentimiento del usuario.

Tarjetas Combi o Híbridas.

Una de las características que hace más versátiles a las tarjetas inteligentes es la de poder ejecutar diferentes aplicaciones con una misma tarjeta. Las tarjetas combi tienen los dos tipos de interfase mencionados anteriormente. Por ello pueden emplearse en aplicaciones que requieran tanto tarjetas con contactos (insertándola en el lector) como sin contactos (acercándola a la región del campo del lector).

Una posible forma de soportar ambos tipos de aplicaciones es mediante una tarjeta que tenga dos circuitos integrados independientes, uno para cada interfaz. Desde el punto de vista de la seguridad esta es la solución más conveniente, desafortunadamente es más costosa que emplear un solo circuito.

La solución más utilizada es llamada de interfaz dual. Consiste en que un solo circuito soportaambos tipos de interfaces. Debe existir una lógica que permita dar prioridad a un tipo de interfaz sobre el otro. Es posible que alguna aplicacion desee utilizar solamente una de las interfaces.

1.2.3. De Acuerdo a la Forma y el Tamaño de las Tarjetas

Debido a la gran cantidad de aplicaciones en las que las tarjetas inteligentes pueden ser utilizadas, es natural que existan otros formatos diferentes al más conocido que es la de las tarjetas de crédito. Por ejemplo, en el caso de la telefonía móvil *GSM*[23], que emplea a las tarjetas inteligentes como *SIMs*[24], sería poco práctico emplear tarjetas de tamaño regular ya que no cabrían en los teléfonos celulares modernos.

Veamos cuales son los formatos más comunes que podemos encontrar:

Formato ID-1

Éste es el formato más conocido y corresponde al de la forma de las tarjetas de crédito, tiene bastante tiempo de utilizarse y el estándar que lo regula es el *ISO-7810*, que trata específicamente de las tarjetas de identificación. De hecho ID significa *"Identification card"*. No todos los fabricantes de tarjetas siguen al pie de la letra al estándar, por lo que no es tan sencillo saber si una tarjeta es estrictamente *ID-1*.

El formato *ID-1* ha sido empleado por mucho tiempo y ha mostrado ser muy práctico debido a que cabe muy bien en la cartera, es flexible y no se pierde con facilidad.

Formato ID-00 (*Minicards*)

Estas tarjetas son más un poco más pequeñas que las *ID-1* y son más fáciles de manipular. Su menor tamaño las hace más rígidas, difíciles de romper y son más baratas de manufacturar. El problema con estas tarjetas es que no hay un solo estándar que sigan todos los fabricantes.

Formato ID-000 (Módulos)

Las tarjetas *ID-000*[25] ofrecen el menor tamaño que sigue respetando la zona de los contactos. Su tamaño reducido las hace apropiadas para ser empleadas como *SAMs*[26] montadas en circuitos impresos o como *SIMs* en telefonía celular.

Es posible adaptar un formato de tarjeta ID pequeño para trabajar como si fuera más grande. Como el área de los contactos eléctricos es idéntica, todo lo que se necesita es empotrar la tarjeta en un adaptador. De esta forma podemos interactuar con una tarjeta *SIM* mediante un lector regular para tarjetas *ID-1*.

Algunas tarjetas de formato *ID-1* o *ID-00* pueden convertirse en formatos más pequeños. Estas tarjetas están diseñadas de forma que si presionamos la zona de los contactos, se desprenderá una tarjeta más pequeña.

[23] Global System for Mobile Communication. Se hablará sobre la tecnología GSM en la sección 1.4 sobre las aplicaciones de las TI.

[24] *Subscriber Identity Modules*.

[25] También conocidas como tarjetas *"plug-in"*.

[26] *Security Application Modules* o *Security Access Modules*.

Tarjetas Inteligentes con Forma de Llave

Esta forma tiene la ventaja de que aparte de la seguridad proporcionada por la tarjeta inteligente, también se añade la seguridad mecánica dada por su forma. Además como las personas ya están acostumbradas a utilizar llaves, su introducción y empleo es más natural.

Dispositivos USB

Debido a la popularidad de los dispositivos USB y al costo que implica instalar un lector de tarjetas convencional, es frecuente que se aproveche la infraestructura USB que ya se tiene instalada en lugar de invertir en hardware adicional, con lo que se ahorran recursos.

Claro que este tipo de tarjetas se comunicarán mediante protocolos de comunicación distintos. Las aplicaciones típicas de este tipo de tarjetas son la autenticación y el acceso a datos restringidos.

1.3. Aspectos, Características y Consideraciones Técnicas sobre las TIs

Se han descrito diversos temas relacionados con las tarjetas inteligentes en secciones anteriores y en los apéndices. Entre ellos se incluye su evolución (sección A.1), los tipos de tarjetas que existen, los componentes de una aplicación real completa (sección A.2) y diversos estándares (sección A.5). En esta sección se tratarán aspectos técnicos y se verá con más detalle el contenido del estándar *ISO 7816* incluyendo temas muy interesantes tales como: los protocolos de comunicación, la detección de errores, el sistema de archivos interno de las TI, la forma en que el lector y la tarjeta inician la comunicación, los comandos disponibles en la mayoría de las tarjetas y otros temas que no se habían tratado previamente.

1.3.1. Especificación de los Contactos

Es necesario que las tarjetas cumplan con el estándar *ISO 7816-1* para asegurar que la tarjeta se encuentre en la posición adecuada después ser insertada. El estándar *ISO 7816-2* especifica la ubicación de los contactos de a tarjeta de forma que queden alineados con los del lector. Esto es necesario para poder tener confianza en la conexión eléctrica y para que nuestra tarjeta funcione en correctamente en cualquier lector.

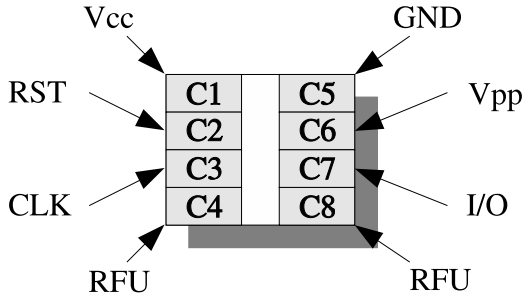

Figura 1.1: Disposición y función de los contactos de acuerdo a *ISO 7816-2*.

El *ISO 7816-2* indica que la tarjeta debe tener 8 contactos, los cuales están numerados como puede verse en la figura desde *C1* hasta *C8*. El uso de los contactos ha evolucionado ligeramente con el paso de los años y con la mejora de la tecnología disponible. Al momento de escribir el estándar, se consideró la

posibilidad de extenderlo en el futuro por lo que se especificó infraestructura adicional a la estrictamente necesaria.

Contacto	Nombre	Descripción
C1	*Vcc*	Fuente de Alimentación.
C2	*RST*	Señal de Reset.
C3	*CLK*	Señal del Reloj.
C4	*RFU*	Reservada para su Uso Futuro.
C5	*GND*	Tierra.
C6	*Vpp*	Voltaje de programación.
C7	*I/O*	Contacto para Entrada y Salida.
C8	*RFU*	Reservada para su Uso Futuro

Cuadro 1.1: Asignación de contactos de acuerdo al estándar *ISO 7816*.

Veamos con más detalle lo que significan estas señales:

- *Vcc* De aquí toma la tarjeta la energía que requiere. El voltaje más común es de 5V. Al igual que los demás contactos, es compatible eléctricamente con la tecnología TTL. Debido al uso de las tarjetas inteligentes en teléfonos celulares han aparecido tarjetas que requieren 3V.
- *RST*. Esta señal la manda el lector a la tarjeta para iniciar la comunicación. Tener varias señales como VCC y GND podría provocar cortos y daños a la tarjeta o al lector si la tarjeta no se encontrara colocada adecuadamente. Para asegurar que no se produzcan estos accidentes, se emplea la señal de *reset* como parte de un protocolo para iniciar y finalizar la comunicación (y la polarización).
- *CLK*. En la gran mayoría de los casos, la señal de reloj es proporcionada por el lector y es empleada por el microcontrolador de la tarjeta. En ocasiones esta señal se divide para hacerla más lenta y ajustarse al hardware interno. Hay que tener en consideración que un aumento en la frecuencia del reloj incrementa el consumo de potencia.
- *RFU*. Dos contactos no se utilizan: *C4* y *C8*, así que en ocasiones no están presentes en las tarjetas comerciales. Gracias a uno de ellos será posible en el futuro la comunicación *full–duplex*.
- *GND*. Se requiere para lograr la polarización correcta de la tarjeta.
- *VPP*. Hace tiempo se usaba este contacto para proporcionarle a la tarjeta el elevado voltaje que requería para programar la memoria EEPROM. En la actualidad, este contacto no se emplea debido a que el circuito integrado de las tarjetas ya contiene los dispositivos para generar este voltaje a partir del voltaje de alimentación.
- *I/O*. Este contacto es la principal vía de comunicación de la tarjeta. Todos los comandos que recibe y las respuestas que regresa pasan por aquí. Información más precisa sobre como logran la tarjeta y el lector comunicarse se dará más adelante.

1.3.2. Respuesta a Reset (*ATR*)

La *respuesta a reset* es una cadena o arreglo de bytes que transmite la tarjeta al lector y que contiene información diversa como el protocolo de comunicación que soporta y datos del fabricante.

Este arreglo de bytes siempre se transmite con el protocolo *T=0* (sección 1.3.3).

Cada bit se presenta en el PIN de entrada / salida durante una cierta cantidad de tiempo llamada *etu*[27] (Unidad Elemental de Tiempo). La *etu* es el resultado de dividir la frecuencia del reloj entre un divisor constante F[28] el cual normalmente es 372 o 512, con lo que tenemos una transferencia de datos a 9600 bits / segundo para relojes con frecuencias de 3.5712 y 4.9152 MHz respectivamente. Para la *ATR* siempre se emplea el divisor 372, pero posteriormente puede emplearse otro de acuerdo a lo indicado en la misma *ATR*.

Es posible que un solo lector soporte varios protocolos así que lo primero que éste debe hacer es determinar cuál es el protocolo que emplea la tarjeta presente.

La *ATR* está formada por los elementos que aparecen en el cuadro 1.2 y en total no deben sobrepasar los 33 bytes.

Byte	Descripción
TS	Carácter inicial que especifica la convención en uso.
T0	Carácter de Formato. Relacionado con los caracteres de interface e históricos.
TA_i, TB_i, TC_i, TD_i	Caracteres de Interfaz (opcionales).
$Ti \mid i = 0 \ldots k, k \leq 15$	Caracteres Históricos (opcionales).
TCK	Caracter de Comprobación (opcional).

Cuadro 1.2: Elementos que componen la *Respuesta a Reset* (*ATR*).

TS. Solamente puede tomar dos valores: 0x3B y 0x3F que corresponden a las convenciones directa e inversa respectivamente. La convención directa manda cada paquete de 8 bits con un bajo lógico representado como un voltaje bajo, y un alto lógico como un voltaje alto. Los bits de cada byte se mandan desde el menos significativo hasta el más significativo. En la convención inversa, los voltajes son los contrarios y se manda primero el bit más significativo y al final el menos significativo. El bit de inicio en ambas convenciones es el mismo, y es un nivel bajo.

T0. Se trata de 2 nibbles. El nibble alto indica la presencia de caracteres de interfaz. El nibble bajo contiene el número de bytes históricos presentes, el rango posible es de 0 a 15.

Bit(s)	Descripción
bit 8	Indica si se enviará TD_1 (bit 8 = 1) o no (bit 8 = 0).
bit 7	Indica si se enviará TC_1 (bit 7 = 1) o no (bit 7 = 0).
bit 6	Indica si se enviará TB_1 (bit 6 = 1) o no (bit 6 = 0).
bit 5	Indica si se enviará TA_1 (bit 5 = 1) o no (bit 5 = 0).
bit 4–bit 1	Número de caracteres históricos. Válido de 0 ('0000') a 15 ('1111').

Cuadro 1.3: Significado de cada bit de T0. El bit 8 es el más significativo.

Caracteres de Interfaz (TA_i, TB_i, TC_i, TD_i). Indican los protocolos de comunicación que soporta la tarjeta, son opcionales, y su cantidad es variable. En caso de que esta cadena no esté presente la comunicación es posible porque hay valores predefinidos para todos los parámetros.

[27] *Elementary time unit*.

[28] Formalmente llamado "factor de conversión de la tasa del reloj"

Hay dos tipos de caracteres de interfaz. El primer tipo es el global que trata sobre aspectos generales de interés para todos o la mayoría de los protocolos. El segundo tipo son los caracteres de interfaz específicos que solamente son útiles para un protocolo de comunicación.

Ahora describiremos cada uno de los caracteres de interfaz:

TD$_i$. Este carácter global funciona como liga para indicar la presencia de caracteres subsiguientes TA$_{i+1}$, TB$_{i+1}$, TC$_{i+1}$ y TD$_{i+1}$. Se necesita un mecanismo para indicar cuantos caracteres son enviados. Este mecanismo lo proporciona este byte, el cual además indica a que protocolo de transmisión se refiere.

Bit(s)	Descripción
bit 8	Indica si TD_{i+1} será mandado (bit 8 = 1) o no (bit 8 = 0).
bit 7	Indica si TC_{i+1} será mandado (bit 7 = 1) o no (bit 7 = 0).
bit 6	Indica si TB_{i+1} será mandado (bit 6 = 1) o no (bit 6 = 0).
bit 5	Indica si TA_{i+1} será mandado (bit 5 = 1) o no (bit 5 = 0).
bit 4–bit 1	Número de protocolo de transmisión. Válido de 0 (’0000’) a 15 (’1111’).

Cuadro 1.4: Significado de cada bit de TD$_i$.

TA$_1$. Carácter de interfaz global que indica el valor del divisor y el factor de ajuste de bit (empleados para calcular la *etu* con respecto a la frecuencia del reloj). TA$_1$ está compuesto por dos nibbles. El primero es FI y el segundo es DI de acuerdo a la siguiente tabla:

Bit(s)	Descripción
bit 8–bit 5	FI. Válido de 0 (’0000’) a 15 (’1111’).
bit 4–bit 1	DI. Válido de 0 (’0000’) a 15 (’1111’).

Cuadro 1.5: Nibbles que componen a TA$_i$.

FI	F	FI	F	FI	F	FI	F
’0000’	Reloj interno.	’0100’	1116.	’1000’	RFU*	’1100’	1536.
’0001’	372.	’0101’	1488.	’1001’	512.	’1101’	2048.
’0010’	558.	’0110’	1860.	’1010’	768.	’1110’	RFU[29].
’0011’	744.	’0111’	RFU*.	’1011’	1024.	’1111’	RFU.

Cuadro 1.6: Codificación del divisor de la frecuencia, *F*, de acuerdo a FI.

DI	D	DI	D	DI	D	DI	D
’0000’	RFU.	’0100’	8.	’1000’	RFU*	’1100’	1/8.
’0001’	1.	’0101’	16.	’1001’	RFU*.	’1101’	1/16.
’0010’	2.	’0110’	RFU*.	’1010’	1/2.	’1110’	1/32.
’0011’	4	’0111’	RFU*.	’1011’	1/4.	’1111’	1/64.

Cuadro 1.7: Codificación del factor de ajuste, *D*, de acuerdo a DI.

Hasta ahora no se ha especificado exactamente de que forma el divisor y el factor de ajuste afectan

la *etu*. Esto lo podemos determinar mediante la siguiente formula:

$$etu = F * T/D \tag{1.1}$$

donde T es el periodo del reloj externo y etu se refiere al tiempo que dura la transmisión de cada bit una vez que han terminado la *ATR* y el *PTS*.

De esta forma podemos ajustar desde la tarjeta la tasa de transmisión a diversos valores, siempre y cuando el lector también soporte dichas tasas.

TB_1. Este byte de interfaz global era usado cuando el lector proporcionaba voltajes especiales a través del contacto V_{pp} para programar la memoria EEPROM de las tarjetas. Como esta funcionalidad es ya obsoleta no se hablará más de este byte.

TC_1. Mediante este byte de interfaz global se codifica un valor extra del tiempo de guardia[30], el cual es el tiempo mínimo que podemos tolerar entre la transmisión de bytes sucesivos. El tiempo de guardia extra se denota como N, y es proporcional al valor del byte TC_1 tomando valores de 0 a 254 etu. Cuando TC_1 vale 255, el tiempo de guardia depende del protocolo empleado. Para *T=1* el tiempo entre bytes sucesivos es de 11 *etu* y para *T=0* de 12 *etu*.

TB_2. Para especificar al lector con mayor exactitud el voltaje de programación se emplea este byte de interfaz global. Al igual que TB_1 ya no es utilizado pero permanece en las especificaciones.

TC_2. Este byte se es específico para el protocolo *T=0* y permite especificar el tiempo de espera máximo entre bytes consecutivos de acuerdo a la ecuación:

$$\text{tiempo de espera} = 960 * D * \mathrm{WI} * etu \tag{1.2}$$

Donde WI equivale al valor del byte TC_2 y por default es 10. El valor por default de D es de 1 y el de la *etu* es $\frac{1}{9600}$s, por lo que el tiempo máximo de espera por default es de 1 segundo exactamente.

TA_2 Determina cómo se llevará a cabo la selección del tipo de protocolo (*PTS*) y será explicado en la sección A.4.

TA_i, TB_i y TC_i para $i \geq 3$. Bytes específicos para cada protocolo.

TA_i, $i \geq 3$. Este byte codifica el número máximo de bytes que puede recibir la tarjeta como parte de el campo de información (*IFSC*). Los valores válidos para este byte van de 1 a 254 y representan directamente el valor del *IFSC*.

TB_i, $i \geq 3$. Empleado para especificar otros parámetros de la transmisión de caracteres y bloques.

TC_i, $i \geq 3$. Especifica el método de corrección de errores. Solamente se utiliza el bit menos significativo y los demás están disponibles para otros usos.

Los dos métodos de corrección de errores son un CRC (bit 1 = 1) y un LRC (bit 1 = 0), ambos serán descritos en la sección sobre protocolos.

[30] Ver la sección 1.3.3 referente a los protocolos de comunicación.

$Ti, i = 0 \ldots k, k \leq 15$ *Caracteres Históricos*. Su uso no está estandarizado pero frecuentemente se emplean para indicar cuál es el fabricante de la tarjeta, la fecha de fabricación, la versión del sistema operativo o cualquier otra información relevante. Como máximo se permiten 15 caracteres históricos.

TCK. Caracter de Comprobación. Consiste en el XOR de todos los caracteres de la *ATR* comenzando con T0 hasta el último byte histórico si hay alguno, o en su defecto, el último byte de interfaz. Si la *ATR* solamente declara el uso del protocolo *T=0* el *TCK* es opcional, pero es obligatorio si se especifica el protocolo *T=1*.

1.3.3. Protocolos de Transmisión de Datos

Cuando se explicó la *ATR* y en el apéncice A.4, que habla sobre el *PTS*, se estableció que había un flujo de información entre el lector y la tarjeta pero no se dieron muchos detalles sobre como se logra esto. Entre los parámetros que se establecen mediante la *ATR* y el *PTS* se menciona un campo de cuatro bits el cual indica el número de protocolo. En esta sección se explicarán a detalle los dos protocolos más frecuentes.

Recordemos que la comunicación entre la tarjeta y el lector es de naturaleza serial y asíncrona (al menos para las tarjetas de microcontrolador). Esto significa que la transmisión de datos puede iniciarse en cualquier momento y no está ligada necesariamente a los pulsos del reloj, por ello se incluyen bits extra que indican el inicio y fin de la transmisión.

También hay que recalcar que el protocolo de comunicación empleado inicialmente para transmitir la *ATR* y en ocasiones el *PTS* (depende de lo que indique la *ATR*) es equivalente al protocolo *T=0* con un divisor fijo de 372. Después de que la *ATR* y el *PTS* han sido transmitidos, el protocolo de comunicación cambia en función de lo que hayan acordado el lector y la tarjeta a un nuevo protocolo, el cual generalmente es el mismo *T=0* o bien *T=1*. Como puede intuirse, todos los protocolos se denominan de la forma *T=#*, lo cual significa "Protocolo de Transmisión #".

Debido a que el protocolo se especifica mediante cuatro bits se tienen en principio 16 posibles protocolos diferentes. Veamos una lista de ellos:

Protocolo	**Descripción**
T=0	Comunicación asíncrona, half–duplex, orientada a bytes. Especificado en *ISO 7816-3*
T=1	Comunicación asíncrona, half–duplex, orientada a bloques de bytes. Especificado en *ISO 7816-3 rev 1*
T=2	Comunicación asíncrona, full–duplex, orientada a bloques de bytes. Especificado en *ISO 7816-4*
T=3	*Full–duplex*. Todavía no especificado.
T=4	Extensión de *T=0*. Todavía no especificado.
T=5 . . . T=13	*Full–duplex*. Todavía no especificados.
T=14	No es estándar ISO. Especificación para tarjetas telefónicas Alemanas (Telekom)
T=15	Todavía no especificado.

Cuadro 1.8: Descripción de los Protocolos de Transmisión.

Ahora veamos los dos protocolos de transmisión más utilizados incluyendo los mecanismos de detección y corrección de errores.

El Protocolo *T=0*

Debido a que éste es el protocolo más antiguo y que cuando se desarrolló los recursos eran muy escasos en términos del *hardware* de la tarjeta, este protocolo es el más sencillo de implantar y es el que incurre en la menor sobrecarga, consiguiendo maximizar la velocidad de transmisión.

El protocolo está orientado a bytes, esto significa que la unidad básica de comunicación es el byte. Cada byte se transmite independientemente, y el mecanismo de control de errores se ocupa solamente de un byte a la vez.

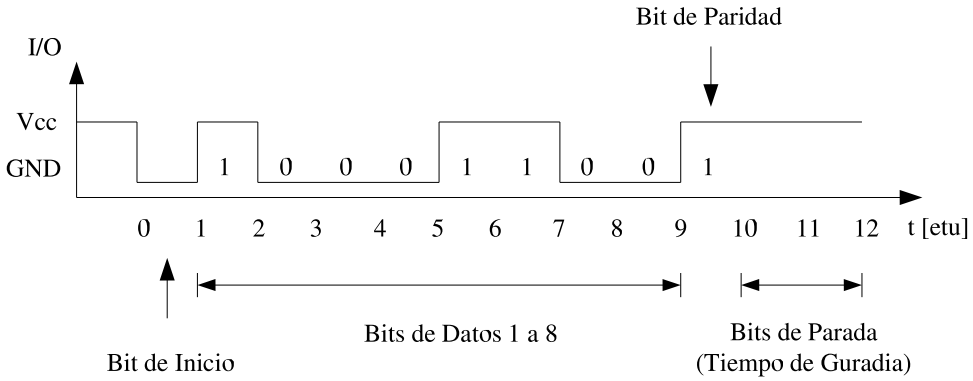

Figura 1.2: Transmisión del byte “00110001” o 0x31 siguiendo la convención directa.

Para transmitir cada byte se ocupan al menos 12 etu que corresponden a 12 bits. El primer bit es un bit de inicio bajo, seguido por 8 bits de datos. El siguiente bit proporciona la detección de errores y es un bit de paridad par. Después viene un *tiempo de guardia* de dos *etu* en el cual no debe transmitirse ningún byte (la línea permanece alta).

Nunca debería ocurrir una colisión entre el lector y la tarjeta en que ambos deseen transmitir información simultáneamente ya que el modelo de relación entre el lector y la tarjeta es maestro–esclavo. Esto significa que la tarjeta nunca iniciará la comunicación a menos que el lector le indique específicamente que lo haga, en cuyo caso éste esperará hasta que la tarjeta complete su respuesta.

La paridad par indica que el número de 1 (unos lógicos) debe de ser par tomando los ocho bits de datos y el bit de paridad.

La detección de errores consiste en simplemente obtener el XOR de los bits que va recibiendo y si el resultado no concuerda con la paridad esperada es indicio de que hubo un número impar de errores.

Cuando la tarjeta o el lector detectan un error, deben solicitar su retransmisión. Esto lo consiguen violando el tiempo de guardia al escribir un voltaje bajo cuando en condiciones normales hay un alto. El lector o la tarjeta detectan este voltaje bajo e inician la retransmisión del byte después del tiempo que toma mandar dos bytes.

El mecanismo de detección de errores descrito tiene la ventaja de poder ser implantado de manera muy eficiente y la sobrecarga del canal es muy baja. La principal desventaja que presenta es que la detección de errores no es muy adecuada porque si se presenta un número par de errores el sistema no tiene forma de detectarlos. Otra desventaja es que no hace una separación clara entre las capas de enlace de datos y la de aplicación, ocasionando problemas cuando se desea implantar mecanismos diferentes en la capa de aplicación como lo es la comunicación con datos cifrados.

En realidad el protocolo *T=0* no se encarga solamente de la transmisión de datos, sino que también se ocupa de otras cuestiones como el control del voltaje de programación y los comandos que manda el

lector a la tarjeta.

El protocolo *T=0* especifica como son enviados los datos en paquetes llamados *TPDU*[31]. El lector transmite un paquete a la tarjeta indicándole la ejecución de un comando y ésta responde con otro paquete. Como puede observarse no hay una clara división entre el protocolo de transmisión de datos y el protocolo a nivel de aplicación.

Los comandos que forman parte del protocolo a nivel de aplicación serán discutidos más adelante cuando se hable de los comandos de la tarjeta y la *APDU*[32].

El Protocolo *T=1*

Este protocolo presenta algunas ventajas sobre el protocolo *T=0*, en particular la detección de errores es más sofisticada y aquí sí se hace la separación entre las capas de enlace y la de aplicación.

La transmisión de datos está orientada a bloques de bytes. Existen tres tipos de bloques:

- *Bloque de Información.*
  Permite la transferencia de datos a nivel de aplicación.
- *Bloque de Notificación de Recepción.*
  Notifica que el bloque fue correctamente recibido o bien indica que se detectó un error en el bloque.
- *Bloque de Sistema*[33].
  Utilizado para transmitir información de control sobre el protocolo mismo.

Cada uno de los tipos de bloques descritos tiene la misma estructura, la cual se presenta en el cuadro 1.9.

Campo de Prólogo			**Campo de Información**	**Campo de Epílogo**
Dirección de Nodo, *NAD*	Byte de Control de Protocolo, *PCB*	Longitud, *LEN*	*APDU*	*EDC*
1 byte	1 byte	1 byte	0–254 bytes	1–2 bytes

Cuadro 1.9: Composición de los bloques del protocolo *T=1*.

Examinemos la construcción de cada uno de estos campos:

- *Campo de Prólogo.*
  Este campo se compone de 3 bytes:
  - *Dirección de Nodo (NAD*[34]*).* Permite identificar el nodo fuente y destino del bloque, esto es útil en sistemas que permiten múltiples flujos.
  - *Byte de Control del Protocolo (PCB*[35]*).* Especifica el tipo de bloque mandado y algunos parámetros de comunicación. Dependiendo del tipo de bloque, el significado de los bits que forman este byte varía de acuerdo a los cuadros 1.11, 1.12 y 1.12.

[31] *Transmission Protocol Data Unit*.
[32] *Application Protocol Data Unit*.
[33] También conocido como “Bloque de Supervisión”.
[34] *Node Address*.
[35] *Protocol Control Byte*.

Bits	Nombre	Función
bit 8 y bit 4	*Vpp*	Controlan el voltaje de programación. (En desuso).
bits 7, 6 y 5	*DAD*	Especifican la dirección destino.
bits 3, 2 y 1	*SAS*	Especifican la dirección fuente.

Cuadro 1.10: Descomposición del byte *NAD* (Dirección de Nodo)

Bit(s)	Función
bit 8	Para bloques de información vale '0'.
bit 7	N(S). Número de secuencia de envío (módulo 2).
bit 6	M. Indica si hay encadenamiento (M=1) o no (M=0). Usado para transmitir información con más de un bloque de datos (encadenamiento).
bit 5–bit 1	Reservados.

Cuadro 1.11: Significado del byte *PCB* para un bloque de información.

Bit(s)	Función
bit 8 y bit 7	Identificador de este tipo de bloque, vale '10'.
bit 6	Siempre debe valer '0'.
bit 5	N(R). Número de secuencia de bloque de notificación de recepción (módulo 2).
bit 4 y bit 3	Siempre deben valer '00'
bit 2 y bit 1	Pueden tomar tres valores: '00' si no hay ningún error. '01' si se detectó un error mediante el EDC. '10' indica otro tipo de error.

Cuadro 1.12: Significado del byte *PCB* para un bloque de recepción.

- *Longitud (LEN).* Indica la cantidad de bytes que vendrán en el campo de información. Los valores válidos son de 0 a 254, el valor 255 está reservado.

- *Campo de Información.*
  Para bloques de información contiene los datos de la aplicación, los cuales a diferencia de lo que sucede con el protocolo *T=0* tienen total libertad en cuanto a su formato y la interpretación que se haga de ellos.

  Para bloques de sistema contiene parámetros como el tamaño del campo de información o algún valor como el tiempo máximo de espera entre caracteres o el tiempo máximo de espera entre bloques. La longitud de este campo es variable y depende del valor del byte LEN contenido en el campo de prólogo.

- *Campo de Epílogo.*
  El campo de epílogo es la última parte del bloque transmitido y con él se realiza la detección de errores. Se compone de uno o dos bytes dependiendo del algoritmo empleado. Los dos algoritmos utilizados son un *LRC* o un *CRC*. El *LRC* (*Longitudinal Redundancy Check*) es el XOR de todos los bytes anteriores y es el algoritmo más empleado debido a los pocos recursos computacionales que requiere. El *CRC* (*Cyclic Redundancy Check*) genera un par de bytes mediante el polinomio divisor $G(x) = x^{16} + x^{12} + x^5 + 1$. Este algoritmo será explicado en el siguiente capitulo.

Bit(s)	**Función**	
bit 8 y bit 7	Identificador de este tipo de bloque, vale '11'.	
bit 6–bit 1	Especifican el tipo de operación a realizar.	
	'000000'	Petición de resincronización (desde el lector).
	'100000'	Respuesta a la resincronización (desde la tarjeta).
	'000001'	Petición para cambiar el tamaño del campo de información (desde el lector).
	'100001'	Respuesta a la petición de cambio de tamaño del campo de información (desde la tarjeta).
	'000010'	Petición para abortar. (desde el lector)
	'100010'	Respuesta a la petición para abortar. (desde la tarjeta)
	'000011'	Petición para modificar el tiempo de espera. (desde el lector)
	'100011'	Respuesta a la petición para modificar el tiempo de espera (desde el lector)
	'100100'	Indica un error en el voltaje de programación. (desde la tarjeta)

Cuadro 1.13: Significado del byte *PCB* para un bloque de sistema.

En caso de que un error sea detectado, no hay forma de corregirlo directamente (no se emplean códigos correctores de errores), sino que se sigue el siguiente procedimiento para la retransmisión del bloque completo.

1. La parte receptora del bloque (ya sea la tarjeta o el lector) calcula el *EDC*[36], lo compara con el *EDC* recibido y si son diferentes se da cuenta de que se produjo un error en el bloque.
2. La parte receptora manda un bloque de notificación de recepción donde señala este error al emisor.
3. El emisor mandar de nuevo el bloque.
4. Si el bloque es recibido correctamente, la transmisión de datos continúa. Si no es posible reestablecer la comunicación adecuadamente (sin encontrar errores), el lector hace una petición de resincronización (mediante un bloque de sistema).
5. La tarjeta da la respuesta a la petición de resincronización. En este caso se reinicia el estado del sistema de comunicación tal como se encontraba justo después de la *ATR*.
6. En caso de que el lector se dé cuenta de que la conexión no pudo ser reestablecida adecuadamente, se ve forzado a reiniciar la tarjeta mediante la señal de reset.

1.3.4. El Sistema de Archivos

Una de las características que hace que las tarjetas inteligentes sean muy superiores a las tarjetas con cinta magnética desde el punto de vista técnico es la cantidad de memoria que pueden almacenar. Mediante la memoria es posible almacenar los datos de las aplicaciones y del sistema operativo. No se trata solamente de un medio de almacenamiento portátil, sino que además se protegen los datos mediante protocolos de seguridad.

[36] *Error Detection Code*.

La capacidad de almacenamiento de las tarjetas inteligentes es reducida debido a que el tamaño del circuito integrado está limitado a al rededor de $25mm^2$ y a que la memoria eleva el costo del mismo ya que ocupa mucha superficie, en especial la memoria EEPROM.

Cabe señalar que en esta sección no se tratará el tema de las tarjetas de memoria sino el de las tarjetas con microcontrolador. Las tarjetas de memoria tienen sus mecanismos propios, en muchas ocasiones propietarios y no estandarizados.

Desde el punto de vista del sistema operativo la memoria de la tarjeta podría organizarse de manera arbitraria, pero para facilitar el desarrollo de aplicaciones que funcionen con relativa independencia de la tarjeta se han desarrollado algunos estándares, principalmente el estándar *ISO 7816* en su parte 4 donde se especifica el sistema de archivos.

Debido a algunas limitaciones, el sistema de archivos no puede ser demasiado sofisticado. A pesar de esto se tienen diversas rutinas que permiten realizar las operaciones sobre archivos más comunes: creación, lectura, escritura y eliminación. A diferencia de algunos sistemas operativos en que el acceso a los archivos es indiscriminado, en las tarjetas inteligentes pueden especificarse las condiciones para permitir el acceso a los archivos y en ocasiones algunos atributos especiales.

El sistema de archivos es jerárquico, existen tres tipos de archivos que equivalen al concepto de directorios y archivos regulares. Los archivos son identificados por un nombre denominado identificador de archivo.

Para acceder al contenido de los archivos existe una gran variedad de comandos, sin embargo no es necesario que la tarjeta soporte todos ellos. Los comandos están diseñados para facilitar la comunicación entre la tarjeta y el lector o sistema *host*, por lo que la especificación de la interfaz de programación trata sobre paquetes de bytes. La *API* del sistema de archivos no está pues enfocada a ser usada por una persona sino por un sistema de cómputo, esto proporciona la ventaja de poder hacer un uso más eficiente de la vía de comunicación.

Veamos ahora el sistema de archivos de las tarjetas inteligentes con mayor detalle, comenzando por los tres tipos de archivos válidos:

- *Archivo Maestro (MF[37])*
  Toda tarjeta inteligente con un sistema de archivos debe contener exactamente un archivo maestro. El *MF* es el equivalente al directorio raiz de un sistema *UNIX*, es el directorio más alto en la jerarquía y puede contener cero o más archivos. El identificador del *MF* siempre es 0x3F00, de forma que las aplicaciones siempre tienen un punto de acceso estándar al sistema de archivos.

- *Archivo Dedicado (DF[38])*
  Desafortunadamente el nombre "Archivo Dedicado" no es muy descriptivo, un nombre que podría ilustrar mucho mejor el significado de este tipo de archivos es "Archivo de Directorio". El *DF* es un archivo que permite agrupar a otros archivos. En la práctica los *DFs* son utilizados para aislar los archivos de cada aplicación.

- *Archivo Elemental (EF[39])*
  Los archivos elementales almacenan los datos de las aplicaciones y en ocasiones del sistema operativo. Cada archivo es identificado mediante un identificador de archivo el cual no necesariamente es único en toda la tarjeta. Lo que sí se garantiza es que el identificador de archivo debe ser único dentro del directorio (*MF* o *DF*) que contiene directamente al archivo.

[37] *Master File*.
[38] *Dedicated File*.
[39] *Elementary File*.

El sistema operativo puede definir la estructura interna y las operaciones válidas del archivo. Existen cuatro tipos de archivos elementales:

- *Archivos Transparentes.* La estructura interna de estos archivos no está definida, simplemente se trata como una cadena de bytes. Para acceder a estos bytes se necesita un desplazamiento inicial y la cantidad de bytes a leer o escribir. Estos archivos son lo más parecido a un archivo regular de un sistema *UNIX*.
- *Archivos de Registros de Longitud Fija Lineales.* Estos archivos agrupan un número definido de bytes en un registro. Las operaciones se efectúan sobre los registros completos.
- *Archivos de Registros de Longitud Variable Lineales.* No siempre es posible conocer de antemano el tamaño de los registros por lo que los archivos con registros de longitud variable pueden ser más adecuados. La ventaja principal que ofrecen es que evitan el desperdicio de memoria, pero tienen la desventaja de que el proceso que debe realizar el sistema operativo es mucho más complejo.
- *Archivos Cíclicos.* Los archivos cíclicos son muy similares a los archivos de registros de longitud fija, en cuanto a que la información se almacena en forma de registros y en que se permite direccionarlos relativamente a la posición actual con la diferencia de que la organización de los datos es como la de un búfer circular. Esto es, una vez alcanzado el "final" del archivo, automáticamente se continúa por el "principio".

  El acceso a los registros individuales puede hacerse independiente de la posición real del registro dentro del archivo, para ello se utilizan ordenes que hacen referencia a la posición relativa del registro de acuerdo a la posición actual.

  Los archivos cíclicos presentan la ventaja de que distribuyen el uso de las celdas de memoria más o menos uniformemente. Esto es conveniente porque la memoria EEPROM se degrada durante el proceso de escritura y si una sola celda es usada intensivamente[40], se corre el riesgo de dañar este byte y dejar inservible a la tarjeta.

  También facilitan la creación de archivos de registro en los cuales se escribe un resumen de las últimas transacciones efectuadas.

Existen otros tipos de archivos elementales, los cuales no necesariamente están regidos por el *ISO 7816-4* y que en ocasiones dependen del *COS*. Entre ellos se encuentran los *archivos ejecutables* y los *archivos de bases de datos*.

1.3.5. Comandos y Unidades de Datos del Protocolo de Aplicación (*APDUs*)

Desde el punto de vista del protocolo a nivel de aplicación, sería deseable que el intercambio de datos fuera totalmente transparente con respecto al protocolo de transmisión, haciendo una clara separación entre estas dos capas. Desgraciadamente esto no es siempre de esta forma.

Con el propósito de lograr la mayor eficiencia en cuanto a la minimización del flujo de datos, el protocolo *T=0* también hace referencia e impone ciertas restricciones a la forma en que el lector y la tarjeta intercambian comandos, los cuales corresponden a la capa de aplicación.

Para el caso de *T=1* sí se garantiza la independencia de los protocolos, específicamente se reserva en los bloques de información un área destinada a transmitir todos los datos del nivel de aplicación.

[40] En las tarjetas inteligentes típicamente se garantizan al menos 10000 operaciones de escritura de cada celda EEPROM.

Al utilizar diferentes protocolos de transmisión, no nos tenemos que preocupar demasiado del formato de las *APDUs* ya que estas no deben cambiar si nos apegamos al estándar *ISO 7816-4*.

El protocolo que hace posible el intercambio de datos a nivel aplicación se basa en la estructura llamada *APDU*. Este arreglo de bytes es el que permite controlar las operaciones que se realizan dentro de la tarjeta.

Como es de esperarse, debido a la gran cantidad y variedad de aplicaciones en las que se utilizan las tarjetas inteligentes, varios fabricantes han definido diferentes mecanismos y funciones, lo cual podría ocasionar incompatibilidad entre tarjetas de distintos fabricantes. Hasta cierto punto esto se regula gracias al estándar *ISO 7816* que en repetidas secciones especifica la funcionalidad o comandos que deben ofrecer las tarjetas y los protocolos empleados.

El protocolo a nivel de aplicación puede dividirse básicamente en dos partes: La primera se refiere a la estructura de los paquetes de datos y el lector y la segunda se refiere a qué acciones realizar de acuerdo al contenido de los paquetes.

La relación entre el lector y la tarjeta es maestro–esclavo. Esta asimetría ocasiona que los paquetes de datos o *APDUs* que se envían a la tarjeta (en ocasiones llamados *APDUs de comandos*) sean distintos a los paquetes que regresa la tarjeta al lector (también conocidos como *APDUs* de respuesta). La tarjeta siempre espera hasta que el lector le manda un comando mediante la *APDU* adecuada. La tarjeta entonces procesa el comando solicitado y contesta con una *APDU* indicando una condición de éxito, error o con la respuesta requerida.

Estos dos tipos de *APDUs* tienen una estructura distinta, veamos cada uno de ellos con mayor detalle:

- *APDU de comando.*
  Se compone de dos partes: La cabecera y el cuerpo.

  - Cabecera. Indica que comando deseamos ejecutar y proporciona algunos parámetros que pueden ser útiles. Consta de cuatro bytes:
    - *Byte de Clase (CLA)*. Permite identificar la aplicación y el conjunto de comandos. Su interpretación se muestra en el cuadro 1.14.

Bits	**Función**
bit 8–bit 5	Especifican a qué estándar corresponden los comandos. Por ejemplo: 0x0 estructura y códigos que cumplen con *ISO 7816-4* 0x8 estructura cumple *ISO 7816-4*, códigos privados 0xA se cumple con *ISO 7816-4* pero hay otros documentos que extienden la especificación como *GSM 11.11* y *EN 726*.
bit 4 y bit 3	Indican el uso de características de seguridad en los paquetes.
bit 2 y bit 1	Especifican a que canal lógico se dirigen los comandos. Esto es útil con tarjetas que soportan varios procesadores de *APDUs*

Cuadro 1.14: Interpretación del byte de clase (*CLA*) de las *APDUs* de comando.

    - *Byte de Instrucción (INS).* Especifica el comando a ejecutar. Debido a que *T=0* utiliza el bit menos significativo para controlar el voltaje de operación (funcionalidad obsoleta) se requiere que todas las instrucciones sean pares.

Byte INS	Nombre del Comando	Descripción	Estándar
0x84	GET CHALLENGE	Pide un número aleatorio a la tarjeta	*ISO 7816-4*
0xE0	CREATE FILE	Crea un archivo nuevo	*ISO 7816-9* *EN 726-3*
0xC0	GET RESPONSE	Pide la información de respuesta	*GSM 11.11* *EN 726-3* *ISO 7816-4*
0xB0	READ BINARY	Lee los datos de un archivo	*GSM 11.11* *EN 726-3* *ISO 7816-4*
0xD6	UPDATE BINARY	Escribe a un archivo.	*GSM 11.11* *EN 726-3* *ISO 7816-4*

Cuadro 1.15: Ejemplos de bytes de instrucción válidos.

En realidad hay decenas de comandos estandarizados como éstos. Cabe señalar que el sistema operativo de la tarjeta utilizada para esta tesis soporta, entre otros, cada uno de los comandos mencionados además de que como se tiene acceso al código fuente se pueden añadir nuevos comandos en caso de que sea necesario.

- *Bytes de parámetros 1 y 2 (P1 y P2).* En muchas ocasiones es necesario transmitir más información acerca del comando, los bytes *P1* y *P2* cumplen este propósito. Frecuentemente no se necesitan pero son obligatorios y se dejan en cero.

- *Cuerpo*. El cuerpo de la APDU no es necesario para todos los comandos. Cuando está presente puede componerse de tres elementos:
  - *Campo Lc (Longitud de Comando).* Indica la cantidad de bytes que serán transmitidos a la tarjeta.
  - *Campo de Datos.* Proporciona los datos adicionales necesarios para la ejecución del comando.
  - *Campo Le (Longitud Ejecutado).* Cantidad de bytes esperada que la tarjeta debe proporcionar como respuesta.

  Existen cuatro tipos de cuerpos, conocidos como casos generales:
  - *Caso 1.* No hay transferencia de datos así que el cuerpo está vacío, solamente se tiene la cabecera.
  - *Caso 2.* Transferencia de la tarjeta al lector. Se incluye solo el campo *Le*.
  - *Caso 3.* Solamente se mandan datos hacia la tarjeta. Campos *Lc* y de datos.
  - *Caso 4.* Es el más completo, se mandan *Lc* datos a la tarjeta y se espera un resultado de *Le* bytes.

- *APDU de respuesta*

  La respuesta de la tarjeta tiene una estructura más simple que la de la *APDU* de comando. Consiste solamente de dos partes:

  - *Campo de Datos.* Es opcional, su presencia depende del comando que haya sido enviado a la tarjeta y de que no se haya producido ningún error.

- *Palabras de Estatus (SW1 y SW2).* Son dos palabras de un byte cada una. Siempre se encuentran al final de la *APDU* y proporcionan el estado de la operación realizada. A pesar de que los códigos están estandarizados para estas palabras, en muchos casos se utilizan códigos propietarios. Algunos ejemplos de *palabras de estatus* son: 0x9000 (operación exitosa), 0x6A82 (archivo no encontrado) y 0x6962 (acceso denegado).

Existen decenas de comandos estandarizados los cuales son útiles para aplicaciones como la telefonía móvil, monederos electrónicos, sistemas de identificación, aplicaciones para clientes frecuentes o distinguidos, etc. A causa de que las tarjetas inteligentes están estrictamente limitadas en cuanto a la memoria disponible, normalmente se implantan solamente los comandos que son realmente necesarios para las aplicaciones en cuestión. Esto no quiere decir que se esté violando ningún estándar, en particular *ISO 7816* proporciona algunos perfiles de conjuntos de comandos que pueden implantarse pero no exige que todos estén presentes en una misma tarjeta.

En el cuadro 1.16 podemos observar los comandos más comunes.

Comando	**Descripción**
SELECT FILE	Permite seleccionar un archivo mediante su identificador (FID) para realizar sobre él futuras operaciones.
READ BINARY	Lee un cierto número de bytes de un archivo (previamente seleccionado) a partir de un desplazamiento dado.
UPDATE BINARY	Escribe un cierto número de bytes a un archivo a partir de un desplazamiento dado.
WRITE BINARY	Similar a UPDATE BINARY pero solamente modifica el estado de la memoria EEPROM de un nivel seguro a uno inseguro. Esto es, no se hace el borrado previo de la memoria. Este comando es útil para implantar contadores seguros que no puedan incrementarse.
SEARCH BINARY	Hace una búsqueda de una cadena en un archivo, si la encuentra devuelve su posición.
DECREASE	Reduce una cantidad dada un registro
CHANGE CHV (PIN)	Actualiza el NIP.
INTERNAL AUTHENTICATE	Permite verificar que la tarjeta es válida mediante un protocolo de reto / respuesta.
CREATE TABLE	Crea una tabla con las columnas dadas (para tarjetas que soportan SCQL (*Structured Card Query Language*).

Cuadro 1.16: Ejemplos de comandos típicos.

En la figura 1.3 podemos observar las *APDUs* que se transmiten al ejecutar un comando de ejemplo: Selección de un Archivo.

De acuerdo a lo que se ha descrito hasta ahora, no hay ningún mecanismo que evite que una entidad intermedia entre la tarjeta y el lector se percate de los datos que están siendo transmitidos o bien altere indebidamente la comunicación.

Para prevenir este tipo de ataques, hay que considerar que en este caso hay dos objetivos particulares los cuales son confidencialidad e integridad.

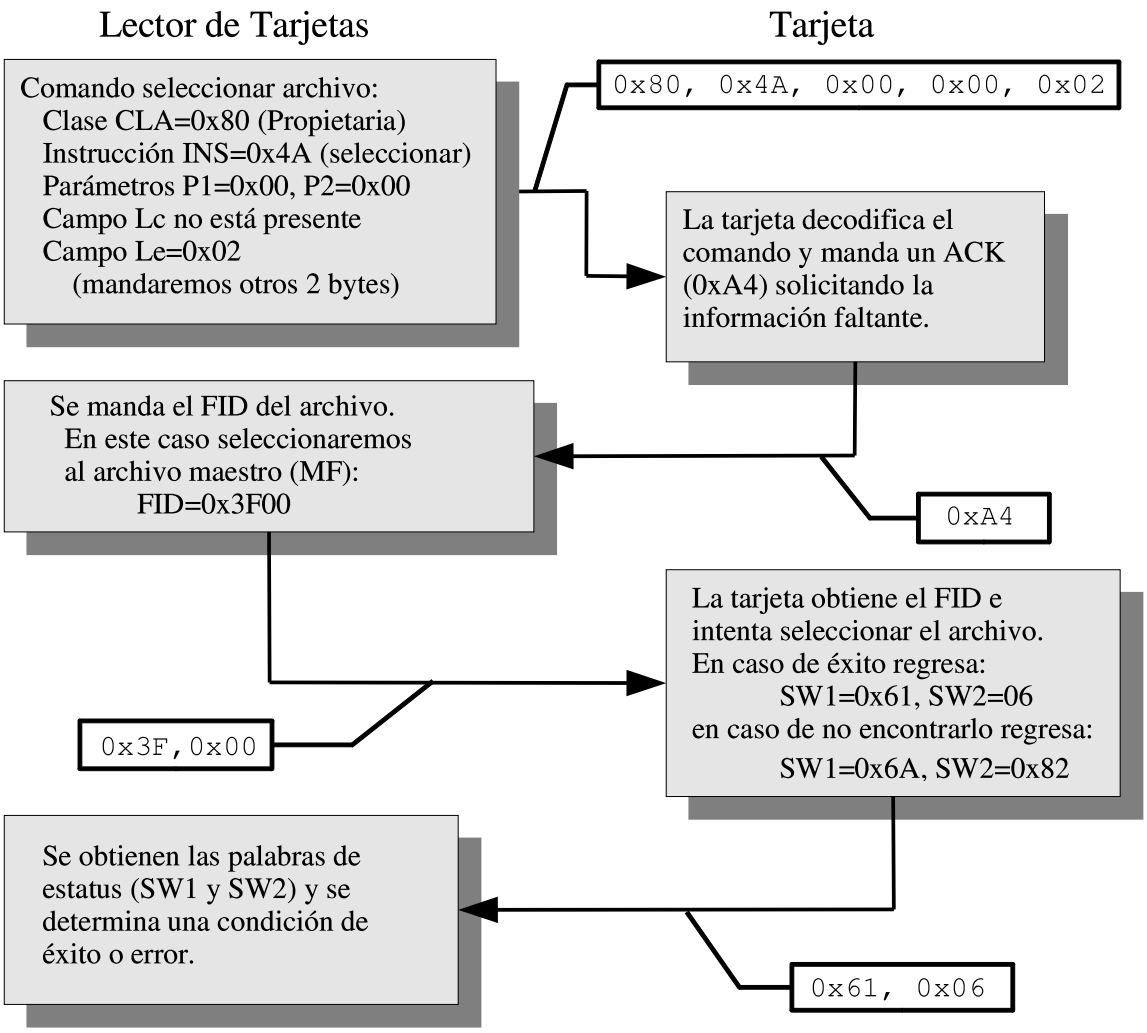

Figura 1.3: Comunicación entre el lector y la tarjeta al seleccionar un archivo.

Mediante el llamado *Modo Auténtico* es posible lograr la autenticidad de los mensajes empleando una suma de comprobación criptográfica que se anexa al final de todos los mensajes y que depende de una clave secreta que comparten la tarjeta y el lector o host.

El *Modo Combinado* ofrece además de autenticidad, confidencialidad ya que junto con la suma de comprobación criptográfica se transmiten todos los datos cifrados. Como no es posible cifrar el encabezado y seguir cumpliendo con el protocolo *T=0*, se creó un comando especial llamado ENVELOPE que en su cuerpo contiene la *APDU* que en realidad deseamos ejecutar y que sí está cifrada.

Estos dos mecanismos comúnmente se acompañan de un número de secuencia, el cual puede ayudarnos a evitar ataques por repetición de mensajes y dificultan el criptoanálisis ya que introducen una variable extra que hace que mensajes iguales sean cifrados de modo distinto.

Se hablará más sobre estos mecanismos de seguridad en capítulos posteriores.

1.4. Aplicaciones Típicas

Inicialmente el principal uso de las tarjetas con microcontrolador era la identificación personal, pero con el paso del tiempo han tenido éxito en otras áreas como en los monederos electrónicos, las tarjetas de crédito, los módulos de acceso de seguridad y telefonía móvil. Es de esperarse que el número de aplicaciones siga incrementándose conforme emerjan tarjetas con mayor capacidad (de memoria y procesamiento) y los costos se reduzcan.

En esta sección exploraremos las aplicaciones en las que las tarjetas inteligentes se han destacado más. Podemos encontrar mayor información en [24] donde el autor, *Mike Hendry*, hace un análisis muy completo sobre el tema. Por otra parte [35] habla sobre aplicaciones reales en donde las tarjetas inteligentes son un componente importante del sistema tal como el sistema de seguros médicos de Alemania, entre otros.

Las mayoría de las aplicaciones de las tarjetas inteligentes (incluyendo a las tarjetas de memoria) pueden dividirse de acuerdo a la siguiente clasificación:

- *Telefonía Pública Fija*
  Es una de las primeras aplicaciones de las tarjetas inteligentes. Surgió debido a la necesidad de superar los problemas que acarrea tener teléfonos que operan mediante monedas, como lo son: tener partes mecánicas móviles, los gastos para la recolección del dinero y el ser más propensos a sufrir de vandalismo.

  En un comienzo, en la época de las tarjetas telefónicas de primera generación, se empleaban tarjetas de memoria con una lógica muy simple. Posteriormente, con las tarjetas de segunda generación, se implantaron mecanismos de autenticación que hacen mucho más difíciles los ataques al sistema (como la emulación de tarjetas). Las tarjetas de tercera generación son muy similares a las de segunda generación en cuanto a los mecanismos que ofrecen con la diferencia de que puede emplearse un protocolo de autenticación de *reto y respuesta*.

- *Telefonía Móvil*
  El éxito de las tarjetas inteligentes en la telefonía móvil está directamente ligado a la gran aceptación de la tecnología *GSM* donde son empleadas como *SIMs*. Mediante los *SIMs* se busca autenticar al usuario y proteger la integridad y confidencialidad de la información personal y del sistema.

  Una de las características que distingue a los *SIMs* de las tarjetas inteligentes regulares es que se permite que la tarjeta inicie la comunicación con el teléfono. También tienen algunos archivos preestablecidos donde se almacena información como el último número marcado, el número de identificación de la tarjeta, el lenguaje preferido, etc.

  El estándar *UMTS*[41] define la tercera generación de teléfonos móviles y unificará a las tecnologías *PCS* (usada principalmente en USA y Japón) y *GSM*. *UMTS* también utilizará algún tipo de *SIM*.

- *Televisión Satelital*
  El estándar para el *broadcasting* digital de señales de video es llamado DVB[42], el cual transmite una señal de video MPEG-2[43] cifrada. Se tienen diversas claves para descifrar esta información (las cuales se encuentran dentro de una tarjeta inteligente) y mecanismos para actualizarlas.

[41] *Universal Mobile Telecommunications System*.
[42] *Digital Video Broadcast*.
[43] *Motion Picture Experts Group level 2*.

Uno de los problemas principales que enfrenta esta tecnología es que hay una cantidad muy grande de información cifrada disponible para cualquiera que tenga la antena receptora correcta. Por ello las claves simétricas con que se cifra el video son actualizadas constantemente con mecanismos de clave pública. En ocasiones las claves públicas han sido comprometidas por lo que pueden obtenerse las claves simétricas. Estas claves son distribuidas por Internet permitiendo a personas acceder a contenido ilegalmente. Este problema es difícil de solucionar porque las claves públicas comprometidas no son reveladas al público general, solamente las claves simétricas.

- *Acceso a Sistemas de Cómputo*
  Con acceso a sistemas de cómputo queremos decir no solamente a sistemas locales sino también posiblemente remotos, a recursos como discos duros o archivos, a programas de comunicaciones y de correo electrónico, etc.

  Frecuentemente se emplean passwords con este propósito, sin embargo esta solución presenta problemas ya que las personas tienden a escoger mal los passwords, olvidarlos, dejarlos visibles, prestarlos o pueden ser robados, además su longitud desde el punto de vista de la criptografía no ofrece tanta resistencia a ataques por fuerza bruta.

  Por ello es recomendable combinar varios mecanismos de autenticación como es una tarjeta inteligente, passwords e información biométrica.

  Las tarjetas inteligentes permiten, si son utilizadas adecuadamente, incrementar la seguridad del sistema, sobretodo en sistemas como la PC donde desde el punto de vista de la seguridad el ambiente es bastante hostil y no se puede tener mucha confianza en el sistema anfitrión. Las claves privadas más importantes, las cuales permiten la autenticación del usuario y el cifrado de información confidencial, no tienen que abandonar nunca la tarjeta, por lo que en teoría no es posible robarlas.

- *Aplicaciones Financieras*
  El manejo del dinero común y corriente trae con sigo varios problemas tanto para el consumidor como para quienes ofrecen un producto o servicio. Entre ellos tenemos que su manejo produce costos en ocasiones elevados, se pierde tiempo en colas, a veces no hay cambio y facilita los robos.

  Las tarjetas inteligentes gracias a sus mecanismos de seguridad son un medio ideal para complementar al dinero común.

  Existen principalmente tres tipos de sistemas de pago electrónico:

  - Tarjetas de crédito. El producto es obtenido antes de hacer el pago. Las tarjetas que habían sido empleadas anteriormente no son ya adecuadas por lo que se está migrando a las de tarjetas inteligentes.
  - Tarjetas de débito. El pago es efectuado en el momento en el que el producto o servicio es dado. El dueño de la cuenta muchas veces recibe intereses por el dinero de su cuenta y puede obtener su dinero en efectivo.
  - Monederos electrónicos. La diferencia fundamental que tienen con las tarjetas de débito es que en los monederos en realidad se emplea dinero electrónico que no equivale a dinero real. Con dinero real se compra cierta cantidad de dinero electrónico. El dinero real puede ser invertido y da intereses mientras que el dinero electrónico se queda congelado hasta que es usado, además puede no ser utilizado nunca.

Las tarjetas inteligentes no son una solución perfecta y hay características que ofrece el dinero (por ejemplo ser anónimo al realizar una transacción) que son muy difíciles de obtener con las tarjetas inteligentes. Se busca que el dinero electrónico sea: procesable, transferible, divisible, descentralizado, auditable, seguro y anónimo.

- *Aplicaciones de Salud*
  Se trata de proveer un medio extra de información médica o personal que pueda ser útil para los doctores o el seguro médico. En Alemania prácticamente toda la población (80 millones) porta una de estas tarjetas y en Francia más de 40 millones de tarjetas se han proporcionado a pacientes.

  Debido a la cantidad limitada de memoria que poseen las tarjetas inteligentes, no es posible incluir todo el historial médico de los pacientes, pero al menos es posible mantener la información más esencial y referencias a más información. Debe considerarse que la confidencialidad es una prioridad.

  Entre las ventajas que ofrecen las tarjetas inteligentes para esta clase de aplicaciónes está que pueden facilitar la prescripción de drogas restringidas, reemplazando las recetas de papel tradicionales por recetas electrónicas que pueden tener firmas digitales. También es posible monitorear al paciente (uso de medicamentos, registro de ejercicios físicos, presión sanguínea) y enviar esa información al doctor por Internet.

Capítulo 2

Antecedentes de Criptografía

2.1. Conceptos Fundamentales

2.1.1. Definiciones Básicas

El origen de la *criptografía* puede remontarse a miles de años atrás, y es prácticamente tan antigua como la escritura. Surgió por la necesidad de realizar una comunicación privada en los ámbitos comercial, religioso y principalmente militar.

La criptografía es el estudio de las técnicas matemáticas relacionadas con aspectos de la seguridad de la información como la confidencialidad, la integridad de los datos, la autenticación de entidades y la autenticación del origen de los datos [30].

La información original o *texto en claro* es transformada con el propósito de ocultarla o enmascararla en un *criptograma* o *texto cifrado* mediante el proceso denominado *cifrado*. Para recuperar el texto en claro a partir del criptograma se realiza el proceso inverso llamado *descifrado*.

Tanto la transformación de cifrado como la de descifrado dependen de al menos una clave, sin la cual realizar estas transformaciones en principio sería imposible.

Por otra parte, el *criptoanálisis* es la ciencia y estudio de los métodos para romper un cifrador [15], esto significa poder recuperar sistemáticamente el texto en claro o la clave empleada para el cifrado (de manera no autorizada). En realidad existen varios niveles en los que esto puede ocurrir: recuperación total de la clave, descifrado de mensajes (de todos o de algunos) y deducción parcial de información.

La *criptología* es la ciencia que comprende tanto a la criptografía como al criptoanálisis. Formalmente la criptología puede considerarse como una rama de las matemáticas, pero también está relacionada con la ingeniería y se le puede concebir como un arte. Respecto a esto los autores del algoritmo *AES* afirman: *"El diseño de cifradores es todavía más ingeniería que ciencia."* [13, p. 97].

El texto en claro o mensaje original se denota como m y pertenece al espacio de texto claro $\mathcal{M}$. El mismo es cifrado mediante la función de cifrado $E_k : \mathcal{M} \to \mathcal{C}$ con la cual obtenemos un criptograma c que pertenece al espacio de texto cifrado $\mathcal{C}$ dada una clave k que pertenece al espacio de claves $\mathcal{K}$.

Formalmente, un sistema criptográfico se especifica mediante la tupla ($\mathcal{M}$, $\mathcal{C}$, $\mathcal{K}$, ε, $\mathcal{D}$) donde los primeros tres elementos ya fueron explicados,
$\varepsilon = \{E_k : k \in \mathcal{K}\}$ es una familia de funciones de cifrado y
$D = \{D_k : k \in \mathcal{K}\}$ es un conjunto de funciones de descifrado.

Si llamamos a la clave para cifrar e y a la clave para descifrar d, entonces tenemos que:

$$E_e(m) = c \tag{2.1}$$

$$D_d(c) = m \tag{2.2}$$

Un algoritmo restringido es aquel en el cual la seguridad del sistema criptográfico se basa en el conocimiento de los algoritmos de cifrado y descifrado. En la práctica se ha comprobado que esta forma de buscar la seguridad no es la más adecuada. De acuerdo a la suposición de *Kerckhoffs* se debe asumir que un posible adversario conoce la clase de métodos empleados [1]

Los servicios de seguridad pueden clasificarse en:

- *Confidencialidad.* Se refiere a que ninguna persona o proceso pueda acceder, leer u obtener el contenido de cierta información sin tener la autorización adecuada. En ocasiones es necesario ocultar no solamente la información sino incluso su presencia.
- *Autenticación.* Busca la identificación de entidades, de forma que nos aseguremos que una entidad sea realmente quien dice ser o suponemos que es.
- *Integridad.* Consiste en que debe haber una correspondencia entre la información que tenemos respecto a la que deberíamos tener, es decir, que no ha sufrido alteración alguna.
- *No Repudio.* El no repudio evita que alguna de las partes niegue haber realizado alguna transacción o proceso, de forma que se puede demostrar a un tercero de que dicha transacción ocurrió.
- *Control de Acceso.* Restringe el acceso a recursos permitiéndolo solamente a ciertas entidades y con límites definidos.
- *Disponibilidad.* Consiste en que la información requerida esté disponible con oportunidad, esto es, cuando se requiera y tantas veces como sea necesario.

Existen dos grandes ramas dentro de la criptografía las cuales examinaremos a continuación.

2.1.2. Criptografía de Clave Simétrica

En esta clase de sistemas, también conocidos como de cifrado convencional, es relativamente sencillo obtener la clave de cifrado e a partir de la clave de descifrado d y viceversa.

Esto quiere decir que en realidad solamente se requiere el conocimiento de una de ambas claves para realizar el cifrado y el descifrado, con tan solo transformar la clave que tenemos disponible en caso de necesitarlo. En muchos sistemas es común encontrar que la clave de cifrado y la de descifrado son iguales, aunque esto no es un requisito.

En el caso particular del *AES* los algoritmos de cifrado y de descifrado comúnmente se definen de acuerdo a la misma clave, pero cuando se desea implantar el algoritmo en un dispositivo con poca memoria, tal como una tarjeta inteligente, es posible definir el algoritmo de forma que opere con una clave para cifrar y con otra equivalente para descifrar, reduciendo la cantidad de memoria empleada en la planificación de clave o bien la cantidad de operaciones a realizar. Esto se explicará con más detalle en capítulos posteriores.

Es importante resaltar que para la criptografía de clave simétrica, es necesario que previamente las partes se pongan de acuerdo a través de un canal seguro sobre la clave a utilizar.

Los cifradores de clave simétrica pueden subdividirse en cifradores de flujos o cadenas y en cifradores de bloques:

[1] Mayor información en la "Máxima número 3" de [6, p. 207].

- *Cifradores de Flujos.* Toman de un flujo de datos un fragmento pequeño, un símbolo de posiblemente 1 bit o 1 byte, y lo transforman individualmente. Este método de cifrado tiene la ventaja de que el cifrado o descifrado pueden iniciar sin que se tenga un *buffer* completo de datos disponibles, lo cual podría tomar cierto tiempo y hacer más lento el proceso. Existen dos tipos de cifradores de flujos:
  - *Síncronos.* En estos cifradores la clave con la que se cifra un carácter no depende completamente de los criptogramas anteriores, por lo que si en el momento de la transmisión se pierde o inserta algún dato, al momento de descifrar la información se obtendría basura indefinidamente ya que a partir del error se estaría descifrando cada carácter con una clave distinta a la clave con que se cifró. Este problema se resuelve mediante mecanismos de resincronización de las claves empleadas.
  - *Asíncronos o Autosincronizantes.* Se mantiene un estado interno que depende de los últimos n caracteres cifrados. Este estado junto con la clave secreta permiten cifrar y descifrar la información. En caso de que se reciba un carácter en error, dicho error se propagará a los siguientes n caracteres, pero después de que el elemento defectuoso salga del estado, la decodificación volverá a la normalidad.
- *Cifradores de Bloques.* La información se divide en grupos de bits llamados bloques, y es cifrada o descifrada un bloque a la vez. Comúnmente son empleados como un componente de los sistemas criptográficos, con los cuales se pueden construir códigos de autenticación de mensajes (MACs), funciones *hash*, etc. Debido a su importancia para esta tesis, serán descritos con mayor detalle en una sección posterior.

2.1.3. Criptografía de Clave Asimétrica

Los algoritmos de clave asimétrica o clave pública, emplean claves diferentes para efectuar el cifrado y el descifrado. Una de estas claves es llamada *clave privada* y la otra *clave pública*.

Se busca que no sea sencillo o práctico el derivar la clave privada a partir de la clave pública.

La clave pública no es necesario mantenerla secreta, y de hecho normalmente están disponibles para quien la necesite.

La clave privada puede ser empleada tanto para cifrar como para descifrar (pero no para ambas cosas a la vez). Si se desea recibir información de forma confidencial, se hace llegar la clave pública a las personas que queremos que nos manden tal información. Estas personas cifrarán la información mediante la clave pública y enviarán el criptograma: $c = E_e(m)$. Mediante la clave privada es posible recuperar el mensaje original: $m = D_d(c)$ donde e es la clave pública y d es la clave privada.

También es posible emplear la criptografía de clave pública para generar firmas digitales. En este caso el empleo de las claves públicas y privadas se invierte. Existen diversos esquemas de firmas digitales, pero a grandes rasgos consisten en cifrar con la clave privada la huella o *hash* de un cierto mensaje. Quienes posean la clave pública pueden comparar el *hash* descifrado del mensaje con el *hash* calculado por ellos. Si coinciden ambas huellas puede considerarse que efectivamente se firmó dicho mensaje.

Para ejemplificar lo anterior veamos una simplificación del algoritmo propuesto en [30, p. 429] en el cual **A** produce la firma $s^\star$ del mensaje m, la cual es verificada por **B**:

1. Generación de la firma.
   a) **A** selecciona un elemento k el cual especifica junto con la clave privada de **A** la transformación para firmar m.

b) **A** calcula una función unidireccional h sobre el mensaje $\widetilde{m} = h(m)$ que podría ser alguna función hash y crea la firma $s^\star = S_{A,k}(\widetilde{m})$ mediante k y la clave privada de **A**.

c) La firma del mensaje m es $s^\star$. Tanto el mensaje como su firma se proporciona a quienes deseen verificar la firma del mensaje.

2. **B** verifica la firma.

a) **B** obtiene la clave pública de **A**.

b) **B** calcula $\widetilde{m} = h(m)$ y $u = V_A(\widetilde{m}, s^\star)$ donde V_A es llamada función de verificación y depende de la clave pública. Si la variable booleana u es verdadera significa que $S_{A,k}(\widetilde{m}) = s^\star$, y en caso contrario es falsa. Esta transformación V_A puede emplear el descifrado con la clave pública de **A**.

c) Si u es verdadero, **B** acepta la firma.

Es importante señalar que la criptografía de clave simétrica y asimétrica en aplicaciones reales no se sustituyen entre sí, sino que se complementan, tomando las ventajas de ambos tipos de criptografía.

Lo más conveniente es emplear algoritmos de cifrado asimétricos para intercambiar claves de sesión simétricas. Posteriormente un algoritmo simétrico junto con la clave de sesión se encargará de proteger la comunicación. A este tipo de sistemas se les llama *sistemas criptográficos híbridos*.

La criptografía asimétrica ayuda con el problema de la distribución de claves y la simétrica proporciona la confidencialidad normalmente utilizando menos recursos computacionales.

Al término de la comunicación, es importante que las claves de sesión sean destruidas para prevenir que la confidencialidad sea dañada posteriormente por algún adversario que las pudiera recuperar.

2.1.4. Clases de Ataques

En general, los ataques pueden dividirse en dos tipos:

- *Ataques Pasivos.* El perpretador de un ataque viola la confidencialidad de la información al acceder a la misma, por ejemplo mientras pasa por la red, pero sin modificarla. Un problema importante con esta clase de ataques es la dificultad de detectarlos, lo cual es necesario para recuperarnos del incidente.

- *Ataques Activos.* El perpetrador tiene la capacidad de alterar la información, ya sea destruyéndola, modificándola o creándola. Debido a ello, no solamente se pierde la confidencialidad, sino también la integridad y la autenticación. Las operaciones que el criptoanalista puede realizar sobre los datos en cuestión incluyen a: la destrucción, interrupción, intercepción, modificación, fabricación y suplantación.

Existen diversas clases de ataques, los cuales se ocupan de diferentes aspectos del sistema. Entre ellos encontramos a los ataques a los protocolos, a los ataques a las implantaciones y a los ataques a los sistemas criptográficos. Veamos una clasificación de los ataques a los sistemas criptográficos de acuerdo al tipo de interacción o manipulación que puede efectuar un adversario [37]:

- *Ataque de solo texto cifrado.* El criptoanalista tiene acceso a un número de criptogramas, pero desconoce los mensajes originales, los cuales desea recuperar o, de ser posible, incluso recuperar la clave.

- *Ataque de texto en claro conocido.* Además de tener criptogramas, el criptoanalista conoce el texto en claro original correspondiente.

- *Ataque de texto en claro escogido.* Significa que el criptoanalista tiene la posibilidad de cifrar el texto en claro de su elección y escoge los textos en claro de los cuales puede obtener mayor información sobre la clave. Un caso más específico de este ataque es el *ataque adaptativo de texto en claro escogido*, en el que el adversario puede especificar el texto en claro a cifrar dependiendo de los criptogramas que ha obtenido anteriormente.

- *Ataque de texto cifrado escogido.* El criptoanalista puede obtener el producto de descifrar el criptograma que desee, pero no tiene acceso a la clave con que se descifra. Cuando el rival puede escoger el texto cifrado con base en el texto en claro recuperado con anterioridad, este ataque recibe el nombre de *ataque adaptativo de texto cifrado escogido*.

Esta no es una lista exhaustiva, es posible encontrar otros tipos de ataques, como por ejemplo: el *ataque de claves relacionadas*, que se aplica al modo de operación *RMAC*.

Los requerimientos para que un cierto ataque sea efectivo pueden especificarse de acuerdo a la complejidad en varios ámbitos:

- *Complejidad de Datos.* Indica cuál es la cantidad esperada de datos necesarios para realizar el ataque. Por ejemplo el número de criptogramas o de pares de criptogramas y textos en claro.

- *Complejidad de Procesamiento.* Se refiere al tiempo o número de operaciones a realizar, las cuales pueden ser elementales o complejas, como número de bloques a cifrar.

- *Complejidad de Almacenamiento.* Es la cantidad de memoria necesaria para montar el ataque.

Debido a que el poder computacional disponible se incrementa constantemente, es necesario que los algoritmos criptográficos que utilicemos estén diseñados para resistir no solo ataques con los recursos del presente, sino que deben poder resistir ataques con los recursos que encontraremos en algunos años. También debemos tomar en cuenta que es posible que se mejoren los ataques actuales o que surjan nuevos con complejidades inferiores.

En el caso del algoritmo *Rijndael* se conocen varios ataques que funcionan contra versiones mermadas del mismo. Con base en estos ataques se ha determinado el número de rondas que ofrece un balance óptimo entre desempeño y seguridad.

A medida que más texto en claro es cifrado, la mayoría de los sistemas criptográficos dejan cierta cantidad de información del texto en claro y la clave en los criptogramas, esto es, algunas claves y textos en claro comienzan a ser más probables que otros. Teóricamente, hay una manera de determinar la cantidad de criptogramas que es necesario obtener para recuperar la clave, a esta cantidad se le denomina *distancia de unicidad* y se expresa en número de caracteres o número de bloques de texto cifrado.

Examinemos con un poco más de detalle esta idea. La equivocación de clave $H_c(k)$ es la cantidad de incertidumbre que queda en la clave k dado que se obtuvo el mensaje cifrado c, o más formalmente la entropía condicional de k dado c:

$$H_c(k) = \sum_c p(c) \sum_k p(k|c) log_2(1/p(k|c)) \tag{2.3}$$

A medida de que la longitud del criptograma es más grande, tenemos que $H_c(k)$ disminuirá hasta llegar a un punto en que no tendremos incertidumbre sobre la clave.

En algunas circunstancias no sucede esto, tal es el caso del *one time pad*, en el que $H_c(k) = H(k)$, esto es, poseer un texto cifrado no modifica en nada el conocimiento *a priori* que tenemos sobre la clave.

La distancia de unicidad no solamente depende de las características inherentes del método de cifrado utilizado, sino también depende de las propiedades del texto en claro. La *tasa de un lenguaje* es $r = H(m)/N$, donde N es la longitud del mensaje (en número de caracteres) y m es la cantidad de bits de información promedio por carácter. En este caso se supone que el mensaje es una cadena sobre un alfabeto formado por L letras, no necesariamente correspondientes a letras de ningún idioma, sino en su forma más general.

La *tasa absoluta de un lenguaje* $R = log_2(L)$ equivale a la cantidad máxima de información que puede transmitirse con un carácter. Debido a las características del mensaje en claro, tenemos que $r \leq R$.

La *redundancia* de un lenguaje $D = R - r$ es la cantidad de información promedio que se desperdicia por cada carácter empleado del lenguaje respecto a la información total que podría mandarse si se hiciera un uso arbitrario de los caracteres (fuera del lenguaje).

De acuerdo a [6], empíricamente con cifradores clásicos se ha determinado que para un mismo lenguaje del texto en claro, la distancia de unicidad es proporcional a $log_2(Z)$ donde Z es la *complejidad combinatoria* de la clase de métodos de cifrado empleados.

Para el idioma Inglés se tiene que: $U \approx log_2(Z)/3.5$

Dicha complejidad combinatoria corresponde a *grosso modo* con la cantidad de transformaciones distintas que puede sufrir el texto en claro, así por ejemplo para el cifrado *Cesar* generalizado $Z = L$ (existen L posibles desplazamientos dentro del alfabeto, aunque uno de ellos produce la función identidad), para la sustitución monoalfabética simple arbitraria $Z = L!$, para una polialfabética $Z = (L!)^d$ donde d es la cantidad de alfabetos distintos independientes y para una transposición donde se reordenan n elementos $Z = n!$.

Es preciso remarcar que la distancia de unicidad depende tanto del tipo de transformaciones que se efectúan como de las características del texto en claro. Por ello es preferible disminuir la redundancia del texto en claro antes de cifrarlo, por ejemplo mediante técnicas de compresión de datos.

Por otro lado, una distancia de unicidad pequeña no significa que el algoritmo de cifrado correspondiente sea débil, ya que para recuperar la clave puede ser necesario efectuar una búsqueda por fuerza bruta, lo cual normalmente no es factible en términos de cómputo.

Más adelante se retomará la distancia de unicidad, específicamente para cifradores de bloques, y se relacionará directamente con la redundancia del lenguaje de entrada.

2.2. Cifradores de Bloques

Un algoritmo de cifrado de bloques opera en grupos de bits o bytes de longitud fija del texto en claro para convertirlos en un bloque de criptograma o viceversa.

La importancia fundamental de los cifradores de bloques radica en que son un componente fundamental para construir primitivas o protocolos criptográficos más complejos tales como *funciones hash*, *MACs*, *protocolos de reto–respuesta*, *generadores de números o secuencias pseudoaleatorias*, etc. Debido a esto y a que el algoritmo principal al que está enfocada esta tesis es un cifrador de bloques, es necesario explicar con mayor detalle el significado del término *cifrador de bloques*.

Formalmente, de acuerdo a [9], un cifrador de bloques es un sistema criptográfico en el cual el espacio de texto en claro y el espacio de texto cifrado son el conjunto Σ^n de palabras de longitud n sobre el alfabeto Σ, donde a n se le denomina *tamaño de bloque*.

Por otra parte, [30] define a un cifrador de bloques de n-bits como una función $E : V_n \times K \rightarrow V_n$ tal

que para cada clave $k \in \mathcal{K}$, $E(m,k)$ es un mapeo invertible (función de cifrado para k) de V_n a V_n, que también expresaremos como $E_k(m)$. El mapeo inverso es la función de descifrado $D_k(c)$. La notación fue ajustada para ser congruente con este documento.

Hay que resaltar que el tipo de transformación que realiza un cifrador de bloques es una permutación (un mapeo biyectivo de un dominio al mismo dominio) por lo que después de cifrar reiteradamente un cierto mensaje, siempre obtendremos, tarde o temprano, el mensaje original. Esto es de consideración especial cuando se emplean para generar secuencias pseudoaleatorias ya que el periodo en el que comienzan a repetirse los elementos de la secuencia puede disminuir inadvertidamente.

El cifrador de bloques está parametrizado por una clave k que pertenece al espacio de claves y que define la transformación o mapeo que sufrirá el texto en claro, o bien el criptograma.

Existe una gran cantidad de algoritmos de cifrado de bloques, es necesario conocer sus características para poder decidir cual es más adecuado para una aplicación dada. Para ello debemos saber cómo será utilizado el cifrador y estimar los requerimientos de velocidad (bits o bloques por segundo), de tamaños de clave, en que arquitectura será implantado el algoritmo (ya sea en *software* o *hardware*). También debemos conocer de que forma debe reaccionar el sistema ante errores de comunicación.

La manera más simple de utilizar un cifrador de bloques es partiendo el mensaje original en bloques de tamaño n. Cada uno de estos bloques es cifrado independientemente con la misma clave k y se transmiten los criptogramas resultantes. Los criptogramas son descifrados individualmente. El esquema planteado presenta muchos problemas, entre ellos que es susceptible a ataques por repetición de bloques, además que un mismo texto en claro siempre es cifrado de una cierta forma.

Para solucionar estos problemas y dar un mayor rango de posibilidades a las aplicaciones de los cifradores de bloques han surgido los modos de operación, algunos de los cuales se describen a continuación.

2.2.1. Modos de Operación

Un modo de operación, o modo, es un algoritmo que emplea a un cifrador de bloques de clave simétrica para proporcionar un servicio de información tal como la confidencialidad o la autenticación [34]. Los modos de operación son la forma en que se utiliza un cifrador de bloques, junto con algunos elementos simples como registros y operaciones elementales, para obtener un algoritmo con el fin de proteger la confidencialidad o integridad de la información.

Un cifrador de bloques estrictamente sólo puede operar en un bloque de tamaño n, pero para aplicaciones reales es necesario definir de que forma se tratarán datos de longitud mayor o menor. El procedimiento mediante el cual se ajusta la información para que ocupe una longitud que sea múltiplo del tamaño del bloque n se denomina *padding*.

Existen diversos métodos para realizar el *padding*. Es posible añadir solamente *ceros* al final del mensaje hasta completar un bloque, aunque esto puede no ser útil ya que de esta forma al momento de descifrar la información no se sabe exáctamente en donde termina el mensaje original. Otra forma de hacer el *padding* consiste en añadir al final del mensaje la cadena: 10^*que consiste en un *uno* seguido de tantos ceros como sea necesario hasta que el mensaje tenga longitud múltiplo de n. Debido a que la longitud del mensaje original casi siempre está dada en bytes, el final del mensaje con el *padding* es entonces 0x80 seguido de 0x00 hasta completar el bloque.

La principal desventaja de este método es que siempre se añade algún bit de *padding*, y en el caso de que el mensaje original ya tenga una longitud múltiplo de n será necesario cifrar un bloque extra, que tendrá el bit inicial en 1 seguido de $n-1$ *ceros*.

Veamos a continuación una descripción de los modos de operación más comunes:

- *Libro de Códigos Electrónico (ECB, Electronic Codebook).*
  Es la manera más sencilla de utilizar un cifrador de bloques. El mensaje original m es dividido en fragmentos m_i de n bits de longitud, los cuales son cifrados individualmente con una misma clave k para obtener los textos cifrados c_i. Esto puede expresarse como:

$$m = m_1|m_2|m_3|... \tag{2.4}$$

$$c_i = E_k(m_i) \tag{2.5}$$

$$c = c_1|c_2|c_3|... \tag{2.6}$$

  donde $|$ es la operación de concatenación de bloques. El proceso de descifrado es muy similar, para obtener cada fragmento del texto en claro original, basta con decodificar cada c_i individualmente mediante la clave k.

$$m_i = D_k(c_i) \tag{2.7}$$

  La principal característica de este modo de operación es que un mismo bloque de texto en claro es cifrado siempre de la misma forma bajo una misma clave, sin importar qué bloques hayan sido cifrados anteriormente.

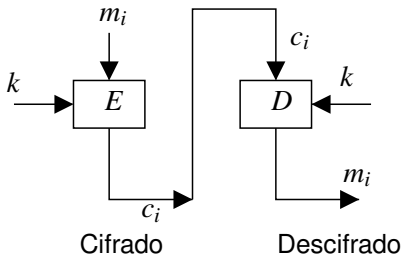

Figura 2.1: Diagrama del modo *ECB*.

Esto ofrece ventajas, por ejemplo, si se tiene un archivo cifrado, es posible descifrarlo parcialmente desde cualquier localidad, es decir, si solamente nos interesara el final del archivo no necesitamos descifrarlo desde el inicio. Por otro lado, la propagación de errores es mínima ya que un bit en error daña solamente el bloque al cual pertenece, aunque en el caso mucho menos frecuente en el que hay pérdida de la sincronización (por ejemplo si se inserta un byte en alguna parte de c), entonces este error será catastrófico y al momento del descifrado se tendrán indefinidamente bloques en error.

El modo *ECB* equivale a que tuviéramos un libro de códigos muy grande (con 2^n entradas) donde a cada entrada que equivale a un texto en claro posible le corresponde un criptograma en particular. Para cifrar simplemente necesitamos hacer una búsqueda en este libro de códigos. El proceso de descifrar es muy similar pero necesitamos otro libro de códigos con la transformación inversa. En realidad la idea del libro de códigos es solamente teórica ya que el tamaño de este libro sería excesivo.

Sin embargo, hay ciertos problemas que se basan en este concepto teórico. Un criptoanalista podría comenzar a acumular pares de texto en claro con su criptograma correspondiente. Siempre que se cifre uno de los textos en claro que ya conoce, sabrá de cual se trata. Además puede montar un ataque basado en la repetición de bloques. Es posible crear, reordenar o insertar mensajes cifrados sin que el usuario autorizado tenga forma de detectarlo al momento de descifrar el mensaje.

- *Encadenamiento de Bloque Cifrado (CBC o Cipher Block Chaining)*
  El modo *CBC* resuelve algunos de los problemas que presenta el modo *ECB*. Dos bloques iguales pueden ser cifrados de forma distinta, de acuerdo a los bloques que hayan sido cifrados previamente y a un bloque VI (*Vector de Iniciación*) de longitud n.

  El principio fundamental del algoritmo está en cifrar el texto en claro mezclado por medio de una or–exclusiva (XOR) con el último criptograma generado. La primera vez que se cifra un bloque, no se tiene un criptograma previo, pero en su lugar se utiliza el vector de iniciación. Podemos observar esquemáticamente el modo *CBC* en la figura 2.2.

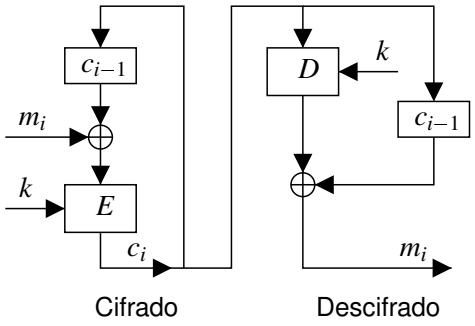

Figura 2.2: Diagrama del modo *CBC*.

Formalmente el cifrado equivale a:

$$c_0 = VI \tag{2.8}$$

$$c_i = E_k(m_i \oplus c_{i-1}); \quad i \geq 1 \tag{2.9}$$

Y el descifrado se expresa mediante:

$$c_0 = VI \tag{2.10}$$

$$m_i = D_k(c_i) \oplus c_{i-1}; \quad i \geq 1 \tag{2.11}$$

Hay que tomar en cuenta que si se utiliza el mismo vector de iniciación, y los mensajes comienzan con los mismos bloques, entonces tendremos el mismo problema que con *ECB* en cuanto a que los criptogramas de cada bloque serán idénticos. Esto sucederá hasta que se presente la primera diferencia en un bloque de texto en claro.

El hecho de que estrictamente el criptograma del bloque actual dependa de todos los criptogramas anteriores no significa que un error en un bloque se propague indefinidamente. Para descifrar un bloque solamente se necesita el criptograma actual y el anterior. Veamos cómo se comporta el descifrado si tenemos un criptograma $c_j^* = c_j \oplus e$ recibido con error e en el bloque j. Se supondrá que los demás bloques c_i^* se han recibido sin error.

Al momento de descifrar el bloque correspondiente a m_j y a m_{j+1}, mediante la ecuación 2.11 aplicada a los criptogramas $c^*{}_j$ y c_{j+1}^* obtenemos lo siguiente: $m_j^* = D_k(c_j^*) \oplus c_{j-1}^*$, y sustituyendo términos tenemos: $m_j^* = D_k(c_j \oplus e) \oplus c_{j-1}$. Debido al error e tendremos un bloque erróneo m_j^* totalmente inservible.

Ahora veamos que ocurre con m_{i+1}^*: $m_{j+1}^* = D_k(c_{j+1}^*) \oplus c_j^*$, sustituyendo obtenemos:
$m_{j+1}^* = D_k(c_{j+1}) \oplus c_j \oplus e$ o lo que es lo mismo: $m_{j+1}^* = m_{j+1} \oplus e$.
Esto quiere decir que al introducir un error e en el criptograma actual podemos producir un error

controlado para el bloque descifrado siguiente. Lo cual puede explotarse sobretodo cuando los mensajes transmitidos contienen cierta estructura, con lo que campos o registros específicos pueden ser modificados. Si bien no se viola la confidencialidad, la integridad se ve fuertemente afectada.

En cuanto al vector de iniciación, es conveniente cambiarlo para cada nuevo mensaje evitando su reutilización. Esta no es una tarea tan complicada debido a que el vector de iniciación no tiene que ser secreto.

- *Retroalimentación de Criptograma (CFB Cipher Feedback Mode)*

Hay ocasiones en que es importante cifrar y descifrar la información en bloques de tamaño mucho más pequeño que el del bloque del cifrador que es el núcleo del modo de operación. Esto puede ser útil, por ejemplo, en aplicaciones de transmisión de voz o multimedia en las que se desea transmitir la información tan pronto como está disponible para minimizar tiempos de retardo.

Por otro lado, por cuestiones de eficiencia, muchas veces es preferible implantar solamente el algoritmo de cifrado, lo cual lejos de significar que solamente podremos cifrar la información, quiere decir que no nos tenemos que preocupar por hacer también la implantación del algoritmo de descifrado. Esto se refleja en un menor uso de recursos ya que se reduce la circuitería o memoria de programa requerida para almacenar los algoritmos.

La principal característica del modo *CFB* es que se genera una secuencia t_i que se suma bit a bit mediante una or–exclusiva (XOR) con el mensaje m_i. El resultado de esta adición es el mensaje cifrado c_i. La generación de la secuencia aditiva se basa en el texto cifrado anterior y la función de cifrado.

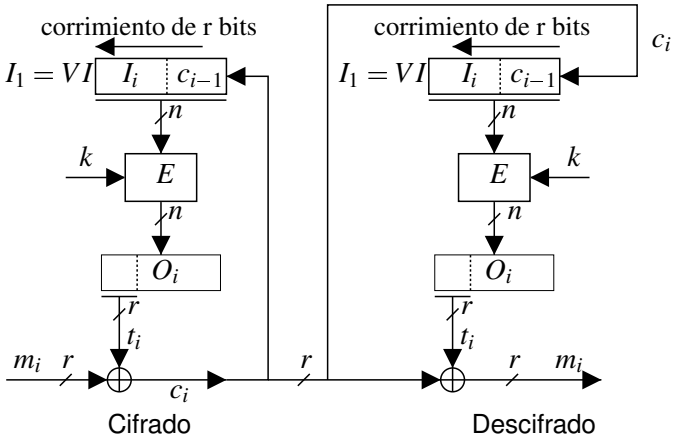

Figura 2.3: Diagrama del modo *CFB*.

La figura 2.3 ilustra el funcionamiento de este modo. Primero que nada debemos notar que el cifrador de bloques trabaja con bloques de longitud n, mientras que los criptogramas c_i son de longitud r. No está prohibido que $r = n$, en cuyo caso no se requiere hacer corrimientos.

La salida del cifrador de bloques es $O_i = E_k(I_i)$. El cuadro donde se encuentra O_i no es un registro, por lo que no introduce ningún retardo. Lo incluímos en el diagrama para mostrar que de la señal O_i se toman los r bits más significativos los cuales son la secuencia aditiva que será sumada módulo 2 (XOR) al mensaje original.

Hay que remarcar que tan pronto como un nuevo mensaje m_i o un criptograma c_i está disponible, se puede cifrar o descifrar inmediatamente con tan solo efectuar una suma módulo 2, ya que las secuencias aditivas para realizarlo ya fueron calculadas con base en m_{i-1} o c_{i-1}. Gracias a esto se minimiza el tiempo de retardo.

Veamos las ecuaciones que rigen el modo *CFB*:

$$I_1 = VI \tag{2.12}$$

$$I_{i+1} = 2^r \cdot I_i + c_j \bmod 2^n; \quad i \geq 1 \tag{2.13}$$

$$O_i = E_k(I_i) \tag{2.14}$$

$$t_i = \textbf{r-bits-más-significativos}(O_i) \tag{2.15}$$

Para el caso del cifrado: $c_i = m_i \oplus t_i$, y para el caso del descifrado: $m_i = c_i \oplus t_i$.

Es importante que el vector de iniciación *IV* sea distinto cada vez, de lo contrario se podría recuperar el texto en claro ya que t_1 sería constante y conociendo una pareja (m_1, c_1) se puede recuperar cualquier texto en claro m_1' a partir de su respectivo c_1'. Este ataque puede extenderse a cualquier m_i' si tenemos la posibilidad de un ataque de texto en claro escogido.

En cuanto a la propagación de errores, un bit en error en el criptograma afecta un bit del texto en claro recuperado del bloque actual, y justamente ocupa la misma posición dentro del bloque de r bits. Posteriormente se tendrán bloques de texto en claro completamente erróneos hasta que el bit incorrecto salga del registro de corrimiento.

Si el criptoanalista conoce el contenido del último bloque de texto en claro de r bits, entonces puede modificar arbitrariamente el valor al que será descifrado.

- *Retroalimentación de la Salida (OFB Output Feedback Mode)*

  El cifrado mediante el modo *OFB* del mensaje en claro m_i no depende de los mensajes anteriores sino del índice i, o más específicamente de un cierto estado que cambia cada vez que se incrementa i.

  Puede entenderse más fácilmente este modo si consideramos que se trata simplemente de la generación de una secuencia pseudoaleatoria que depende del vector de inicialización VI. La cual se suma módulo 2 con el mensaje original.

  Este modo de operación es muy similar al *CFB*, pero es más sencillo realizar manipulaciones del texto en claro recuperado ya que los criptogramas no dependen de los mensajes anteriores. Invertir el valor de bits individuales del texto en claro recuperado es trivial, basta con invertir los bits correspondientes en el texto cifrado.

  Veamos también formalmente las ecuaciones que rigen este modo de operación:

$$I_1 = VI \tag{2.16}$$

$$O_i = E_k(I_i) \tag{2.17}$$

$$I_{i+1} = O_i \tag{2.18}$$

$$t_i = \textbf{r-bits-más-significativos}(O_i) \tag{2.19}$$

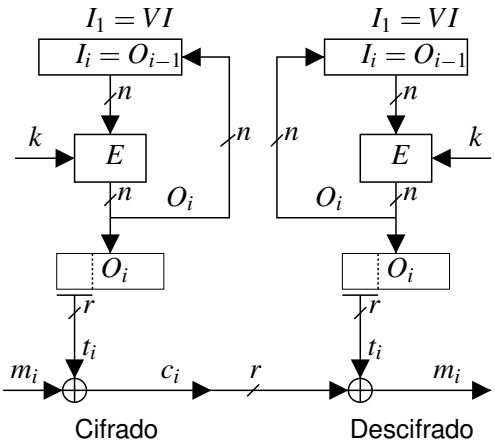

Figura 2.4: Diagrama del modo *OFB*.

Para el caso del cifrado: $c_i = m_i \oplus t_i$, y para el cado del descifrado: $m_i = c_i \oplus t_i$.

No hay ninguna restricción que evite que $r = n$. También en este modo es importante que el vector de iniciación nunca se reutilice, de lo contrario un criptoanalista podría romper la confidencialidad si logra un ataque de texto en claro conocido de la siguiente forma: Si tiene una pareja de cualquier texto en claro y su correspondiente texto cifrado (m_i', c_i'), entonces puede calcular $t_i = m_i' \oplus c_i'$ y puede descifrar un criptograma c_i ya que $m_i = c_i \oplus t_i$.

La propagación de errores es mínima ya que un bit en error en el texto cifrado solamente afecta a un bit del texto en claro recuperado.

- *Modo Contador (CTR Counter Mode)*
  Vamos a discutir el modo contador de acuerdo a [28], aunque con varias simplificaciones y cambiando de notación. El modo *CTR* es similar al modo *OFB* en cuanto a que se genera una secuencia pseudoaleatoria t formada por bloques t_i independiente de los textos en claro previos. A diferencia del modo *OFB*, la secuencia pseudoaleatoria no depende de valores previos de la misma. Dicha secuencia se suma módulo 2 con el mensaje y se produce el texto cifrado. El proceso de descifrado consiste en volver a sumar la misma secuencia pseudoaleatoria al criptograma.

  La principal ventaja del modo *CTR* es que permite una gran eficiencia sin debilitar la seguridad. Además puede implantarse de forma muy eficiente tanto en *hardware* como en *software*, y debido a la independencia de valores previos pueden computarse los bloques t_i de la secuencia antes de tener el texto en claro o el criptograma disponible. Otra ventaja de este modo es que se puede descifrar un bloque arbitrario sin tener que descifrar previamente un bloque anterior o posterior.

  El punto principal de este modo de operación es que se tiene un contador (el cual no debe reutilizarse) y al cifrarlo se obtiene un bloque que compone la secuencia pseudoaleatoria. Veamos las ecuaciones que formalizan el funcionamiento de este modo:

$$\mathrm{ctr}_{i+1} = \textbf{siguiente}(\mathrm{ctr}_i) \tag{2.20}$$

$$t_i = E_k(\mathrm{ctr}_i) \tag{2.21}$$

  Cuando estamos cifrando calculamos:

$$c_i = m_i \oplus t_i \tag{2.22}$$

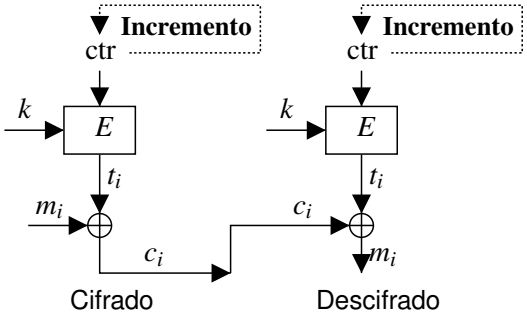

Figura 2.5: Diagrama del modo Contador.

y cuando desciframos obtenemos m_i mediante:

$$m_i = c_i \oplus t_i \tag{2.23}$$

La secuencia pseudoaleatoria descrita es simplemente $t = t_1|t_2|t_3|\cdots$; donde $|$ es la operación de concatenación.

La función **siguiente** consiste en un incremento, normalmente de 1 aunque puede ser diferente, tal como la salida de un generador de secuencias pseudoaleatorias, sin importar que dicho generador sea seguro o no.

En las ecuaciones no se especificó el valor de ctr_1, esto se debe a que hay formas distintas de generarlo, una de ellas consiste en que ctr_1 se deriva de un entero o *nonce* el cual es de 64 bits y define la parte alta de ctr_1 el cual es de 128 bits. La parte baja de ctr_1 se llena con ceros. Es importante recalcar que se debe evitar la reutilización del contador, de lo contrario se compromete la seguridad de este esquema. Para una descripción más completa de las formas de especificar el *nonce* se recomienda consultar [28].

En cuanto al efecto de los errores, un bit en error en el criptograma corresponde a un bit en error en el texto en claro recuperado. De igual forma a como ocurre en el modo *OFB*, esto puede ocasionar manipulación de los mensajes por parte de un criptoanalista si éste conoce su estructura. Hay que señalar que la mayoría de los modos de operación por si solos no aseguran la integridad, para ello deben conjuntarse con otros mecanismos.

Hasta ahora se han examinado cinco modos de operación los cuales son algunos de los más populares y cuyo propósito es el cifrar la información, sin embargo esta no es una lista exhaustiva, existe una gran cantidad de modos con usos diferentes. Tal es el caso de los modos que proporcionan la autenticación de mensajes, esto es, a partir del mensaje y una clave secreta se obtiene un bloque de información llamado *Código de Autenticación de Mensaje* o MAC. Normalmente el MAC es anexado con la información original y permite detectar las modificaciones (accidentales o intencionales) del mensaje original. Si deseamos comprobar la integridad de un mensaje recibido, tenemos que calcular el MAC del mensaje y compararlo con el MAC recibido. Si no coinciden es seguro que el mensaje (o el MAC) fue modificado. El MAC más común es el *CBC–MAC* y se basa en la misma estructura del modo *CBC*, la diferencia es que la única salida que produce el algoritmo es el último bloque de texto cifrado. Algunas variaciones del *CBC–MAC* aplican algunas otras transformaciones a este bloque. Por ejemplo, *RMAC* es un modo de operación, el cual es una tipo de MAC aleatorio y que por excelencia funciona utilizando al *AES* como cifrador de bloques base (aunque podría utilizar a otro cifrador).

Otros modos de operación permiten la creación de funciones huella. La autenticación de mensajes puede lograrse si se aplica una función huella a un mensaje y posteriormente dicha huella es autenticada. Como se verá más adelante hay distintos tipos de funciones huella, y debemos conocer sus características para darles el uso apropiado.

2.2.2. Algunos Tipos de Cifradores de Bloques

Ahora presentaremos una breve descripción de cuatro tipos de cifradores de bloques para ir conociendo algunas de las características del *AES*.

- *Cifradores de Bloques Iterativos [13]*
  Esta clase de cifradores consiste en la composición de un cierto número de rondas, las cuales son permutaciones $\rho^{(i)}[k^{(i)}]$ que dependen de la clave $k^{(i)}$ donde i indica el número de ronda, $\rho^{(i)}$ es la transformación de la ronda i y $k^{(i)}$ es la clave de ronda i.

  Las claves de ronda son derivadas de la clave del algoritmo, k, mediante el procedimiento denominado *expansión de clave*.

  Nótese que hay total libertad en la elección de cada $\rho^{(i)}$ para todas las rondas, excepto que estas transformaciones deben ser biyecciones.

  El cifrado completo es el resultado de aplicar cada una de las rondas. Si tenemos r rondas, entonces la transformación total B podemos expresarla como:

  $$B[k] = \rho^{(r)}[k^{(r)}] \circ \rho^{(r-1)}[k^{(r-1)}] \circ \cdots \circ \rho^{(2)}[k^{(2)}] \circ \rho^{(1)}[k^{(1)}] \tag{2.24}$$

  Donde $\circ$ es la operación de composición. Tres tipos especiales de cifradores iterativos que imponen ciertas restricciones a las transformaciones $\rho^{(i)}$ son los siguientes:

- *Cifradores de Bloque Iterados*
  Son aquellos cifradores de bloque iterativos en los que las funciones de ronda son idénticas, y solamente se habla de una función de ronda, la cual, por supuesto, es una biyección parametrizada para cada clave de ronda $k^{(i)}$.

- *Cifradores de Bloque con Clave Alternante*[2]
  La clave es aplicada de una forma especialmente simple. La alternancia reside en que se intercala una adición de la clave y una transformación de ronda independiente de la misma.

  La primera adición de la clave de ronda se hace antes de la ronda inicial y la última al término de la ronda final.

  Esta estructura presenta ventajas tales como el ahorro de memoria de programa para almacenar la transformación de ronda (puesto que todas las transformaciones de rondas son iguales, solamente se tiene que almacenar una de ellas). Desde el punto de vista de la seguridad del algoritmo, se facilita el análisis, sobretodo el que se basa en matrices de correlación ya que se simplifican las operaciones en este dominio.

- *Cifradores de Bloque con Clave Iterada*[3]
  Los cifradores de bloque con clave iterada son un caso particular de los cifradores de bloque con

[2] *Key-Alternating Block Ciphers*.
[3] *Key-Iterated Block Ciphers*.

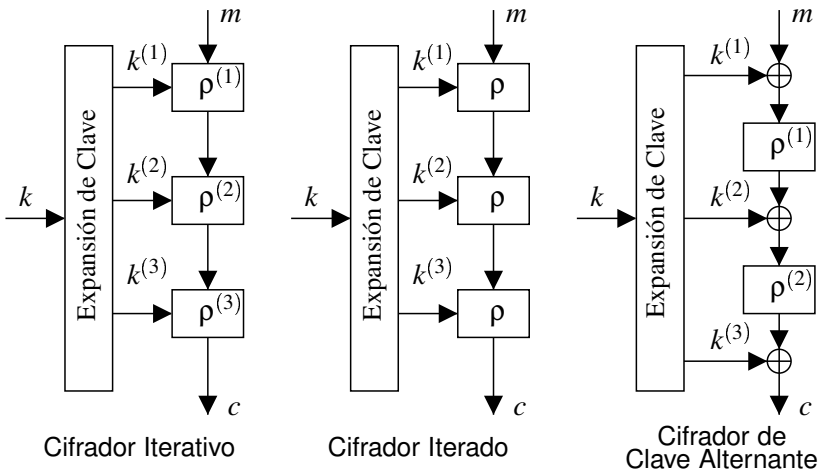

Figura 2.6: Tres tipos de cifradores de bloques.

clave alternante en los que todas las transformaciones de ronda son idénticas e iguales a ρ. Como se verá más adelante, el *AES* pertenece a esta clase de cifradores.

2.2.3. Distancia de Unicidad para Cifradores de Bloques

La distancia de unicidad fue presentada en una sección anterior (página 35), y se define como la cantidad mínima de criptograma que se necesita para determinar la clave empleada con recursos computacionales ilimitados. Ahora veremos como se aplica este concepto a cifradores de bloques, para lo cual analizaremos el caso de un cifrador aleatorio.

En el siguiente desarrollo consideraremos bloques de N caracteres. Se tiene una tasa absoluta del lenguaje R, que es la cantidad máxima de bits de datos que el lenguaje utilizado permite transmitir por cada carácter y una tasa del lenguaje de r, que es la cantidad de bits de información promedio por cada carácter. Si estamos utilizando bytes para definir nuestro alfabeto entonces $R = 8$ bits / carácter y estimaciones para el idioma Inglés indican que la cantidad de información promedio es de $r = 1.5$ bits / carácter. La redundancia $D = R - r$ es entonces de 6.5 bits / carácter. El alfabeto se puede limitar solamente a los caracteres empleados, en ese caso hay 26 opciones para cada carácter y es posible especificar un carácter en particular con $R = log_2(26) \approx 4.7$ bits / carácter, por lo que se tendría una redundancia de 3.2 bits / carácter.

Un cifrador aleatorio se caracteriza porque para cada clave k, el descifrado de $D_k(c)$ es una variable uniformemente distribuida sobre todos los $2^{R \cdot N}$ mensajes posibles. Este mapeo aleatorio no es arbitrario ya que para cada k se debe tener una permutación para conseguir el descifrado único.

La distancia de unicidad para este tipo de cifrador es $U = H(k)/D$. Donde $H(k)$ es la entropía del espacio de claves. Esta variable está muy relacionada con lo que anteriormente denominamos la *complejidad combinatoria* (Z) de la clase de métodos de cifrado empleados.

Si cada clave define un método de cifrado distinto, entonces intuitivamente $H(k) = log_2(Z)$, esto concuerda con los datos empíricos presentados en la página 36.

Ahora pongamos el caso del *AES*. Por ahora solamente es necesario considerar al *AES* como algún cifrador de bloques, el cual para este ejemplo trabaja con bloques y claves de 128 bits (16 bytes) aunque soporta otros tamaños de claves. Los mensajes a cifrar estarán en el idioma Inglés codificados en *ASCII*[4]

[4] *American Standard Code for Information Interchange*.

(utilizando los 26 símbolos básicos) y se utilizará un byte por letra. Tenemos que $R = 8$ bits / carácter, $r = 1.5$ bits / carácter y $H(k) = 128$ bits. Entonces $D = 8 - 1.5 = 6.5$ bits / carácter y $U = \frac{H(k)}{D} \approx 19.69$ caracteres, como se tienen 16 caracteres por bloque, esto quiere decir que se necesitan $U = 19.69/16 = 1.23$ bloques aproximadamente para recuperar inequívocamente la clave k utilizada. Suponiendo que codificamos los caracteres con 5 bits cada uno, entonces tendríamos $D = 5 - 1.5 = 3.5$ bits / carácter y $U \approx 36.57$ caracteres ≈ 2.286 bloques. Con esto debe quedar claro que teóricamente recuperar la clave requiere de un número pequeño de bloques cifrados, aunque para lograrlo se requeriría una cantidad impresionante de recursos computacionales.

2.2.4. Composición de Cifradores

Si tenemos algunos cifradores de bloques y queremos incrementar la seguridad que proporcionan, una solución podría ser cifrar dos o más veces el mensaje en claro utilizando primero uno de los cifradores y luego otro. En general el método de cifrado de la composición en su conjunto no es equivalente a ningún método de cifrado de los algoritmos utilizados sino que es uno nuevo. Por ejemplo, si utilizamos el *AES* dos veces con claves distintas k_1 y k_2, obtendremos un cifrado que no corresponde a un cifrado con el *AES* para ninguna clave k. Un contraejemplo es el caso de la sustitución. Aplicar dos veces la sustitución equivale a aplicarla una sola vez utilizando la clave adecuada.

Podría darse el caso de que en realidad se reduzca la seguridad debido a propiedades de los cifradores, aunque la probabilidad de que esto suceda si se utilizan claves independientes es pequeña. Veamos cuáles son las formas de implantar la composición de los cifradores:

- *Cifradores en Cascada.* Es la forma más general de composición, consiste en utilizar dos o más cifradores en serie con claves independientes. Hay dos tipos:
  - *Caso general.* Los cifradores pueden ser distintos.

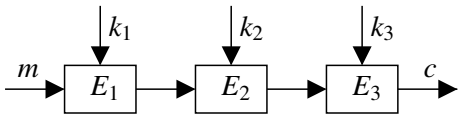

Figura 2.7: Cifrado mediante un esquema de cascada en general.

  - *Con cifradores idénticos.* Se utiliza el mismo cifrador base.

- *Cifrado Múltiple.* Se utiliza el mismo algoritmo E para cifrar el mismo mensaje, también está permitido utilizar el algoritmo inverso E^{-1}. Es similar al cifrador en cascada con cifradores idénticos pero las claves no tienen que ser independientes.

Ahora describiremos algunas formas de cifrado múltiple, bajo el supuesto de que la composición de transformaciones nos arrojará un método de cifrado fuera del esquema de cifrado original.

- *Cifrado doble*
  Formalmente se define como: $E(m) = E_{k_2}(E_{k_1}(m))$ Podría pensarse que al duplicarse el tamaño real del material de clave de n a $2n$, estaremos incrementando significativamente la seguridad. Recuperar la clave de un cifrador con longitud de clave n mediante la fuerza bruta requiere de un máximo de 2^n operaciones. Al aplicar el cifrado doble no incrementamos el trabajo necesario por

un ataque de fuerza bruta a las 2^{2n} sino tan solo a 2^{n+1} utilizando una memoria de 2^n. Al final de esta subsección hablaremos del ataque de *encuentro en la mitad*, que es el ataque que hace esto posible.

- *Cifrado Triple*
  En general consiste en la aplicación de tres operaciones de cifrado o descifrado con tres claves independientes. La manera más común de utilizar el cifrado triple es mediante el modo *EDE* (cifrar, descifrar, cifrar) con 2 claves distintas, por ejemplo, el cifrado triple con dos claves se define como: $E(m) = E_{k_1}(D_{k_2}(E_{k_1}(m)))$. [37] muestra que el orden de aplicación de las claves no es arbitrario y se puede caer en un ataque de encuentro en la mitad simple: $E(m) = E_{k_2}(E_{k_1}(E_{k_1}(m)))$ puede ser roto con 2^{n+2} operaciones.

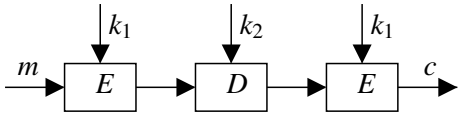

Figura 2.8: Cifrado triple con dos claves: Modo *EDE*.

Es común que los ataques puedan intercambiar un menor requerimiento de cálculo por un mayor uso de memoria y a la inversa. Teóricamente romper el cifrado triple *EDE* con dos claves requiere de 2^{n-1} cifrados utilizando una memoria de 2^n bloques. Cuando se utilizan tres claves independientes, romper el algoritmo requiere de 2^{2n} cifrados utilizando una memoria de 2^n bloques.

Ataques de Encuentro en la Mitad

A continuación se describe un tipo de ataque de texto en claro conocido al cual son especialmente susceptibles los cifradores en cascada, y en particular desacreditan al cifrado doble.

Cuando tenemos una transformación que se puede expresar como la aplicación de dos transformaciones consecutivas parametrizadas por claves distintas, entonces es factible recuperar dichas claves sin realizar todo el trabajo computacional de un ataque común por fuerza bruta tomando la transformación completa.

Para ejemplificar este ataque utilizaremos el cifrado doble, en el cual el texto en claro es cifrado dos veces, primero con una clave k_1 y luego con la clave k_2: $c = E_{k_2}(E_{k_1}(m))$, ambas claves de longitud n. Dados unos cuantos pares de textos en claro m_i con su criptograma c_i, la fuerza bruta probaría todas las combinaciones de k_1 y k_2 hasta que una de estas combinaciones cifre correctamente los textos en claro conocidos de acuerdo a los criptogramas previamente obtenidos. El trabajo para probar todas estas combinaciones es de 2^{2n} y con probabilidad 1 recuperaremos las claves.

El primer paso de un ataque de encuentro en la mitad es cifrar un mensaje m_1 con distintas claves $k_1 = i$. El resultado $s_i = E_i(m_1)$ junto con la clave i es almacenado y ordenado en una tabla. Esto tomará 2^n cifrados. El segundo paso consiste en descifrar el criptograma c_1 con cada una de las claves distintas $k_2 = j$, obteniendo un estado intermedio $s_j = D_j(c_1)$. Se busca en la tabla concordancias entre s_j y s_i, cuando se encuentra una tenemos una pareja candidata de claves (i, j) que pueden ser las k_1 y k_2 originales. Para descartar meras coincidencias se utiliza una segunda pareja de texto en claro conocido m_2 y su criptograma c_2, si al intentar cifrar m_2 con las claves i y j el criptograma coincide, es muy probable que tengamos las claves correctas, si no, podemos seguir con la siguiente clave j.

Un inconveniente de este ataque es la cantidad de memoria que utiliza el algoritmo, la cual es del orden de 2^n, pero en el peor de los casos se realizan 2^{n+1} cifrados, lo cual es muy pequeño comparado

con los 2^{2n} cifrados que requiere el ataque por búsqueda exhaustiva.

2.3. Funciones de Huella, o *Hash*

Una de las primitivas criptográficas más importantes son las funciones de huella. Muchos protocolos basan su seguridad en ciertas propiedades que exhiben este tipo de funciones. Sus aplicaciones más frecuentes incluyen a la protección de la integridad y a la autenticación del origen de los mensajes.

Normalmente los algoritmos que nos permiten generar una firma digital son pesados computacionalmente, pero en lugar de firmar el documento completo es posible firmar su huella, siempre que ésta sea generada correctamente.

En sistemas de almacenamiento, es difícil asegurar la integridad de archivos grandes, en lugar de ello es posible asegurar la integridad de la huella dado que es un bloque de información generalmente mucho más pequeño. Esta huella puede ser almacenada de forma segura en un dispositivo como una tarjeta inteligente.

Las funciones de huella no son utilizadas exclusivamente con fines criptográficos, sino en aplicaciones en general, especialmente en las que requieren búsquedas para facilitar la indexación, así que es necesario conocer sus características para utilizar el tipo de función huella más adecuada.

Una función huella h se define como [9] un mapeo $h : \sum^* \to \sum^n, \quad n \in N$ que mapea cadenas de longitud arbitraria $\sum^* = \bigcup_{i=0}^{\infty} \sum^i = \sum^0 \cup \sum^1 \cup \cdots$ sobre el alfabeto $\sum$ a cadenas $\sum^n$ de longitud fija n. Comúnmente se utiliza un alfabeto que representa un byte o el alfabeto binario $\{0,1\}$.

Relacionado con el concepto de funciones huella está el concepto de las funciones de compresión, las cuales de acuerdo a [9] son una función: $h : \sum^m \to \sum^n, \quad m,n \in N, \quad m > n$. Es decir, mapean una cadena de longitud m a una más pequeña de longitud n.

La primera restricción que se añade a las funciones huella (y a las de compresión) es que deben ser fáciles de calcular.

Sin embargo, una función huella general tal y como fue definida es de poco valor para la criptografía, para que nos sea de utilidad es necesario añadirle ciertas propiedades.

2.3.1. Funciones Unidireccionales (*One–Way Functions*)

Una función h es unidireccional si es difícil de encontrar el valor de algún elemento m tal que $h(m) = s$ para un s dado. A esta propiedad se le denomina *resistencia a preimágenes*. La nomenclatura estándar indica que para $s = h(m)$, s es llamada imagen y m es la preimagen.

En [29, p. 16] se proporciona una definición informal la cual en esencia dice que una función unidireccional debe ser calculable en tiempo polinomial y en promedio debe ser difícil de invertir, en el sentido de que la probabilidad de que cualquier función A, calculable en tiempo polinomial, invierta a h es pequeña.

Cuando encontramos una pareja de valores m, m', tales que $s = h(m) = h(m')$ se dice que hemos producido o encontrado una *colisión*.

En el supuesto de que un criptoanalista pueda conocer $s = h(m)$, si la función unidireccional tiene resistencia a preimágenes, entonces sería muy difícil en la práctica que éste pudiera crear una preimagen m' tal que $h(m') = s$, sin embargo es común que el criptoanalista tenga acceso también al mensaje original m, en cuyo caso existe la posibilidad de que el adversario sea capaz de generar un nuevo mensaje m' tal que $h(m') = s$ aún cuando h sea resistente a preimágenes.

Como solución al problema planteado surge la propiedad llamada *resistencia a segunda preimagen*, la cual indica que dado un mensaje m, es difícil encontrar una segunda preimagen $m' \neq m$ tal que $h(m') = s = h(m)$.

La resistencia a la segunda preimagen puede ser útil, por ejemplo, en el caso de que tengamos un archivo guardado en el disco duro y deseamos proteger su integridad. Para conseguirlo guardamos su imagen s en un medio seguro, tal como una tarjeta inteligente. Cuando deseemos comprobar la integridad del archivo lo único que tenemos que hacer es volver a calcular su imagen s' y la comparamos con la imagen s que teníamos almacenada previamente, si coinciden es bastante probable que el archivo no ha sufrido modificaciones.

Una tercera propiedad es la *resistencia a colisiones*, ésta consiste en que es difícil para un criptoanalista producir dos mensajes m y m' con la misma imagen.

Veamos un ejemplo en el que esta propiedad es deseable: si un tercero (malicioso) nos presenta un documento A para que lo firmemos, como un contrato, el tercero podría generar muchas versiones de A con alteraciones mínimas. Hacer esto es sencillo; una forma es insertando espacios extra en n posiciones distintas del mensaje. Variando cuales espacios ponemos y cuales no, generaríamos un total de 2^n documentos similares. El adversario también generaría distintas versiones de un documento perjudicial para nosotros B. Debido a la paradoja del cumpleaños (será descrita más adelante) no es tan difícil encontrar entre los mensajes producidos dos mensajes A' y B' con la misma imagen. El tercero nos presenta una versión A' del documento original el cual firmamos, con nuestra firma puede alegar que en realidad firmamos B' lo cual no era nuestro propósito.

Una función unidireccional que tiene las propiedades de ser resistente a preimágenes y a una segunda preimagen se le denomina *función unidireccional con resistencia débil a colisiones.* Si la función presenta la propiedad de ser resistente a colisiones, entonces se le agrega el calificativo de *con fuerte resistencia a colisiones*, o bien simplemente *resistente a colisiones*.

La Paradoja del Cumpleaños

La propiedad fundamental de las funciones de huella resistentes a colisiones es la dificultad de obtener dos preimágenes que compartan la misma imagen.

Una situación en la que las colisiones pueden representar un problema es cuando se firma la huella de un documento en lugar del documento en sí. La consecuencia de esto es que si encontramos una colisión en la función de huella podríamos señalar que el documento firmado fue otro. Es decir, si una entidad A firmó la huella $s = h(m_1)$ del mensaje m_1, entonces si podemos encontrar otro mensaje m_2 con la misma huella $h(m_2) = s$, entonces podríamos afirmar que A no está firmando m_1 sino m_2.

La paradoja del cumpleaños nos indica que no es tan difícil obtener dos mensajes con la misma huella como podría pensarse. Por poner un ejemplo, esta paradoja dice que si tomamos a 23 personas aleatoreamente, la probabilidad de que dos o más de ellas tengan el mismo cumpleaños es mayor a $1/2$. La paradoja se relaciona con las funciones de huella cuando consideramos que el cumpleaños es análogo al valor de la huella.

La aplicación de la paradoja da lugar a los llamados *ataques de cumpleaños*. Hay variaciones pero en general si hay $N = 2^m$ posibles valores huella diferentes, tenemos una buena probabilidad de obtener dos imágenes iguales después de calcular la imagen de $\sqrt{N} = 2^{m/2}$ mensajes distintos. El factor $1/2$ en el exponente disminuye considerablemente el esfuerzo requerido para encontrar una colisión, por lo que tamaños de clave más grandes que los tradicionales para el cifrado son requeridos.

Ahora podemos definir ciertos tipos de funciones huella que son más útiles para la criptografía que las funciones huella en general.

2.3.2. Funciones Huella (*Hash*) Útiles en la Criptografía

Los dos tipos principales de funciones de huella útiles para la criptografía son las:

- *Funciones huella unidireccionales (OWHF One Way Hash Functions).* Tienen las propiedades combinadas de una función unidireccional con resistencia débil a colisiones y las de una función huella: resistencia a preimagen, a segunda preimagen, facilidad de cálculo y salida de longitud fija para cualquier entrada.
- *Funciones huella con resistencia a colisiones (CRHF Collision Resistant Hash Functions).* Proporcionan resistencia a colisiones (y por lo tanto resistencia a segunda preimagen) y las características de una función huella.

De acuerdo a su modo de operar respecto a una clave puede clasificarse a las funciones huella en dos tipos:

- *Sin Clave.* Tal y como se han definido hasta ahora.
- *Con Clave.* La función huella depende tanto del mensaje original como de una clave.

Es posible construir una función huella resistente a colisiones a partir de una función de compresión resistente a colisiones. A continuación se describe una forma de hacerlo [9]. Primero que nada partimos de que tenemos una función de compresión resistente a colisiones $g : \{0,1\}^m \to \{0,1\}^n$. Para el desarrollo siguiente se requiere que $r > 1$ con $r = m - n$ y produciremos una función huella $h : \{0,1\}^* \to \{0,1\}^n$. Es necesario realizar un preprocesamiento a la entrada x de la cual obtendremos la huella:

1. Se añaden ceros al inicio de x para que su longitud sea múltiplo de r (*padding*)
2. Se añade un bloque de longitud r con ceros al final de x.
3. Obtenemos la longitud n del mensaje x original y se procesa en forma binaria dividiéndola en subcadenas de longitud $r-1$, a cada una de estas subcadenas se le agrega un uno al inicio (de forma que cada una tenga r bits) y todas juntas son colocadas al final de la cadena x.

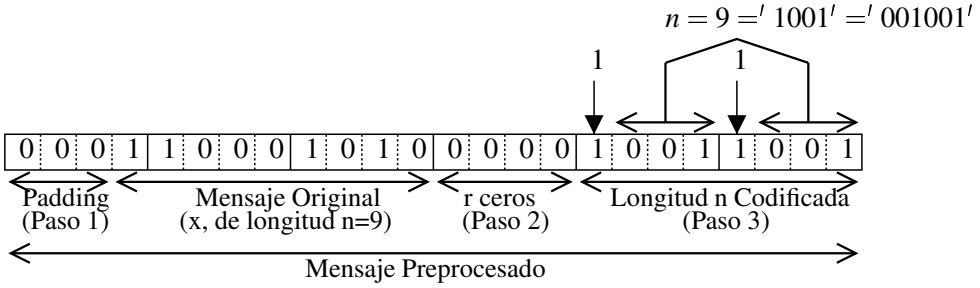

Figura 2.9: Preprocesamiento de la cadena x="10001010" para obtener su hash. $r = 4$.

Al final tendremos una nueva cadena que por claridad llamaremos y, la cual tiene longitud $t \cdot r$. La cadena y puede expresarse como: $y = y_1|y_2|\cdots|y_t$ con $|$ representando la operación de concatenación y a

cada mensaje distinto x corresponde exactamente una cadena distinta y.

Después aplicamos el siguiente proceso iterativo:

1. Asignamos $H_0 = 0^n$, es decir un bloque de longitud n de ceros.
2. Para $i = 1$ a $i = t$ calculamos:
   $H_i = g(H_{i-1}|y_i)$
3. Asignamos $h(x) = H_t$

Una variación de este método, con la misma idea fundamental aunque algo limitada, puede encontrarse en [37, p. 333].

Ahora veamos otra clasificación de las funciones huella de acuerdo a su uso:

- *Códigos de Detección de Manipulación (MDCs, Manipulation Detection Codes).*
  Estas funciones crean una imagen de un mensaje la cual permite identificar directamente modificaciones al mensaje original, esto es, verificar su integridad. Los MDC son funciones *hash* sin clave y hay de dos tipos: unidireccionales (OWHF) y resistentes a colisiones (CRHF).
- *Códigos de Autenticación de Mensajes (MACs, Message Authentication Codes).*
  Con los MACs se genera una huella de un mensaje utilizando una clave secreta. La huella resultante permite comprobar además de la integridad de los datos, su origen.

Los MDCs tienen la limitación importante de que por sí solos no nos aseguran quién originó los datos (autenticación), puesto que un tercero podía generar mensajes y su huella respectiva.

Una ejemplo de esto es la comunicación en las tarjetas inteligentes mediante los protocolos *T=0* y *T=1* (aunque no son funciones huella ni tienen la propiedad de unidireccionalidad) en donde los códigos empleados solamente permiten la detección de errores que destruirían la integridad, sin embargo la modificación de los datos transmitidos causada por un ataque activo pasaría inadvertida. El criptoanalista solamente tiene que asegurarse de calcular los valores nuevos de los códigos de detección de errores. Esto se debe a que dichos protocolos no fueron pensados para evitar amenazas maliciosas a la integridad.

Veamos los códigos de detección de errores que forman parte de los protocolos *T=0* y *T=1*.

El mecanismo más sencillo de detección de errores es un bit de paridad. El protocolo *T=0* especifica que cada byte transmitido está acompañado de un bit de paridad par. Desgraciadamente este esquema solamente detecta la presencia de un número impar de errores.

El protocolo *T=1* especifica dos mecanismos de detección de errores, el primero es muy similar al bit de paridad pero opera sobre bytes, es decir, se calcula la suma de comprobación de un paquete a partir de la suma módulo 2 de cada uno de los bytes que conforman el paquete. El resultado se anexa al final del bloque. A este esquema también se le conoce con un nombre más elegante: *comprobación longitudinal de redundancia*[5]. Una desventaja de esta suma de comprobación es que no es sensible al reordenamiento de los bytes ni a un error repetido ya que se cancelaría a sí mismo.

El segundo mecanismo del protocolo *T=1* es más sofisticado y puede implantarse en *hardware* mediante un registro de corrimiento con retroalimentación lineal. Es una *CRC*[6], o *Comprobación Cíclica*

[5] *Longitudinal Redundancy Check*.
[6] *Cyclic Redundancy Check*.

de Redundancia. Matemáticamente equivale a representar el mensaje como un polinomio (cada bit del mensaje es un coeficiente) y calcular su residuo al ser dividido por otro polinomio fijo $G(x)$. Si $G(x)$ es de grado 16, tendremos un residuo de grado 15 que equivale a 16 coeficientes de un bit y que se puede empaquetar en 2 bytes.

El CRC es sensible al reordenamiento de los datos y en general es mucho más robusto, sin embargo su desventaja principal es que es demasiado lento en la mayoría de los casos para una tarjeta inteligente sobretodo si se implanta en *software*.

Utilizando un MDC aunque sepamos que los datos están íntegros, no tenemos forma de decidir si podemos confiar en el contenido de un mensaje o no. Los MACs solucionan este problema y merecen ser estudiados con más detalle.

2.3.3. Códigos de Autenticación de Mensajes (MACs)

Los códigos de autenticación de mensajes son funciones huella unidireccionales parametrizadas mediante una clave secreta k.

Como ya se mencionó, su propósito es el de permitir a una segunda parte verificar que nosotros originamos un mensaje dado y por lo tanto que éste se encuentra íntegro.

La clave k es necesaria tanto para generar el MAC como para su verificación y debe compartirse previamente por un canal seguro ya que si es conocida por un tercero, éste podría forjar pares de mensajes y su MAC respectiva.

A partir de un cifrador de bloques es posible construir tanto MDCs como MACs, veamos el esquema más frecuente: el *CBC–MAC*, que está muy relacionado con el modo de operación *CBC*.

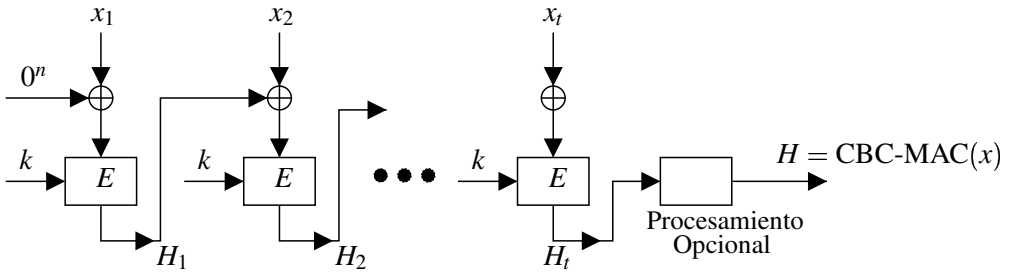

Figura 2.10: Obtención del *CBC–MAC* para un mensaje $x = x_1|x_2|\cdots|x_t$

Para el caso del *CBC–MAC*, a diferencia del modo *CBC*, el vector de inicialización no es tan importante y no tiene que ser único, debido a que H_1 es secreto y no hay problema de que el mismo x_1 sea procesado dos veces al calcular dos MACs.

El procesamiento opcional puede hacer más fuerte al algoritmo, en [30] se sugiere un fortalecimiento del *CBC–MAC* descifrando H_t con una clave distinta k' y posteriormente volviéndola a cifrar con la clave k.

Debido a la naturaleza de las funciones huella, es imposible eliminar la posibilidad de encontrar dos mensajes que produzcan el mismo MAC, sin embargo se busca que para un adversario sea muy difícil generar un mensaje con su correspondiente MAC válido.

Cuando un tercero produce un mensaje con un MAC válido que no fue producido legítimamente por su supuesto originario, se dice que se ha forjado un mensaje. Existen dos tipos de forjados. En orden de severidad son:

- *Forjado Existencial.* El adversario puede forjar un mensaje con su MAC, pero no tiene la capacidad de escoger qué mensaje.
- *Forjado Selectivo.* Aquí el adversario especifica el mensaje del cual generará su MAC correspondiente válido, con el cual podrá hacerlo pasar como auténtico.

Es importante notar que mediante el sólo uso de un MAC, estamos dejando abierta la posibilidad de un ataque por repetición, en el que un criptoanalista copia un par (mensaje, MAC) y lo retransmite tal cual tantas veces como sea necesario y en el momento que considere más oportuno. Como el MAC no incluye *per se* ninguna información sobre la secuencia o validez temporal, el verificador no tiene forma de detectar este tipo de ataques.

Frecuentemente se busca también la confidencialidad, desafortunadamente el uso de MACs no soluciona este problema[7]. En este caso se debe buscar un esquema que combine algún algoritmo de cifrado que proporcione la confidencialidad con un MAC que proporcione integridad y autenticación, o bien con un MDC si no se requiere la autenticación. Una posibilidad para lograrlo es simplemente envolviendo la pareja (mensaje, MAC) con algún algoritmo de cifrado: $c = E_k(m|h_{k'}(m))$ donde $h_{k'}(m)$ es la MAC de m, k es la clave para el cifrado y k' es la clave del MAC.k' debe ser distinta a la k utilizada para cifrar, esto para que en caso de que se pierda la confidencialidad no se pierdan los demás atributos del mensaje.

Es común encontrar conversiones entre algoritmos en las que con base en una primitiva criptográfica obtenemos otra distinta. Un MDC h puede ser utilizado para crear un MAC h_k, sin embargo este proceso no es tan sencillo como parece, en [30] se presentan dos construcciones inseguras: el método del prefijo secreto y el método del sufijo secreto.

2.4. Autenticación de Entidades

En la sección anterior estuvimos hablando sobre la autenticación de origen y la forma de conseguirla mediante el uso de códigos de autenticación de mensajes. En esta sección trataremos otro tipo de autenticación, la *autenticación de entidades*. El objetivo fundamental de los protocolos de autenticación de entidades (o simplemente de autenticación) es que una primera parte demuestre a una segunda parte que ésta está llevando a cabo en dicho momento una interacción con esa entidad y no con otra.

El término *entidad* se aplica tanto para referirnos a personas como a sistemas de cómputo que participan en el protocolo. Obviamente los mecanismos específicos son diferentes para cada caso. Cuando nos referimos a personas, se utiliza más frecuentemente el término *identificación*, aunque en realidad autenticación e identificación pueden emplearse como sinónimos.

Un protocolo requiere la participación de las siguientes partes:

- *Aspirante o Probador*[8]. Es quien pretende demostrar su identidad al verificador.
- *Verificador.* Es quien después de alguna comprobación acepta que la otra parte es quien dice ser.
- *Tercera parte confiable (no siempre está presente).* En algunos protocolos es necesaria una tercera entidad la cual selecciona y proporciona parámetros criptográficos y/o certificados. Los algoritmos que requieren de una tercera parte confiable comúnmente utilizan criptografía asimétrica, y no se hablará más de ellos.

[7] En realidad sí existe una forma de proporcionar confidencialidad mediante MACs, pero en nuestra opinión es poco ortodoxa.

[8] También llamados claimant y prover

Los protocolos de autenticación de entidades se relacionan con los MACs en cuanto a que ofrecen cierto tipo de autenticación. Los protocolos de autenticación en sí mismos no transmiten ningún mensaje o información extra aparte de la identidad que desean probar. Los MACs directamente no establecen un marco temporal sobre la validez del mensaje, por lo que este pudo ser creado mucho tiempo atrás y no requiere la presencia de la entidad que lo creó en la comunicación. Por el contrario, la autenticación asegura que en ese preciso momento se encuentra el probador presente.

Como ya se mencionó en el capítulo sobre tarjetas inteligentes, todas las técnicas de autenticación se basan en tres tipos de características del probador:

- *Algo que se sabe.* Información como claves, *passwords* o PINs la cual se supone que es desconocida para otras entidades.
- *Algo que se tiene.* Este método de cierta forma lo utilizamos todos los días, por ejemplo al abrir una chapa con nuestras llaves. De interés para la criptografía podemos nombrar a las tarjetas inteligentes, generadores de claves y dispositivos de *hardware* para evitar la copia de *software*. En caso de que alguno de estos artefactos se pierda o sea robado, primero que nada debería ser evidente que así ha sido y posteriormente el sistema verificador debe ser notificado.
- *Algo que se es.* Se busca una propiedad intrínseca de la entidad, que varíe lo menos posible y que no pueda ser copiada fácilmente. Cuando se trata de humanos se les llama características biométricas, para objetos inanimados se usan características como su forma, resistencia eléctrica, etc.

El hecho de demostrar la identidad por sí solo no es de ninguna utilidad. La aplicación principal de la autenticación es obtener el acceso a algún recurso, ya sea un sistema de cómputo, una máquina local, un proceso remoto o información almacenada dentro de una tarjeta inteligente.

Para evaluar la seguridad de un esquema de identificación es necesario conocer los objetivos o criterios que se persigue en los protocolos de identificación. En caso de que un aspirante legítimo a una identidad desee autenticarse con un verificador (con el cual está ligado de alguna forma), sin ninguna intervención externa imprevista, entonces el resultado del protocolo será la aceptación de la identidad del aspirante.

Es importante que solamente el aspirante legítimo pueda completar exitosamente el protocolo, en caso contrario se dice que ha habido una *impersonificación*, es decir, un tercero ha adquirido la identidad del aspirante de forma no autorizada.

Por otro lado, el protocolo debe evitar que la demostración de identidad sea transferible. A partir de que un protocolo de autenticación se ha completado satisfactoriamente, el verificador no debe ser capaz de impersonificar al aspirante original hacia un tercero.

Para comparar diferentes protocolos de identificación se sugiere determinar al menos las siguientes propiedades [30]:

1. *Reciprocidad de Identificación.* Hay dos formas de autenticación: *unilateral* y *mutua*. En la unilateral una entidad comprueba su identidad con otra, pero no sucede lo recíproco. En la mutua ambas partes comprueban la identidad de su contraparte. Las tarjetas inteligentes preferentemente utilizan la autenticación mutua con el sistema *host*. El usuario de la tarjeta comúnmente se identifica unilateralmente mediante su PIN.

2. *Eficiencia Computacional.* Número de operaciones o transformaciones que requiere el protocolo. Esto es especialmente importante para las tarjetas inteligentes ya que su capacidad computacional está limitada.

3. *Eficiencia en la Comunicación.* Cantidad de mensajes intercambiados y su longitud en bytes o bits. En las tarjetas inteligentes la comunicación es un proceso lento, para que el sistema reaccione oportunamente se debe minimizar el número de mensajes mandados y su longitud. De acuerdo a [35, p. 148] el tiempo total aproximado (autenticación mutua) es de 218.5 *ms* y al rededor del 46 % se gasta en la transferencia de los datos.

4. *Involucramiento en Tiempo Real de un Tercero.* Es posible que se necesite de una tercera entidad con la cual podamos obtener claves simétricas o certificación.

5. *Almacenamiento de Secretos.* Se refiere a cómo se guardan las claves y en que tipo de dispositivo, por ejemplo, en una tarjeta inteligente en claro protegidos contra su lectura o modificación.

Los métodos con los que se consigue la identificación, de acuerdo a las garantías que ofrecen sobre la no–revelación de claves, pueden clasificarse en tres grandes grupos:

- *Autenticación Débil.*
  Es la forma más elemental de conseguir la autenticación. El aspirante presenta cierta información estática cada vez que desea identificarse, tal como un *password* o un PIN. El problema principal es que un escucha ilegal podría interceptar y aprender dicha información, con la cual posteriormente seriá capaz de impersonificar al aspirante legítimo.

- *Autenticación Fuerte.*
  De lo que se trata es de que el aspirante no proporcione el secreto tal cual al verificador, sino otra información que está en función del secreto y algún parámetro variable con el tiempo (número de secuencia, un número aleatorio, un número único, etc.). El verificador con base en esta información puede determinar si se trata de un aspirante autorizado.

- *Protocolos de Cero Conocimiento*
  La autenticación fuerte puede revelar parte del secreto. Un adversario podía ganar conocimiento sobre la clave paulatinamente hasta recuperar el secreto en el que se basa la seguridad del esquema.

  Una solución a este problema son los protocolos de cero conocimiento en los que es demostrable que el sistema no revelará absolutamente nada de información sobre el secreto en sí, solamente sobre el hecho de que sabe o no el secreto.

  El aspirante proporciona ciertas respuestas a preguntas del verificador. Si el aspirante tiene el secreto podrá contestar correctamente todas las que se le presenten, en caso contrario tiene cierta probabilidad de acertar. El procedimiento puede repetirse hasta hacer que la probabilidad de que las preguntas hayan sido contestadas correctamente sin saber el secreto sea negligible. Los detalles sobre como funciona este tipo de algoritmos están más allá del alcance de esta tesis.

La autenticación débil y la autenticación fuerte juegan un papel muy trascendente para las tarjetas inteligentes por lo que serán estudiadas con mayor profundidad.

2.4.1. Autenticación Débil

Los protocolos de autenticación débil pretenden acreditar una cierta entidad bajo el supuesto de que solamente la entidad legítima posee un secreto determinado, el cual comparte previamente de alguna manera con el verificador. El aspirante demuestra que conoce dicho secreto proporcionándolo tal cual al verificador.

El problema principal de este esquema está en que algún escucha podría interceptar el mensaje con el secreto y aprenderlo, para posteriormente repetirlo e impersonificar a la entidad legítima. O bien, que el adversario se haga pasar por el verificador para que el aspirante ingenuamente le revele el *password*.

El secreto es por lo general un *password* de longitud pequeña, o un PIN. Frecuentemente la autenticación débil es utilizada con personas. Esto trae consigo ciertos problemas, principalmente porque las personas olvidan fácilmente sus *passwords* o no hacen buen manejo de ellos. Cuando pueden escogerlos, por lo general tienen baja entropía y un adversario puede encontrarlos probando primero lo más factibles y luego los menos. El resultado será que el número de cifrados o intentos es mucho menor que si se utilizara aleatoreamente todo el espacio de *passwords*.

Un ejemplo que materializa esta deficiencia es un ataque de diccionario, en el que se tiene una lista precalculada (se puede generar una nueva) de *passwords* probables en orden decreciente de probabilidad y que son buenos candidatos de ser el correcto. Un adversario va intentando acceder al sistema utilizando secuencialmente cada uno de ellos.

Desde el punto de vista práctico y no tanto criptográfico, el verificador se encuentra con el problema de mantener una lista secreta con los *passwords* e identidades, ya que si esta información se libera se compromete totalmente la seguridad del sistema. Por esto si el archivo de *passwords* se quiere guardar en claro se debe restringir su acceso para que no sea leído o escrito excepto por las rutinas de autenticación del sistema. En sistemas reales esto puede ser algo complicado, sobretodo si pensamos en temas como la memoria virtual y en los respaldos al sistema.

Existen algunas técnicas que mejoran la seguridad del esquema presentado. La más importante es la aplicación de funciones unidireccionales sobre los *passwords* en claro. El secreto sigue transmitiéndose en claro del aspirante al verificador, pero éste no tiene una lista de *passwords* en claro sino una lista con las imágenes de una función *hash* sobre ellos. La verificación no se hace ahora comparando el secreto en claro sino la imagen calculada del *password* transmitido con la imagen almacenada. Este esquema es comúnmente referido como "encriptar los *passwords*" aunque estrictamente no se están cifrando (no es una función bidireccional).

Veamos cuáles son las ventajas de utilizar esta técnica. Por las características de las funciones unidireccionales, es muy difícil recuperar el *password* original a partir de su imagen. Debido a esto ya no es necesario mantener la lista de secreta. Lo que sí es necesario es mantener la integridad, de lo contrario un contrincante podría suplantar la imagen de un *password* por la imagen del *password* de su predilección. Concretamente solamente se requiere proteger el archivo con las imágenes de los secretos contra escritura. Es importante hacer notar que la técnica de ataque por diccionario descrita sigue aplicando prácticamente tal cual, solo que ahora se necesita aplicar la función unidireccional a cada *password* y continuar la búsqueda de la misma forma hasta que una imagen de un *password* del diccionario coincida con una imagen del archivo de *passwords*.

Para disminuir la eficacia de un ataque por diccionario, puede emplearse la técnica llamada *salting*, la cual consiste en modificar la transformación unidireccional con base en un cierto campo llamado *salt*. Dos *passwords* iguales pero que tienen asociado un *salt* diferente tendrán una imagen distinta. Ya no es

posible utilizar un solo diccionario ya que para cada *salt* tendríamos uno distinto. Si bien esto aumenta la seguridad en general del sistema, la seguridad de una entidad o usuario específico es prácticamente la misma porque el adversario puede realizar una búsqueda con un diccionario creado justamente con el *salt* del usuario respectivo.

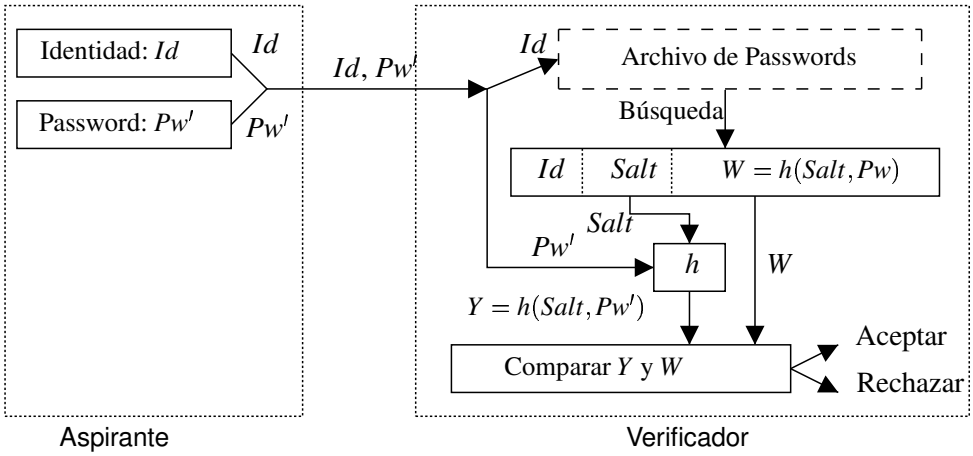

Figura 2.11: Esquema de la autenticación débil incorporando una función unidireccional h y un *salt*.

Una técnica útil para incrementar el trabajo del criptoanalista es aumentar la carga computacional de la función unidireccional, así cada intento será más tardado y podemos llegar a un punto en el que el tiempo estimado para recuperar un *password* sea suficientemente largo como para desalentar a un rival, o para que la información que se protege para el momento en que se comprometa ya no sea valiosa[9].

2.4.2. Autenticación Fuerte

En la sección anterior se mostraron las deficiencias de la autenticación débil y algunas técnicas para fortalecerla, sin embargo el problema de que un escucha ilegal adquiera el *password* en claro no pudo ser eliminado. Una solución a este problema es la autenticación fuerte, también conocida como protocolos de reto–respuesta. El secreto nunca es transmitido en claro sino solamente información derivada de él y de la información del reto.

El reto consiste en un parámetro que varía con el tiempo y que difícilmente se repite. Los tres tipos utilizados son números aleatorios, números secuenciales y estampas de tiempo. Examinemos cada uno de ellos. Las estampas de tiempo evitan un ataque por repetición ya que son válidos solamente por un intervalo corto de tiempo. El problema principal surge cuando nos damos cuenta de la necesidad de mantener sincronizados los relojes, lo cual no es sencillo y para las tarjetas inteligentes comunes que no cuentan con reloj interno ni una fuente de alimentación permanente es muy difícil.

Los números secuenciales se refieren a utilizar un número tomado de una secuencia monotónicamente creciente. En este caso el número actual de la secuencia debe establecerse de antemano entre ambas partes. Aplicada esta idea a las tarjetas inteligentes, es necesario asegurarnos que cuando utilicemos el

[9]El esquema tradicional de protección de los *passwords* en *UNIX* utiliza 12 bits de *salt* y aplica una función unidireccional que consiste en 25 iteraciones de una variante del cifrado *DES*, utilizando el *password* como clave. El algoritmo *DES* ha sido modificado para dificultar su implantación en dispositivos de *hardware*, que de otra forma podrían acelerar el proceso de un ataque por diccionario o por fuerza bruta.

número actual modifiquemos el estado de la tarjeta para que en la siguiente sesión no lo repitamos, esto incluso si la sesión fue interrumpida retirando la tarjeta antes de tiempo. De nuevo aparece el problema de la sincronización.

Los números aleatorios son una opción más viable al menos para las tarjetas inteligentes, aunque también muestran inconvenientes, especialmente la dificultad de generarlos.

A continuación presentamos dos protocolos de autenticación fuerte que nos interesan porque son especialmente aplicables a las tarjetas inteligentes y utilizan algoritmos de clave simétrica [30], la notación empleada es la siguiente: **A** es el aspirante, **B** es el verificador, r_B es un número aleatorio generado por **B**, r_A es un número aleatorio generado por **A**, $E_k(m)$ es el cifrado del mensaje m mediante la clave secreta k que comparten el verificador y el aspirante legítimo. B^* es un identificador del verificador.

- *Autenticación unilateral mediante números aleatorios.*
  El protocolo es el siguiente:

  $$\mathbf{A} \leftarrow \mathbf{B}: \quad r_B \tag{2.25}$$

  El verificador manda al aspirante un número aleatorio.

  $$\mathbf{A} \rightarrow \mathbf{B}: \quad E_k(r_B|B^*) \tag{2.26}$$

  El aspirante concatena el número aleatorio con un identificador de **B** (B^*), el resultado se manda al verificador.

  Después el verificador descifra el criptograma y verifica que tiene el mismo número aleatorio r_B que mandó y que el identificador sea el adecuado. En caso de que estas condiciones se cumplan acepta la identidad del aspirante.

  Es importante notar que solamente se intercambian dos mensajes y no son tan largos, lo cual es aceptable considerando las limitaciones de las tarjetas inteligentes, pero normalmente se desea que la autenticación sea mutua.

- *Autenticación mutua mediante números aleatorios.*
  Al protocolo es muy similar al anterior pero ahora ambas entidades generan un número aleatorio:

  $$\mathbf{A} \leftarrow \mathbf{B}: \quad r_B \tag{2.27}$$

  El verificador manda al aspirante un número aleatorio.

  $$\mathbf{A} \rightarrow \mathbf{B}: \quad E_k(r_A|r_B|B^*) \tag{2.28}$$

  $$\mathbf{A} \leftarrow \mathbf{B}: \quad E_k(r_B|r_A) \tag{2.29}$$

  La parte de la autenticación de **A** es totalmente análoga al caso anterior. En cuanto a la autenticación de **B**, ahora **A** acepta la identidad de **B** si cuando descifra el tercer mensaje coinciden los números obtenidos con los números aleatorios utilizados para generar el segundo mensaje.

En [35, pp. 144–148] se describen los mecanismos simétricos de autenticación de las tarjetas inteligentes, como puede observarse tanto para autenticación unilateral como bilateral son muy similares a los dos protocolos mostrados, con la excepción de que el campo de identificación B^* no está presente.

La función de cifrado $E_k(m)$ debe operar sobre el mensaje completo, no es conveniente que se aplique por bloques independientemente, como en el caso del modo *ECB*, porque se presta para un ataque mediante la manipulación de los bloques.

2.5. Algunos Antecedentes Matemáticos

Hemos hablado hasta ahora de los cifradores de bloques como si fueran una caja negra que simplemente toma una entrada en claro y una clave y produce un criptograma a su salida, sin embargo se aproxima el momento en que hablaremos sobre el funcionamiento interno del algoritmo *AES*. Antes de comenzar de lleno la descripción del *AES* y de las transformaciones que lo constituyen, es conveniente establecer la teoría básica sobre las operaciones matemáticas elementales mediante las que se especifican dichas transformaciones.

Las operaciones en el *AES* involucran diferentes estructuras algebraicas y la aritmética entera convencional no se puede aplicar directamente. Una parte fundamental de la representación de los datos está relacionada con los campos de *Galois* de característica 2, y con polinomios definidos sobre estos campos.

Partiremos de estructuras algebraicas más generales, tales como los grupos y avanzaremos hacia estructuras más específicas como los anillos y los campos. Explicaremos como se realizan las operaciones dentro de estas estructuras. Mencionaremos algunas propiedades y definiciones fundamentales, y más que mostrar un estudio detallado y formal de las estructuras algebraicas, trataremos de establecer los conceptos más relevantes y útiles para el desarrollo del *AES*.

Las fuentes fundamentales de la teoría que se presenta a continuación son: [17], [7], [13], [30] y [9]. En caso de que se desee una estudio más completo y preciso, se recomienda consultar esta bibliografía o algún libro de álgebra abstracta.

2.5.1. Grupos

Un *grupo* es una dupla ordenada $(G,*)$ donde G es un conjunto y $*$ una operación binaria que cumplen con las siguientes propiedades:

1. *Cerradura.* $\forall a,b \in G, \quad a*b \in G$. Lo cual quiere decir que si operamos dos elementos cualesquiera de G con $*$ el resultado también estará en G. En ocasiones se omite esta propiedad porque va incluida al decir que $*$ es una operación binaria.

2. *Asociatividad.* $\forall a,b,c \in G, \quad a*(b*c) = (a*b)*c$.

3. *Elemento idéntico.* $\exists e \in G \,|\, \forall a \in G, \quad a*e = e*a = a$. Existe un elemento e llamado elemento idéntico tal que al ser operado con cualquier elemento de G preserva su valor.

4. *Elementos inversos.* $\forall a \in g, \quad \exists a^{-1} \in G \,|\, a*a^{-1} = a^{-1}*a = e$. Esto significa que cada elemento de G tiene un elemento inverso.

Si además de las propiedades mencionadas un grupo presenta conmutatividad ($\forall a,b \in G, \quad a*b = b*a$), entonces se le denomina *grupo conmutativo* o *grupo abeliano*.

Las condiciones mostradas son suficientes para comenzar a derivar otras propiedades de los grupos, por ejemplo: el elemento idéntico es único, cada elemento tiene un inverso único, el inverso del inverso es el elemento original, el inverso del resultado de una operación es igual a la operación con los inversos de los operandos en el orden contrario $(a*b)^{-1} = (b^{-1}*a^{-1})$, entre otras.

Veamos algunos ejemplos de grupos:

- Los números enteros, racionales, reales y complejos forman un grupo con la operación común de adición $+$ respectiva . El elemento idéntico es 0 y el inverso de a es $-a$.

- Los números racionales, reales y complejos quitando el elemento cero forman un grupo con la operación de multiplicación $\times$ respectiva. El elemento idéntico es 1 y el inverso de a es $a^{-1} = 1/a$. Aquí los enteros no forman un grupo porque la mayoría de los elementos no tienen inverso.

Como se acaba de mostrar hay variaciones en la notación. Hay una notación aditiva y otra multiplicativa. Los inversos se representan como a^{-1} y $-a$ para estas notaciones respectivamente. Además de la diferencia entre la notación de los elementos inversos, si queremos representar $(a * a * \cdots * a)$ con r veces el elemento a, podemos abreviar como a^r en la notación multiplicativa y como $r \cdot a$ en la notación aditiva.

En ocasiones se habla del grupo y del conjunto que forma el grupo bajo un mismo nombre, por el contexto debe quedar claro cuando nos referimos al grupo y cuando al conjunto. Las siguientes definiciones utilizan la notación multiplicativa.

El orden de un grupo G, denotado como $|G|$ es el número de elementos que forman el conjunto. Para los ejemplos mencionados el orden ha sido infinito, pero puede ser finito, tal es el caso del grupo de los enteros módulo n, Z_n, bajo la operación de suma cuyo orden es n.

El orden de un elemento $a \in G$ es el menor entero n tal que $a^n = 1$ (notación multiplicativa) y se denota como $|a|$. Existe una cierta relación entre el orden de los elementos de G con el orden de G dada por el teorema de *Lagrange*.

Un elemento a de orden n permite definir un conjunto P_a formado por las potencias distintas de a: $P_a = \{a, a^2, \ldots, a^n = 1\}$, una notación alternativa para P_a es $\langle a \rangle$. Si el orden de algún elemento α del grupo G es igual al orden del grupo, entonces se dice que α genera G y tenemos que $P_\alpha = G$, además se dice que el grupo G es cíclico.

2.5.2. Anillos

Los grupos solamente se definen para una operación, sin embargo en ocasiones es útil tener estructuras con propiedades que involucren simultáneamente a dos operaciones sobre un mismo conjunto, tal es el caso de los anillos.

Un *anillo* es una tripla $(R, +, \cdot)$ donde R es un conjunto, $+$ y $\cdot$ son dos operaciones binarias por conveniencia llamadas adición y multiplicación respectivamente y que cumple las siguientes condiciones:

1. $(R, +)$ forman un grupo abeliano.

2. La operación $\cdot$ es cerrada bajo R, es asociativa y tiene un elemento idéntico 1. A dicho elemento se le llama la *unidad del anillo*.

3. Ambas operaciones se relacionan por la distributividad bilateral de la multiplicación sobre la adición: $\forall a, b, c \in R, \quad a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ y $\forall a, b, c \in R, \quad (b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

Nótese que la definición encontrada en [17, p. 225] difiere de la nuestra y de las demás encontradas en la bibliografía porque ahí no se incluye la condición de que la operación $\cdot$ tenga elemento idéntico. Para ellos, éste es un caso particular de anillo que llaman anillo con identidad.

Los anillos pueden adquirir calificativos si presentan algunas propiedades. Un anillo cuya multiplicación es conmutativa se le denomina *anillo conmutativo*.

Algunos ejemplos de anillos son los siguientes:

- Los números enteros, racionales, reales y complejos con las operaciones respectivas de adición y multiplicación son un anillo conmutativo. Excepto por los números enteros, estos anillos presentan propiedades que los hacen un tipo de anillo más especial, los campos.

- Los enteros módulo n, Z_n, con aritmética modular también son un anillo conmutativo.

2.5.3. Campos

Las operaciones involucradas en el *AES* requieren estudiar un tipo de anillo llamado *campo*, en particular nos interesan los campos con un número finito de elementos.

Un campo F es un anillo conmutativo en el que cada elemento distinto de cero tiene un inverso multiplicativo.

Otra forma de definir a un campo $\langle F, +, \cdot \rangle$ es que tanto $\langle F, + \rangle$ y $\langle F - \{0\}, \cdot \rangle$ son grupos abelianos y tenemos la propiedad de distributividad de la multiplicación sobre la adición.

La característica de un campo F es el menor entero positivo p tal que $\sum_{i=1}^{p} 1 = 0$, si este entero existe. En caso contrario se define la característica de F como cero.

La característica en el primer caso necesariamente es un número primo, de lo contrario ($p = a \cdot b$) tendríamos:

$$\sum_{i=1}^{p} 1 = 0 = \sum_{i=1}^{a} 1 \cdot \sum_{i=1}^{b} 1 \tag{2.30}$$

lo que implica que alguna de las dos últimas sumatorias es igual a cero lo cual contradice la hipótesis sobre p.

En la sección anterior se dieron algunos ejemplos de campos infinitos, aquellos con un número infinito de elementos, sin embargo para la criptografía los campos de mayor interés son los campos finitos.

Campos Finitos

Los campos finitos permiten manejar cada elemento del campo con una cierta cantidad determinada de memoria. Al realizar una operación, ya sea adición o multiplicación, siempre tendremos una operación inversa bien definida que la cancela, es decir, tenemos transformaciones invertibles. Un campo finito es un campo con un número finito de elementos. Otro nombre para este tipo de campos es *campos de Galois*.

El orden n de un campo es el número de sus elementos y siempre es una potencia entera positiva m de un número primo p. Un campo de orden n se denota como $GF(n)$ y su característica es p.

En realidad para cada potencia de un número primo p^m, estructuralmente existe exactamente un campo de *Galois* $GF(p^m)$ y cualquier otro campo $GF_2(p^m)$ del mismo orden es isomórfico con el primero.

Un homomorfismo es un mapeo de una estructura algebraica a otra en que se preserva la estructura de las operaciones de la primera estructura a la segunda.

Un homomorfismo para dos anillos R y S se define como un mapeo $\varphi : R \to S$ tal que se satisfacen las siguientes dos condiciones:

- $\varphi(a+b) = \varphi(a) + \varphi(b)$ para todo $a, b \in R$. Notar que la operación $+$ del lado izquierdo de la igualdad es la adición en R y la del lado derecho es la adición en S.
- $\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$ para todo $a, b \in R$. Aquí también hay que notar que las multiplicaciones son distintas, la primera es la multiplicación en R y la segunda es la multiplicación en S.

Si dicha función φ es biyectiva, entonces se trata de un isomorfismo.

Esto quiere decir que los campos de *Galois* $GF(p^m)$ son en esencia el mismo, solamente cambia la representación de sus elementos.

Un ejemplo de un campo de *Galois* cuando n es un número primo p, es Z_p con los números enteros de 0 a $p-1$ y la aritmética módulo p tradicional. Cabe señalar que esto no es válido cuando n no es un número primo, en esos casos es necesaria alguna representación diferente.

Si bien ya ha quedado establecido que todos los campos finitos del mismo orden tienen la misma estructura, la representación concreta de los elementos del campo tiene consecuencias en cuanto a la facilidad de cómputo requerido para efectuar las operaciones del campo.

Antes de ver dos representaciones populares de los campos, debemos presentar la teoría de polinomios sobre campos.

2.5.4. Polinomios Sobre Campos

Un polinomio $f(x)$ en la variable indeterminada x sobre un campo F es una expresión de la forma:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 \tag{2.31}$$

El grado del polinomio es el mayor valor n tal que a_n es distinto de cero. En caso de que todos los coeficientes sean cero se dice que el grado del polinomio es $-\infty$. Cuando el término de mayor grado es igual a 1 se dice que el polinomio es mónico.

Las operaciones para manipular los polinomios son la multiplicación y suma común de polinomios, excepto que al momento de hacer operaciones entre los coeficientes se utilizan las operaciones definidas en el campo. El conjunto de polinomios sobre un campo F se denota como $F[x]$. Al conjunto de polinomios de grado menor a l se le denota como $F[x]|_l$

Al igual que con los enteros, para los polinomios surge el concepto de aritmética modular, la diferencia es que en este caso el módulo no es un entero sino un polinomio. De manera similar al concepto de primalidad en los números enteros, en los polinomios sobre campos aparece el concepto de irreducibilidad.

Un polinomio $f(x)$ de grado al menos 1 sobre un campo F es irreducible sobre F si no puede expresarse como el producto de dos polinomios de $F[x]$ ambos de grado mayor o igual a 1.

Hay que notar que un polinomio irreducible sobre un campo puede no ser irreducible para otro campo. Para ejemplificar esto, veamos el polinomio $f(x) = x^2 + 1$, sobre el campo $GF(2)$ dicho polinomio es reducible ($f(x) = (x+1)(x+1)$), para los reales es irreducible ($f(x)$ no tiene raíces reales) y sobre los complejos es reducible ($f(x) = (x+i)(x-i)$).

La clase de equivalencia de los polinomios en $F[x]$ de grado menor que $n = grado of(x)$, donde las operaciones de adición y multiplicación se realizan módulo $f(x)$ se denota como $F[x]/(f(x))$.

Para entender esto debemos aclarar que se forma una relación de congruencia en la que dos polinomios $g(x)$ y $h(x)$ son congruentes por tener el mismo residuo al ser divididos entre $f(x)$. La idea es análoga a la de la congruencia en aritmética modular.

El circulo se cierra cuando nos percatamos de que $F[x]/(f(x))$ es un anillo conmutativo, y si escogemos correctamente $f(x)$ (que sea irreducible sobre F) entonces tenemos un nuevo campo. En términos prácticos, a partir de $GF(2)$ podemos construir mediante la representación polinomial $GF(2^m)$, o al menos un anillo conmutativo.

Un elemento $\alpha \in F$ es un elemento primitivo de F si todo elemento de F distinto de cero puede ser generado mediante las potencias de α. Un polinomio irreducible $f(x) \in Z_p[x]$ es llamado polinomio primitivo si x es un generador de $F^*_{p^m}$, donde $F^*_{p^m}$ es el grupo multiplicativo de todos los elementos distintos de cero en $F_{p^m} = Z_p[x]/(f(x))$.

Ahora es conveniente que mostremos de forma concreta de que forma podemos generar y representar los elementos de un campo finito, además de realizar las operaciones con sus elementos. En particular veremos la representación polinomial y la exponencial. La representación como vector consiste en meter a un vector los coeficientes de los polinomios de la representación polinomial (cuando se utiliza la base

polinomial, aunque hay otras bases) y se omitirá.

La representación polinomial de un campo $GF(p^m)$ consiste en considerar a los elementos como polinomios sobre $GF(p)$ de grado menor a m. Las operaciones de adición y multiplicación requieren de un polinomio irreducible $f(x)$ sobre $GF(p)$.

Para ejemplificar esto, tomemos al polinomio irreducible $f(x) = x^5 + x^2 + 1$ sobre $GF(2)$, el campo $GF(2^5)$ es el campo $GF(2)[x]/f(x)$ que se forma por todos los polinomios de grado menor a cinco con las operaciones sobre polinomios módulo $f(x)$.

La suma de $x^4 + x^2 + x + 1 \in GF(2)[x]/f(x)$ con $x^2 + 1 \in GF(2)[x]/f(x)$ se obtiene sumando cada coeficiente independientemente y el resultado es $(x^4 + (1+1)x^2 + x + (1+1)) = x^4 + x$.

La multiplicación puede realizarse primero multiplicando los polinomios y luego sacándoles el módulo $f(x)$: $(x^4 + x^2 + 1) \cdot (x^2 + 1) = x^6 + (1+1)x^4 + (1+1)x^2 + 1 = x^6 + 1$, calculando el módulo de $x^6 + 1$ obtenemos $x^3 + x + 1$.

Como puede observarse para esta representación es más sencillo efectuar las sumas (las cuales corresponden a una suma módulo 2) que las multiplicaciones.

Polinomial	**Exponencial**
0	0
1	α^0
x	α^1
x^2	α^2
x^3	α^3
x^4	α^4
x^2+1	α^5
x^3+x	α^6
x^4+x^2	α^7
x^3+x^2+1	α^8
x^4+x^3+x	α^9
x^4+1	α^{10}
x^2+x+1	α^{11}
x^3+x^2+x	α^{12}
$x^4+x^3+x^2$	α^{13}
$x^4+x^3+x^2+1$	α^{14}

Polinomial	**Exponencial**
$x^4+x^3+x^2+x+1$	α^{15}
x^4+x^3+x+1	α^{16}
x^4+x+1	α^{17}
$x+1$	α^{18}
x^2+x	α^{19}
x^3+x^2	α^{20}
x^4+x^3	α^{21}
x^4+x^2+1	α^{22}
x^3+x^2+x+1	α^{23}
$x^4+x^3+x^2+x$	α^{24}
x^4+x^3+1	α^{25}
x^4+x^2+x+1	α^{26}
x^3+x+1	α^{27}
x^4+x^2+x	α^{28}
x^3+1	α^{29}
x^4+x	α^{30}

Cuadro 2.1: Representaciones polinomial y exponencial de $GF(2^5) = GF(2)[x]/f(x)$ con $f(x) = x^5 + x^2 + 1$.

La segunda representación que vamos a ver es la exponencial. Consiste en que mediante un generador podemos describir todos los elementos excepto al cero especificando el exponente correspondiente. Tomando a α (o en esta caso a x) como generador, sabemos que $\alpha^0 = 1, \alpha^1 = x, \alpha^2 = x^2, \ldots, \alpha^5 = x^2 + 1, \alpha^6 = x^3 + x, \ldots$. La representación utiliza un índice i para especificar a cada elemento mediante α^i, excepto al cero que no se puede representar de esta forma y se representa como cero.

Gracias a esta representación podemos efectuar la multiplicación muy fácilmente, por ejemplo: $\alpha^4 \cdot \alpha^2 = \alpha^{(4+2)} = \alpha^6 = x^3 + x$. El producto equivale a la suma de los exponentes involucrados módulo $p^m - 1$. La multiplicación que mostramos al explicar la representación polinomial equivale a: $\alpha^{22} \cdot \alpha^5 = \alpha^{27}$ con

la representación exponencial.

Debido a que $\alpha^{p^m-1} = \alpha^0 = 1$ tenemos una forma fácil de encontrar los inversos multiplicativos, el inverso de α^k es α^{p^m-1-k} ya que $\alpha^k \cdot \alpha^{p^m-1-k} = 1$.

La adición es más compleja, no es tan sencillo determinar que $\alpha^0 + \alpha^4 = \alpha^{10}$, lo que equivale simplemente a $(1) + (x^4) = x^4 + 1$ en la notación polinomial.

Hay que señalar que la estructura resultante de los polinomios sobre un campo F operados módulo $f(x)$, $F[x]/(f(x))$ forma un campo solamente cuando el polinomio $f(x)$ es irreducible, cuando no es así tenemos solamente un anillo conmutativo. Los campos garantizan la existencia de inversos multiplicativos para todos los elementos distintos de cero, sin embargo los anillos no garantizan esto. Si queremos hacer una transformación invertible en un anillo conmutativo no podemos multiplicar un elemento original $g(x)$ con un segundo elemento cualquiera $h(x)$ porque es probable que el segundo no tenga elemento inverso para revertir la transformación, por ello debemos escoger al segundo elemento $h(x)$ para que sí tenga inverso. Esto se consigue haciendo que no tenga factores en común con $f(x)$.

Capítulo 3

El *AES* (Estándar Avanzado de Cifrado)

3.1. Introducción

El algoritmo criptográfico más utilizado aún hasta ahora es muy probablemente el *DES*[1] o Estándar de Cifrado de Datos. La *NBS*[2] u Oficina Nacional de Estándares de los EUA a mediados de la década de los 70 concluyó un esfuerzo para establecer un algoritmo común de cifrado. El algoritmo resultante fue el *DES* y su ámbito de operación inicial era la comunicación del gobierno que involucraba información no clasificada, aunque también tuvo una muy buena aceptación en la industria (p. ej. en bancos).

Desde el principio aparecieron diversas objeciones en contra del *DES* y está claro que con los avances en las técnicas de criptoanálisis y con el aumento del poder computacional disponible, el nivel de seguridad que ofrece ya no es suficiente y es necesario migrar a un algoritmo más seguro.

El *NIST*[3] (antes la *NBS*) a principios de 1997 comenzó un proceso para la selección de un nuevo algoritmo de cifrado que sería aprobado como un Estándar Federal de Procesamiento de la Información (*FIPS*[4]) y que sustituirá al *DES* y al *Triple DES*. El nombre de este estándar es el *AES*[5] o Estándar Avanzado de Cifrado.

La convocatoria estuvo dirigida a compañías e investigadores de todo el mundo sin discriminar nacionalidad. Inicialmente se tenían 15 algoritmos candidatos oficiales. Los algoritmos participantes más adecuados para el estándar y que quedaron como finalistas fueron: *Serpent*, *Twofish*, *RC6*, *Mars* y *Rijndael*.

Para evaluar a los candidatos se definió qué características se esperaban, entre ellas se incluían:

- *Seguridad.* Por lo menos similar a la del *Triple DES*. Es muy difícil probar la seguridad de un cifrador, por lo que en su lugar se aceptó el hecho de no haber encontrado ninguna vulnerabilidad.
- *Características de la implantación.* Es importante que el algoritmo pueda ser implantado eficientemente en una gran variedad de plataformas como: servidores, computadoras personales y tarjetas inteligentes con procesadores de 8 bits. Debe poder ser implantado tanto en *software* como en *hardware*. También el tiempo que toma la preparación de un cifrado (por ejemplo para hacer la expansión de la clave) debe ser corto. Otro factor que se solicitó fue la simplicidad.

[1] *Data Encryption Standard*.
[2] National Bureau of Standards.
[3] *National Institute of Standards and Technology*.
[4] *Federal Information Processing Standard*.
[5] *Advanced Encryption Standard*.

Finalmente, el algoritmo seleccionado para convertirse en el *AES* fue el conocido como *Rijndael*[6], un cifrador creado por dos criptógrafos investigadores Belgas: *Joan Daemen* y *Vincent Rijmen*.
El *AES* cifra bloques de 128 bits (16 bytes) con claves de 128, 198 o 256 bits (16, 24 o 32 bytes). Estudiaremos el funcionamiento del *AES*, pero nos limitaremos al tamaño de clave de 128 bits. Las fuentes principales de información de este capítulo son directamente la propuesta de *Rijndael* para convertirse en el *AES* [12] y el libro de texto *"el diseño de Rijndael"* [13].

A continuación se explicará la estructura interna del algoritmo, las operaciones que efectúa y se hará referencia a documentos relacionados con su criptoanálisis. Se pondrá énfasis en aclarar o ejemplificar mejor algunas partes de la especificación que en nuestra opinión merecen mayor atención. Intentarémos describir las transformaciones para facilitar la comprensión de nuestra futura implantación en ensamblador.

3.2. Especificación del *AES*

La mayoría de las transformaciones tienen lugar sobre lo que se denomina el *estado*, que es un arreglo matricial de 4 renglones por N_b columnas, donde cada elemento es un byte. El estado directamente establece la longitud de los bloques. Para el *AES* $N_b = 4$ y con ello los bloques resultantes son de 128 bits (16 bytes).

Algo similar sucede con la clave. También puede representarse como una matriz de cuatro renglones, pero el número de columnas N_k es variable: 4 columnas (claves de 128 bits), 6 columnas (claves de 198 bits) u 8 columnas (claves de 256 bits).

En realidad el *AES* acepta solamente una selección de algunos de los tamaños de bloque y clave que soporta *Rijndael*, los cuales en principio son independientes y pueden variar con la condición de tener longitudes de entre 128 y 256 bits siempre y cuando sean múltiplos de 32 bits.

Rijndael es un *cifrador de clave iterada* (ver página 44). Primero que nada el mensaje en claro se copia al estado y es transformado mediante sumas de claves de ronda o transformaciones de ronda independientes de la clave. Después de un cierto número de iteraciones, 10 para el tamaño de clave de 128 bits, el estado se ha convertido en el criptograma y podemos extraerlo directamente.

El bloque de texto en claro m puede escribirse como un grupo de bytes $m = p_0 p_1 p_2 \cdots p_{4 \cdot N_b - 1}$, con p_0 como el primer byte del texto en claro. La clave c puede expresarse como $c = c_0 c_1 c_2 \cdots c_{4 \cdot N_k - 1}$.

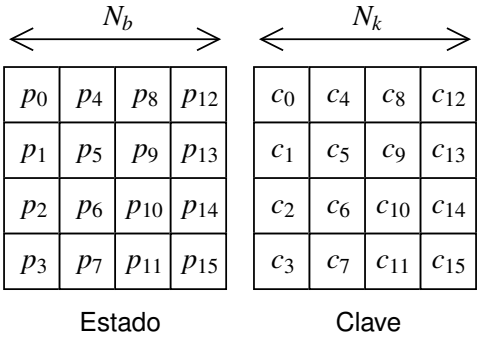

Figura 3.1: Disposición del estado inicial ($N_b = 4$) y de la clave del cifrador ($N_k = 4$).

[6]Nombre que proviene de los apellidos de sus autores.

En la figura 3.1 podemos observar cómo se colocan los bytes en el estado y la clave. El resultado será totalmente distinto si colocamos los datos en otro orden, por ejemplo transponiendo las matrices. En la figura mostramos la clave del cifrador que equivale en este caso a la clave de la ronda 0. La clave de ronda siempre es del mismo tamaño que el estado.

Una vez que hemos cargado el mensaje en claro en el estado, éste se afectará por la suma de la clave de ronda. Después pasará por $N_r - 1$ rondas idénticas, y por último aplicaremos la ronda final que es muy similar a la ronda normal. Todas las transformaciones de ronda involucran, como uno de los pasos que las componen, la suma al estado de la clave de ronda correspondiente.

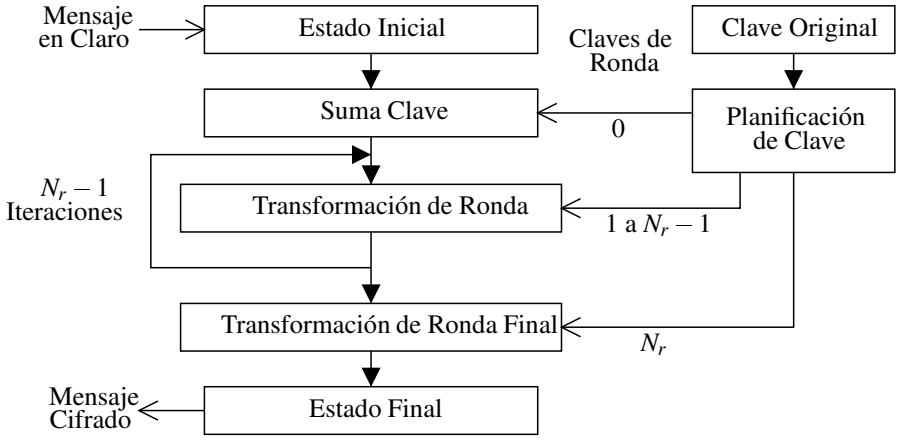

Figura 3.2: Esquema del cifrado completo.

Es posible analizar separadamente las transformaciones de ronda y el proceso mediante el cual se obtienen las claves de ronda a partir de la clave original (la *planificación de clave*). Veamos entonces la transformación de ronda y después la planificación de clave.

3.2.1. Transformación de Ronda

Cada ronda se compone de cuatro transformaciones o pasos muy diferentes con funciones bien definidas, cada uno de ellos opera sobre el estado:

- *SubBytes (Sustituye Bytes).* Sustituye individualmente cada byte del estado por otro de acuerdo a una tabla fija.
- *ShiftRows (Recorre Renglones).* Toma cada renglón del estado completo (N_b bytes) y hace un corrimiento cíclico un determinado número de bytes o columnas que depende del renglón del que se trate.
- *MixColumns (Mezcla Columnas).* Opera idénticamente con cada columna completa (4 bytes) aplicando una transformación lineal.
- *AddRoundKey (Suma Clave de Ronda).* Modifica el estado sumándole módulo 2 (XOR) byte a byte la clave de la ronda correspondiente.

La transformación de ronda final es idéntica a la transformación de ronda normal excepto que se omite el paso *Mezcla Columnas* por razones que serán aclaradas cuando tratemos el cifrador inverso.

Cada uno de los pasos le da un significado a los datos de acuerdo a representaciones distintas en las cuales se tienen diferentes operaciones. Expliquemos cada paso con más detalle.

3.2.2. *SubBytes*

Se trata de una permutación que convierte cada byte de entrada en un byte de salida aplicando un mapeo o sustitución. La sustitución siempre es la misma para los 16 bytes del estado. También se utiliza este mapeo (las cajas S) en la planificación de clave.

Este paso es sumamente importante porque se trata de la única transformación no lineal del algoritmo y varias propiedades útiles en el criptoanálisis diferencial y lineal se relacionan con ella, sin esta transformación sería trivial romper el algoritmo.

En caso de consultar una especificación vieja de *Rijndael*, la nomenclatura para esta función cambia ligeramente y es llamada *ByteSub*. A su vez la transformación *SubBytes* puede expresarse como la composición de dos funciones: Una inversión g sobre $GF(2^8)$ que es el componente no lineal y una función *affine* f:

- *Transformación g*
  Esta transformación involucra los principios básicos de la teoría de campos de *Galois* que explicamos en el capítulo anterior.

  La transformación g mapea un elemento a a un elemento b de $GF(2^8)$. La representación de dicho campo es mediante polinomios sobre el campo $GF(2)$, con operaciones módulo el polinomio irreducible escogido para este algoritmo $m(x) = x^8 + x^4 + x^3 + x + 1$. Es decir podemos representar a a y b como polinomios de grado menor o igual a 7 con coeficientes booleanos. De acuerdo a la notación del capítulo anterior nuestro $GF(2^8)$ equivale a $GF(2)[x]/(m(x))$.

  Es conveniente recordar este polinomio irreducible $m(x)$, porque todas las veces que aparece $GF(2^8)$ se trata del mismo campo, el cual opera específicamente con respecto a este módulo (porque podría operar con otros módulos). El paso mezclar columnas también ocupa este campo y por consiguiente el mismo módulo.

  La definición de la función g es:

  $$g : a \to b = a^{-1} \tag{3.1}$$

  es decir,

  $$b = g(a) = a^{-1} \tag{3.2}$$

  lo cual significa que b es el inverso multiplicativo de a dentro del campo de trabajo. Para conservar la biyección y superar el problema de que 0 no tiene inverso, definimos a $g(0) = 0$, es decir, el cero se mapea a sí mismo.

  Por ejemplo, el inverso multiplicativo de $x^3 + x$, que equivale al elemento 0x0A en notación hexadecimal, es $x^5 + x^3 + 1$ (0x29). Esto lo podemos comprobar fácilmente ya que:

  $$(x^3 + x) \cdot (x^5 + x^3 + 1) = x^8 + (1 + 1) \cdot x^6 + x^4 + x^3 + x = x^8 + x^4 + x^3 + x$$

  De acuerdo al módulo tenemos:

  $$x^8 \equiv x^4 + x^3 + x + 1$$

por lo que:

$$(x^3+x)\cdot(x^5+x^3+1) \equiv (1+1)\cdot x^4+(1+1)\cdot x^3+(1+1)\cdot x+1 \equiv 1$$

Lo cual comprueba que x^5+x^3+1 es el inverso de x^3+x (y viceversa).

- *Transformación f*
  Se trata de una función *affine* que mezcla los bits de un byte. Toma un byte de entrada a el cual se interpreta como un vector booleano $a = [a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0]^T$ y produce un vector $b = [b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0]^T$. El bit más significativo de a es mapeado al coeficiente 7 y el menos significativo al coeficiente 0. La transformación *affine* consiste en la multiplicación de una matriz M_1 por el vector de entrada y al resultado se le suma un vector constante de la siguiente forma:

$$\vec{b} = f(\vec{a}) \tag{3.3}$$

$$\vec{b} = M_1 \times \vec{a} + \vec{V_1} \tag{3.4}$$

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{3.5}$$

  Donde la matriz M_1 es invertible y cada uno de los coeficientes es interpretado como un elemento de $GF(2)$ por lo que la adición equivale a una suma módulo 2 (*XOR*) y la multiplicación a un *AND*.

La sustitución completa puede expresarse como $S_{RD}(a) = f(g(a))$ y puede implantarse como una consulta a una tabla.

3.2.3. *ShiftRows*

El paso ShiftRows opera sobre cada uno de los renglones del estado efectuando una transposición de los bytes que lo componen. Específicamente se trata de un corrimiento circular a la izquierda. Cada uno de los renglones es recorrido cíclicamente C_i bytes donde i es el número del renglón.

Los corrimientos o desplazamientos de cada renglón en *Rijndael* dependen del tamaño del bloque y están dados en la tabla 3.1. En el *AES* todos los bloques son del mismo tamaño por lo que solamente nos interesa el caso para $N_b = 4$.

Es importante resaltar que los corrimientos son a nivel de bytes y no de bits, por lo que su implantación en *software* es más sencilla e independiente de la plataforma.

Gráficamente podemos entender más fácilmente el efecto de aplicar la transformación *ShiftRows* sobre el estado en la figura 3.3.

N_b	C_0	C_1	C_2	C_3
4	0 (no hay corrimiento)	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Cuadro 3.1: Desplazamientos a la izquierda utilizados por el paso *ShiftRows* de *Rijndael*.

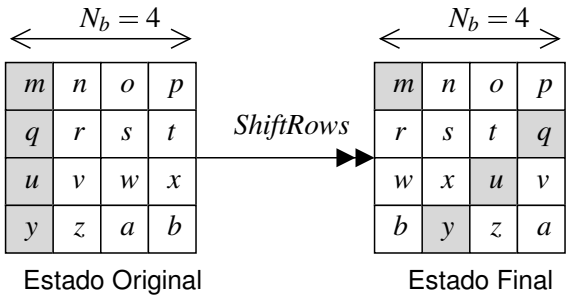

Figura 3.3: Paso *ShiftRows* para el *AES*.

3.2.4. *MixColumns*

La transformación *MixColumns* trabaja de manera idéntica sobre cada una de las columnas del estado (es una función "capa de ladrillo"). Esta operación consigue difusión entre los elementos que forman cada columna y es un mapeo lineal invertible.

La columna i, formada por los elementos $[a_{0,i}\ a_{1,i}\ a_{2,i}\ a_{3,i}]^T$, o simplemente $[a_0\ a_1\ a_2\ a_3]^T$, es interpretada como un polinomio $a(x)$ de grado menor o igual a 3 sobre $GF(2^8)$:

$$a(x) = a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0 \tag{3.6}$$

La transformación es lineal sobre $GF(2)$ y consiste en una multiplicación por otro polinomio $c(x)$ sobre $GF(2^8)$ módulo el polinomio reducible x^4+1:

$$c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02 \tag{3.7}$$

El resultado $b(x)$ después de procesar $a(x)$ es simplemente:

$$b(x) = c(x) \times a(x) \ (\mathrm{mod}\, x^4+1) \tag{3.8}$$

Aquí hay que notar que el polinomio x^4+1 no es irreducible sobre $GF(2^8)$ por lo que no todos los elementos tienen inverso multiplicativo, sin embargo no se pierde la biyección porque el polinomio escogido $c(x)$ sí tiene inverso. Cada columna corresponde a un elemento de un anillo conmutativo, el cual de acuerdo a la notación del capítulo anterior está dado por $GF(2^8)[x]/(x^4+1)$.

La multiplicación de polinomios puede expresarse también como una multiplicación matricial, en donde la columna a es transformada en la columna b de la siguiente forma:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{3.9}$$

Todas las operaciones entre los coeficientes (que se expresan en hexadecimal al igual que en la matriz inversa que veremos más adelante) se realizan en $GF(2^8)$. Como puede observarse los coeficientes son simples. Esto fue planeado para dar una alta velocidad de cifrado ya que las operaciones involucradas pueden implantarse con aritmética de 8 bits.

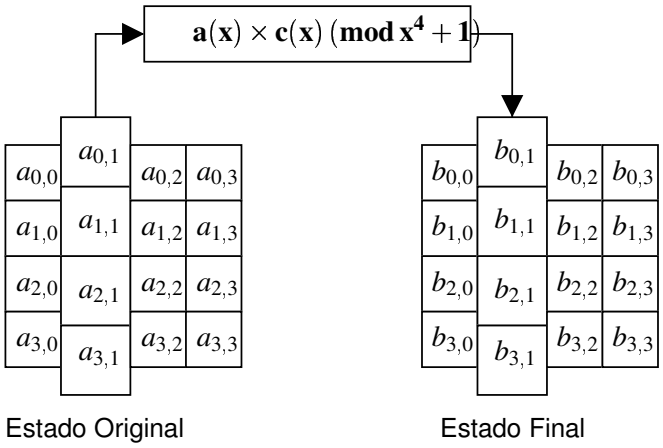

Figura 3.4: Representación del paso *MixColumns* para uno de los renglones del estado.

3.2.5. *AddRoundKey*

La única intervención de la clave sobre el estado se da en este paso. Consiste en una adición módulo 2 (XOR) de la clave de ronda a los bytes del estado. La clave de ronda es derivada de la clave del cifrador y es proporcionada mediante la planificación de clave.

De esta forma el estado resultante puede especificarse de acuerdo a la transformación de cada uno de sus elementos:

$$b_{i,j} = a_{i,j} + k_{i,j} \tag{3.10}$$

donde a representa al estado original, b al estado final y k a la clave de ronda.

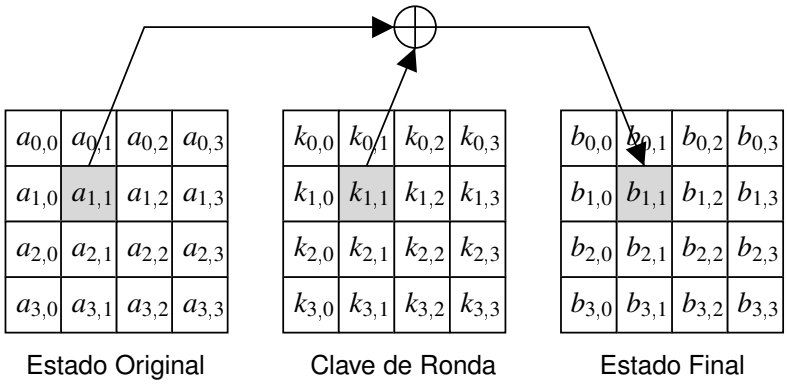

Figura 3.5: Representación del paso *AddRoundKey* enfatizando la transformación de un byte del estado.

3.2.6. Planificación de Clave

En esta sección se describirá el proceso mediante el cual se genera cada una de las claves de ronda a partir de la clave del cifrador para claves de 128 bits. Para las otras longitudes (192 y 256 bits) los algoritmos correspondientes son muy similares, pero no serán tratados aquí.

La planificación de clave se compone de dos procesos: la expansión y la selección de la clave de ronda. La expansión convierte la clave original en $N_r + 1$ claves de ronda (una para la adición de clave inicial y una para cada una de las N_r rondas).

Estamos hablando de $4 \cdot N_b \cdot (N_r + 1) = 176$ bytes de la clave expandida lo cual para la mayoría de las tarjetas inteligentes es demasiado como para mantenerla en memoria *RAM*. Afortunadamente los diseñadores del algoritmo pensaron en esto y es posible derivar la clave de la ronda $i+1$ a partir de la clave de ronda i sin requerir utilizar memoria adicional.

Desgraciadamente la descripción del algoritmo para expandir la clave en las especificaciones está dada en pseudocódigo y utiliza notación que no es tan simple de entender, además hay variaciones en la forma en que se especifica la clave en la propuesta del algoritmo, en el código de referencia en ANSI C y en su documento de diseño, aunque desde luego son equivalentes.

Por ser más útil para comprender nuestra implantación del algoritmo, en vez de mostrar como se genera toda la clave expandida, veremos como se pasa de la clave de ronda i a la clave de ronda $i+1$. La clave de ronda 0 se define igual a la clave del cifrador. En el caso de la expansión de la clave, a diferencia de la transformación de ronda, no se tiene simetría completa respecto a todas las iteraciones, sino que para generar la clave de ronda i interviene una constante $RC[i]$ para $i = 1 \ldots N_r$ que hace variar ligeramente el proceso con el fin de prevenir cierto tipo de ataques.

Las constantes $RC[i]$ pertenecen a $GF(2^8)$ y se obtienen de la siguiente forma (representando a los elementos de $GF(2^8)$ como polinomios sobre $GF(2)$):

$$RC[i] = x^{i-1} \qquad i = 1 \ldots N_r \tag{3.11}$$

Los valores de $RC[i]$ pueden consultarse en la tabla 3.2.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
RC[i]	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36

Cuadro 3.2: Valor de las Constantes de Ronda $RC[i]$ empleadas en la expansión de clave.

Para dejar las cosas más claras y sin lugar a interpretaciones discrepantes, a continuación presentamos las ecuaciones que permiten calcular la siguiente clave de ronda k^2 a partir de la actual k^1. Preferimos llamar a las claves de esta forma en lugar de k^i y k^{i-1} para mejorar la legibilidad de las expresiones.

Las siguientes ecuaciones se pueden derivar del pseudocódigo propuesto en la especificación. Las mostramos en el mismo orden en el que podríamos calcularlas sin requerir memoria auxiliar para guardar los datos de la clave de ronda actual:

$$k^2_{0,0} = k^1_{0,0} \oplus S_{RD}(k^1_{1,3}) \oplus RC[i] \tag{3.12}$$

$$k^2_{1,0} = k^1_{1,0} \oplus S_{RD}(k^1_{2,3}) \tag{3.13}$$

$$k^2_{2,0} = k^1_{2,0} \oplus S_{RD}(k^1_{3,3}) \tag{3.14}$$

$$k^2_{3,0} = k^1_{3,0} \oplus S_{RD}(k^1_{0,3}) \tag{3.15}$$

$$k^2_{0,1} = k^1_{0,1} \oplus k^2_{0,0} \quad (3.16)$$
$$k^2_{0,2} = k^1_{0,2} \oplus k^2_{0,1} \quad (3.17)$$
$$k^2_{0,3} = k^1_{0,3} \oplus k^2_{0,2} \quad (3.18)$$
$$k^2_{1,1} = k^1_{1,1} \oplus k^2_{1,0} \quad (3.19)$$
$$k^2_{1,2} = k^1_{1,2} \oplus k^2_{1,1} \quad (3.20)$$
$$k^2_{1,3} = k^1_{1,3} \oplus k^2_{1,2} \quad (3.21)$$
$$k^2_{2,1} = k^1_{2,1} \oplus k^2_{2,0} \quad (3.22)$$
$$k^2_{2,2} = k^1_{2,2} \oplus k^2_{2,1} \quad (3.23)$$
$$k^2_{2,3} = k^1_{2,3} \oplus k^2_{2,2} \quad (3.24)$$
$$k^2_{3,1} = k^1_{3,1} \oplus k^2_{3,0} \quad (3.25)$$
$$k^2_{3,2} = k^1_{3,2} \oplus k^2_{3,1} \quad (3.26)$$
$$k^2_{3,3} = k^1_{3,3} \oplus k^2_{3,2} \quad (3.27)$$

En la figura 3.6 podemos apreciar la transformación mediante la cual se genera la clave de ronda siguiente k^2 a partir de la clave de ronda actual k^1. Es necesario explicar el significado de las flechas y operaciones ejecutadas:

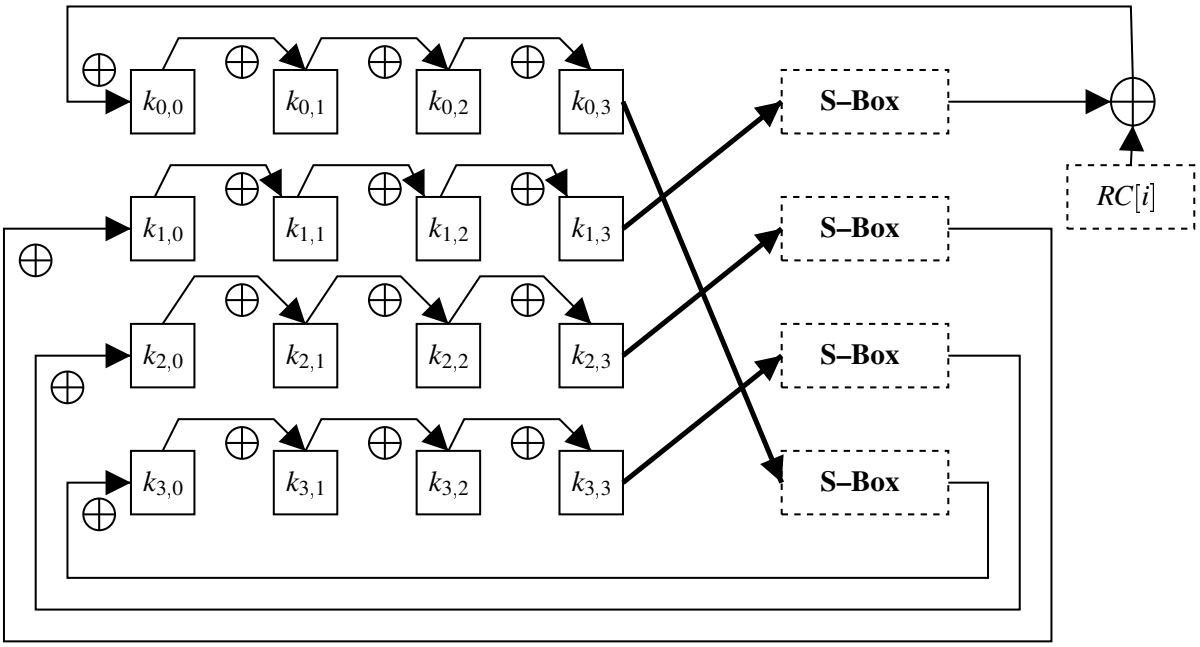

Figura 3.6: Transformación de la clave de ronda $i-1$ en la clave de ronda i. (Ver las notas para entender el significado del diagrama).

- En el esquema se reutilizan las localidades puesto que la clave de ronda siguiente k^2 sobrescribe a la clave de ronda actual k^1.
- Una flecha que llega a una casilla indica el valor que tendrá $k^2_{i,j}$, la etiqueta $\oplus$ significa que el valor no será copiado sino que será sumado módulo 2.

- Una sustitución mediante una S-box igual a las descritas en el paso *SubBytes* se indica con un cuadro que dice "S-box".
- Una flecha gruesa que sale de una casilla (i, j) representa el valor viejo del mismo byte: $k^1_{i,j}$.
- Una flecha común que sale de una casilla (i, j) representa el valor nuevo del mismo byte: $k^2_{i,j}$.

3.3. El Cifrador Inverso

Debido a que todas las transformaciones que sufre el estado son invertibles, el cifrador inverso de la forma directa o más inmediata puede definirse como la composición de las inversas de cada una de las transformaciones pero aplicadas en el orden contrario. En realidad el tipo de operaciones involucradas en el *AES* hace posible modificar el orden en que se aplican y permite llegar a un *cifrador inverso equivalente* con una estructura muy parecida a la del cifrador directo.

Desgraciadamente solamente la transformación *AddRoundKey* es una involución, por lo que todas las demás transformaciones tienen una inversa diferente a la función original, lo que puede causar un mayor tamaño del código. Además el cifrador está especialmente optimizado para efectuar el cifrado directo y el proceso inverso requiere más recursos computacionales.

Otra desventaja del cifrador inverso es que el orden en que se aplican las claves para descifrar es el contrario (primero se usa la última clave de ronda) y el procedimiento para derivar la última clave de ronda requiere calcular cada una de las claves de ronda anteriores. Una forma de evitar esta carga extra es manteniendo la clave expandida completa en memoria. Otra forma es almacenando la última clave de ronda y mediante un proceso iterativo ir calculando cada una de las otras de forma similar a como se explicó en la sección anterior, solo que en la dirección contraria. Esto se debe a que la transformación que afecta a la derivación de claves es bidireccional y a partir de cualquier clave de ronda podemos derivar todas las demás.

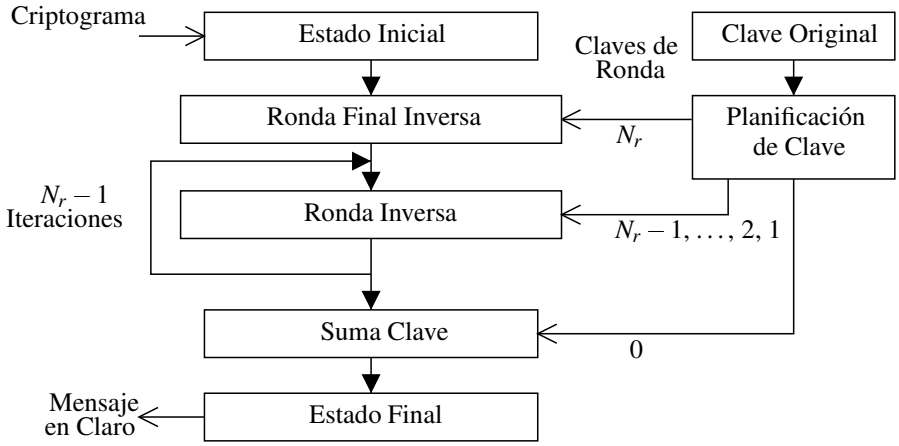

Figura 3.7: Diagrama del algoritmo de descifrado.

Podría pensarse que el cifrado inverso se utiliza tan frecuentemente como el cifrado directo, sin embargo no es así ya que como vimos en el capítulo anterior hay modos de operación que solamente

requieren efectuar el cifrado directo. El procedimiento inmediato para descifrar un criptograma puede observarse en la figura 3.7 y es el siguiente:

- Aplicar la transformación inversa de ronda final que consiste en:
  - Sumar la última clave de ronda (la clave N_r)
  - Transformación recorre renglones inversa (*InvShiftRows*)
  - Transformación sustituye bytes inversa (*InvSubBytes*)
- Aplicar la transformación inversa de ronda N_r veces con $i = N_r - 1, \cdots, 2, 1$:
  - Sumar la clave de ronda i
  - Transformación mezcla columnas inversa (*InvMixColumns*)
  - Transformación recorre renglones inversa (*InvShiftRows*)
  - Transformación sustituye bytes inversa (*InvSubBytes*)
- Sumar la clave de ronda 0 (la inicial)

No se darán detalles de como se puede llegar al cifrador inverso equivalente porque dicho desarrollo es útil principalmente para arquitecturas de 32 bits y no se obtiene ninguna ventaja clara para una arquitectura de 8 bits como la utilizada para esta tesis. La razón por la que la ronda final no tiene el paso *MixColumns* es precizamente porque esto hace posible crear el cifrador inverso equivalente.

Veamos en que consiste cada uno de los pasos inversos.

3.3.1. *InvSubBytes*

La sustitución de bytes inversa puede descomponerse en una transformación *affine* inversa f^{-1} seguida de una inversión g^{-1} en $GF(2^8)$. Esto se debe a que $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$

La transformación *affine* inversa puede calcularse mediante el siguiente desarrollo. Recordemos que para este paso un byte se representa como un vector:

De la ecuación (3.4):

$$\vec{b} = M_1 \times \vec{a} + \vec{V}_1$$

Como M_1 tiene inversa M_1^{-1}, multiplicamos ambos lados de la igualdad por M_1^{-1} y recordando que la operación inversa de "+" es la misma "+" en aritmética módulo 2,

$$\vec{a} = M_1^{-1} \times \vec{b} + M_1^{-1} \times \vec{V}_1 \tag{3.28}$$

Específicamente $a = f^{-1}(b)$ puede expresarse como:

$$\begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{3.29}$$

La transformación g^{-1} puede despejarse fácilmente a partir de la definición de g dada en la ecuación (3.2):

$$b = g(a) = a^{-1}$$

Obteniendo el inverso multiplicativo del primer y último término de la ecuación tenemos:

$$(b)^{-1} = (a^{-1})^{-1} \tag{3.30}$$

por lo que:

$$a = b^{-1} = g^{-1}(b) = g(b) \tag{3.31}$$

Las operaciones se efectúan en el $GF(2^8)$ y cada elemento puede interpretarse como un polinomio de grado menor o igual a 7 tal y como se explicó en el paso *SubBytes*. Este resultado quiere decir que g es una involución ($a = g(g(a))$). Para el caso particular de $b = 0$ en el que su inverso no existe se mapea $b = 0$ a $a = 0$.

La composición de las transformaciones puede realizarse con la consulta de una tabla la cual indica la salida correspondiente para cada byte de entrada requiriendo un total de 256 bytes.

La obtención de las tablas para la transformación *InvSubBytes* se dará en el capítulo siguiente, por ahora basta decir que la transformación *affine* directa y su inversa pueden representarse como multiplicaciones y sumas entre polinomios módulo $x^8 + 1$.

3.3.2. *InvShiftRows*

El paso *InvShiftRows* recorre circularmente cada uno de los renglones del estado un cierto número de bytes (los cuales pueden consultarse en la tabla 3.1), pero en la dirección contraria que el paso *ShiftRows*, es decir ahora a la derecha.

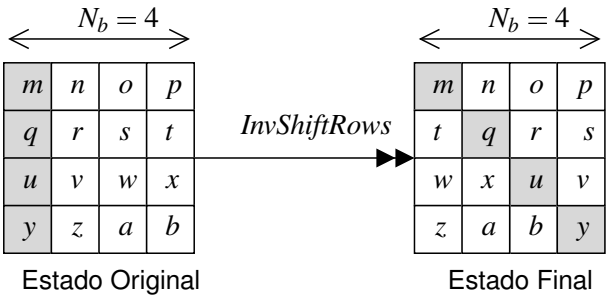

Figura 3.8: Paso *InvShiftRows* del cifrador inverso.

Otra forma equivalente de especificar los corrimientos es efectuándolos a la izquierda pero un número de columnas $N_b - C_i$ para el renglón i. Donde C_i es el desplazamiento utilizado por el paso *ShiftRows*.

3.3.3. *InvMixColumns*

Al igual que el paso *MixColumns*, la transformación *InvMixColumns* opera sobre cada una de las columnas de manera idéntica e independiente.

La columna i, formada por los elementos $[b_{0,i}\ b_{1,i}\ b_{2,i}\ b_{3,i}]^T$, o simplemente $[b_0\ b_1\ b_2\ b_3]^T$, es interpretada como un polinomio $b(x)$ de grado menor o igual a 3 sobre $GF(2^8)$, el resultado es otra columna dada por los coeficientes de $a(x)$.

De acuerdo a la ecuación 3.8 donde se especifica el paso *MixColumns* tenemos:

$$b(x) = c(x) \times a(x) \pmod{x^4 + 1} \tag{3.32}$$

Ahora, como ya se dijo, el elemento $c(x)$ sí tiene inverso y se denota como $d(x) = c^{-1}(x)$, multiplicando ambos lados de la igualdad por $d(x)$, recordando que $c(x) \times d(x) = 1$ y reordenando términos tenemos que:

$$a(x) = d(x) \times b(x) \pmod{x^4 + 1} \tag{3.33}$$

Todos los polinomios involucrados en esta ecuación están sobre $GF(2^8)$ tal como se definió en el paso *SubBytes*. El polinomio $d(x)$ está dado por:

$$d(x) = 0B \cdot x^3 + 0D \cdot x^2 + 09 \cdot x + 0E \tag{3.34}$$

La transformación puede expresarse matricialmente de la siguiente forma:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3.35}$$

El cálculo de $d(x)$ será explicado en el siguiente capítulo donde se mostrará un programa que genera $d(x)$ a partir de $c(x)$ mediante el algoritmo extendido de *Euclides*.

Los coeficientes no son tan sencillos como los usados en el paso *MixColumns* (01, 02 y 03), ahora tenemos: 0E, 0B, 0D y 09 en hexadecimal, por lo que es un poco más costoso computacionalmente calcular la salida $a(x)$ a menos que se optimice el algoritmo mediante tablas. Para la mayoría de las tarjetas inteligentes esto no es factible pero es posible simplificar la multiplicación factorizando la matriz, o equivalentemente factorizando el polinomio $d(x)$. El crédito de esta idea es para *P. Barreto*:

$$d(x) = (04 \cdot x^2 + 05) \cdot c(x) \pmod{x^4 + 1} \tag{3.36}$$

Donde $c(x)$ es el polinomio utilizado en *MixColumns*. En notación matricial tenemos que:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix} \tag{3.37}$$

Podría pensarse que estamos duplicando el trabajo porque ahora tenemos que hacer dos multiplicaciones de polinomios, sin embargo tenemos la ventaja de que podemos reutilizar la rutina que hace la multiplicación por $c(x)$, además la multiplicación por el polinomio $(04 \cdot x^2 + 05)$ puede implantarse fácilmente.

En el documento del diseño de *Rijndael* se indica cómo se puede efectuar el preprocesamiento de la columna de entrada multiplicándola por el polinomio $(04 \cdot x^2 + 05)$ eficientemente.

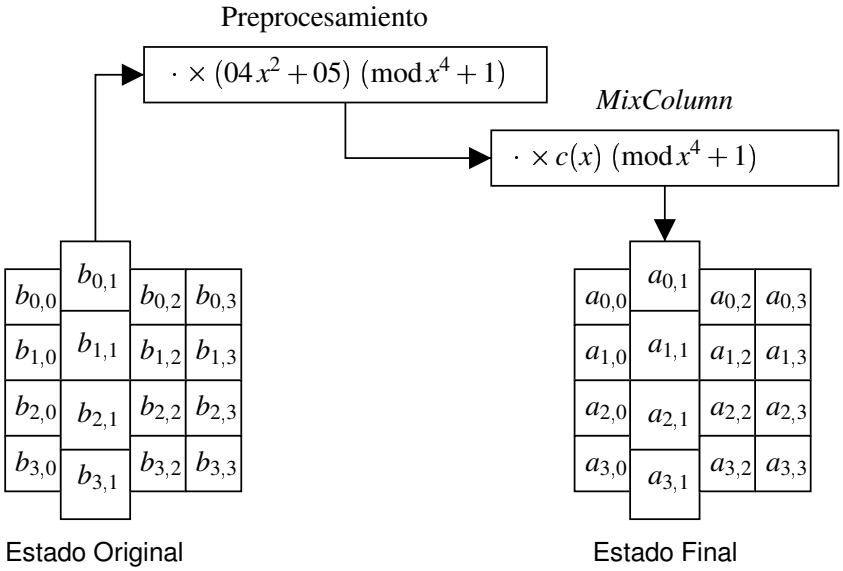

Figura 3.9: Paso *InvMixColumns* realizado mediante un preprocesamiento y la transformación *MixColumn*.

3.3.4. Planificación de Clave para el Cifrador Inverso

De acuerdo a la figura 3.7 el cifrador inverso utiliza las mismas claves de ronda que fueron utilizadas por el cifrador directo pero en el orden contrario.

Esto significa que podemos utilizar el mismo algoritmo de expansión de clave y solamente cambiar el algoritmo de selección de las claves de ronda. Desgraciadamente, como ya se mencionó, en las tarjetas inteligentes por lo regular no es conveniente mantener toda la clave expandida en memoria.

Una primera solución a este problema es derivar cada una de las claves de ronda que se necesitan a partir de la clave del cifrador de acuerdo a lo explicado en la sección 3.2.6. Esto obviamente no es práctico porque estaríamos repitiendo cálculos innecesariamente.

Una mejor solución es partir de la última clave de ronda e invertir el proceso de derivación de la siguiente clave de ronda, de forma que las obtendríamos en el orden en que las necesitamos ahora.

En la propuesta del algoritmo no se precisa como hacer este proceso inverso, pero en el libro del diseño de *Rijndael* podemos encontrar pseudocódigo que muestra como generar la clave expandida en el orden adecuado [13, p. 57].

Gracias a la forma en que presentamos las ecuaciones en una sección anterior para la generación de la clave de ronda i a partir de la clave de ronda $i-1$, fácilmente podemos ahora manipularlas para determinar el proceso inverso sin tener que meternos con el pseudocódigo.

De las ecuaciones de (3.16) a (3.27) se puede despejar fácilmente el término de la clave $k^1_{i,j}$, y se convierten en las ecuaciones (3.38) a (3.49) las cuales han sido reordenadas y se muestran a continuación. Debemos recordar que k^2 y k^1 equivalen a k^i y k^{i-1} respectivamente.

$$k^1_{0,3} = k^2_{0,3} + k^2_{0,2} \tag{3.38}$$
$$k^1_{0,2} = k^2_{0,2} + k^2_{0,1} \tag{3.39}$$
$$k^1_{0,1} = k^2_{0,1} + k^2_{0,0} \tag{3.40}$$

$$k^1_{1,3} = k^2_{1,3} + k^2_{1,2} \quad (3.41)$$
$$k^1_{1,2} = k^2_{1,2} + k^2_{1,1} \quad (3.42)$$
$$k^1_{1,1} = k^2_{1,1} + k^2_{1,0} \quad (3.43)$$
$$k^1_{2,3} = k^2_{2,3} + k^2_{2,2} \quad (3.44)$$
$$k^1_{2,2} = k^2_{2,2} + k^2_{2,1} \quad (3.45)$$
$$k^1_{2,1} = k^2_{2,1} + k^2_{2,0} \quad (3.46)$$
$$k^1_{3,3} = k^2_{3,3} + k^2_{3,2} \quad (3.47)$$
$$k^1_{3,2} = k^2_{3,2} + k^2_{3,1} \quad (3.48)$$
$$k^1_{3,1} = k^2_{3,1} + k^2_{3,0} \quad (3.49)$$

Ahora que tenemos $k^1_{i,3}$ podemos calcular $k^1_{i,0}$:

$$k^1_{0,0} = k^2_{0,0} + S_{RD}(k^1_{1,3}) + RC[i] \quad (3.50)$$
$$k^1_{1,0} = k^2_{1,0} + S_{RD}(k^1_{2,3}) \quad (3.51)$$
$$k^1_{2,0} = k^2_{2,0} + S_{RD}(k^1_{3,3}) \quad (3.52)$$
$$k^1_{3,0} = k^2_{3,0} + S_{RD}(k^1_{0,3}) \quad (3.53)$$

En donde de acuerdo a la notación empleada anteriormente, a partir de la clave de la ronda i, k^2, se produce la clave de la ronda $i-1$, k^1. Es importante notar que no hay problema si utilizamos las mismas localidades de memoria para representar los elementos de la clave de ronda anterior y la siguiente ya que en el momento en que hacemos una asignación a una casilla (i, j), el valor viejo no volverá a ser utilizado.

3.4. Algunas Consideraciones sobre el *AES*

En esta sección se tratarán brevemente múltiples aspectos sobre el *AES*, entre los que se encuentran los principios que llevaron a la elección de su estructura y componentes. También se mencionan algunos conceptos básicos de criptoanálisis, algunos tipos de criptoanálisis y como se relacionan con los principios seguidos para el diseño del *AES*.

Las operaciones empleadas para el cifrado y descifrado en el *AES* ya fueron expuestas, pero no son las únicas disponibles y el número de cifradores propuestos es muy grande. Es necesario conocer entonces la justificación para la elección de la estructura y operaciones del *AES*.

Existen diversos criterios generales para el diseño de un cifrador de bloques entre los que sobresale la seguridad, la cual implica entre otras cosas que no deben haber técnicas exitosas para quebrar completamente al cifrador más fácilmente que la búsqueda exhaustiva de la clave por *fuerza bruta*. Respecto a la seguridad surge el concepto de *margen de seguridad*, el cual es una medida de que tan lejos se encuentran los ataques a versiones con rondas reducidas respecto al número de rondas del cifrador completo. Por dar un ejemplo, si tenemos un cifrador de $n = 10$ rondas y existen ataques para una versión reducida del cifrador con $o = 8$, entonces tenemos un margen de seguridad de $k = n - o = 2$ rondas o un margen de seguridad relativo de $k/n = 0.2$.

Otra característica que se exige o desea de los cifradores es la eficiencia respecto a los recursos de memoria, superficie de silicio (implantaciones en *hardware*) y velocidad de cifrado / descifrado.

Muchas veces el cifrador es utilizado con claves distintas y puede haber una sobrecarga si el proceso de planificación de clave es lento. En estos casos es deseable que la clave pueda ser cambiada con facilidad sin incurrir en retrasos significativos. A esta propiedad se le llama *agilidad de la clave*.

Los criterios recién presentados pueden oponerse, por lo que se debe balancear la eficiencia con la seguridad. Estos criterios no son únicos y hay algunos principios muy específicos seguidos en el *AES*.

La *simplicidad* es uno de los principios más importantes que guiaron el diseño del *AES*. Podemos hablar de simplicidad en la especificación y de simplicidad en el criptoanálisis. La simplicidad en la especificación permite entender e implantar al cifrador con menos dificultades. La simplicidad en el criptoanálisis facilita el trabajo de demostrar que el cifrador es resistente a ciertos tipos de ataques. Debe quedar claro que la simplicidad no es sinónimo de debilidad, sino por el contrario puede ayudar a establecer límites en cuanto a la resistencia del cifrador frente al criptoanálisis, especialmente en este caso contra los criptoanálisis lineal y diferencial.

Una manera de lograr la simplicidad es mediante la simetría. En *Rijndael* podemos encontrar simetría en varios niveles. La simetría entre las rondas significa que las transformaciones de ronda son esencialmente las mismas, por lo que solamente tenemos que especificar una de ellas y al analizar el cifrador podemos estudiar las propiedades de una sola ronda con más precisión. También tenemos una cierta simetría cuando comparamos el cifrador con el cifrador inverso, en especial con el equivalente. La simetría dentro de una ronda consiste en que el proceso mediante el cual se transforma un conjunto de datos es similar para cada uno de ellos.

Una de las técnicas fundamentales utilizadas es la llamada *Wide Trail Strategy*, con la cual se pueden crear cifradores de clave alternante con buena resistencia a los ataques de criptoanálisis diferencial y lineal, obteniendo un buen desempeño.

De acuerdo a la *Wide Trail Strategy* la ronda es expresada mediante la composición de dos transformaciones:

- γ, una transformación no lineal cuya salida depende de algunos bits de entrada vecinos (cercanos).
- λ, una transformación lineal que proporciona difusión, es decir mezcla la información de forma que de un bit de entrada dependan muchos bits de salida. En [13, pp. 130-131] se explica qué se entiende por difusión para *Rijndael* específicamente.

La transformación γ se traduce en *Rijndael* en las llamadas *cajas S* (transformación *SubBytes*), en las cuales particularmente se buscó minimizar las probabilidades de propagación de diferencias y la correlación de entradas con salidas. Estos dos factores son los más importantes para incrementar el trabajo que requeriría un ataque utilizando criptoanálisis diferencial y lineal respectivamente.

Además la combinación de las funciones g y f hace más difícil la representación y el manejo algebraico de la transformación, la cual puede expresarse gracias a la *interpolación de Lagrange* como un polinomio de grado 254 con coeficientes en $GF(2^8)$. Si la expresión algebraica del cifrador fuera compacta se podría dar lugar a los llamados *ataques por interpolación*.

La caja S, que es el componente fundamental del paso *SubBytes*, opera sobre un byte de entrada y produce un byte de salida. Puede considerarse como 8 funciones booleanas independientes cada una produciendo uno de los bits de la salida. Sin embargo en [19] se muestra que en realidad cada una de estas funciones es equivalente respecto a transformaciones *affine*. Sin entrar en los detalles se encontró que cada una de las funciones que conforman las cajas S puede representarse como cualquiera de las otras

con un simple mapeo lineal de sus entradas. Por ejemplo, la función $b_2(x)$ que proporciona el segundo bit de la salida puede expresarse en términos de la función $b_1(x)$ que genera el primer bit mediante la transformación lineal a la entrada:

$$b_2(x) = b_1(D_{12} \cdot x) \tag{3.54}$$

donde la matriz D_{12} está dada por:

$$D_{12} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.55}$$

No está claro como sería posible aprovechar este resultado para montar un ataque a *Rijndael* y probablemente no tendrá ninguna repercusión.

La transformación λ que nos proporciona difusión es construida mediante la aplicación de dos mapeos: un mapeo θ que logra la difusión local y un mapeo π que logra dispersión de los datos llevándolos a posiciones distantes de la original.

θ se implementa como una transformación *capa de ladrillos* que opera sobre cada columna independientemente, corresponde a *MixColumns*, y puede expresarse como una multiplicación matricial. Para disminuir el uso de memoria y de acuerdo al criterio de simetría se utilizó una matriz circulante.

π realiza una transposición renglón por renglón de los bytes y corresponde a *ShiftRows*. Para incrementar la difusión los desplazamientos de cada renglón son distintos.

Las operaciones empleadas en *Rijndael* no son las más simples y tradicionales que pueden efectuarse en un microprocesador, como la aritmética modular sobre enteros (sumas y multiplicaciones) y las transposiciones parametrizadas por los bits de la clave o del estado.

Hay varias razones por las que no se utilizaron este tipo de operaciones. Por ejemplo, en el caso de las aritméticas hay diferencias importantes en su desempeño y facilidad de implantación en distintas arquitecturas, especialmente causadas por los tamaños de palabra. También para la implantación en *hardware* del algoritmo no es recomendable utilizar operaciones que pueden requerir un tiempo largo de propagación de las señales, tal como en las sumas (cuando se propaga un acarreo, aunque puede acelerarse el proceso empleando más *hardware*).

Otro problema es el orden de los bytes (*little endian*[7] o *big endian*[8]). En ciertas arquitecturas en las que el orden de los bytes sea el contrario habrá una disminución importante del desempeño al ejecutar operaciones aritméticas.

Los corrimientos dependientes de la clave o el estado tienen la desventaja de que no todas las arquitecturas los realizan con igual facilidad, por ejemplo en la arquitectura empleada para esta tesis solamente hay instrucciones para corrimientos y rotación de bits en una posición a la vez [3] (en realidad también hay una instrucción para rotar un byte 4 bits).

[7] El byte menos significativo está en la localidad de memoria inferior.

[8] El byte más significativo se encuentra en la dirección de memoria más baja.

Esta clase de operaciones también incrementa los problemas por ataques a la implantación, especialmente a algunos de los denominados ataques de *side–channel*, en los que el criptoanálisis aprovecha información obtenida de alguna señal física generada o emitida por el dispositivo durante la operación normal del algoritmo. Algunos ejemplos de los tipos de información que pueden hacer vulnerable a un cifrador son: el consumo de potencia, la radiación electromagnética, la disipación de calor y el tiempo en que completa su función.

La medición de la potencia consumida da lugar a los ataques por análisis de potencia, y se puede dividir en ataques de análisis simple de la potencia (*SPA*[9]) y en ataques de análisis diferencial de la potencia (*DPA*[10]). Sin entrar en detalles respecto al *SPA* podemos decir que la operación de los buses de datos revela información útil para el criptoanálisis. De acuerdo a la operación realizada en un momento dado y a las características físicas del diseño del microprocesador puede revelarse dos tipos de información [31]: el peso de *Hamming* del bus de datos y el número de transiciones o cambios de estado (producidas por el cambio de polarización de compuertas controladas por el bus de datos).

El *DPA* hace un análisis estadístico de las mediciones de consumo de potencia, con lo que por ejemplo es posible reducir el efecto del ruido.

Existen técnicas para proteger a los cifradores de estos tipos de ataques mediante el enmascaramiento de los operandos [21], de forma que un adversario no obtenga información sobre los datos verdaderos, pero frecuentemente incrementan el procesamiento. *Rijndael*, por sus operaciones, es más fácil de proteger contra este tipo de ataques que los otros algoritmos que fueron candidatos para ser el *AES*.

Respecto a los ataques por medición de tiempos[11], *Rijndael* en principio no tiene ningún problema porque casi todas las operaciones tienen duración constante. La única posible debilidad está en la función xtime(), empleada en la implantación del paso *MixColumns* y que equivale a una multiplicación por x, pero no es difícil evitar este problema.

[9] *Simple Power Analysis*.
[10] *Differential Power Analysis*.
[11] *Timing Attacks*.

Capítulo 4

Implantación del Algoritmo, las Librerías y las Aplicaciones

En este capítulo se tratarán los aspectos más importantes relacionados con nuestra implantación del *AES* en una tarjeta inteligente, su inclusión en un sistema operativo real, el desarrollo de los programas a nivel de *host* y todas las librerías, programas y herramientas auxiliares.

Deberá quedar claro qué herramientas fueron utilizadas, cuáles fueron modificadas para adaptarse a nuestros objetivos y cuáles fueron desarrolladas. El lector con base en la información expuesta en el capítulo podrá reproducir el trabajo de esta tesis y continuarlo.

Es importante recalcar que los programas desarrollados no solamente se enfocan a las tarjetas sino también al sistema *host*, y los principios de desarrollo y herramientas disponibles para cada caso son muy diferentes.

Se discutirán los problemas encontrados y la forma en que fueron solucionados. Cuando sea pertinente señalaremos qué criterios escogimos para llegar a dicha solución y qué soluciones alternativas ideamos. Los problemas y tareas realizadas varían enormemente e incluyen la selección de las tarjetas inteligentes, su simulación, la simulación del protocolo de comunicación, la optimización del algoritmo de cifrado, la simulación de interfaces de programación relacionadas con el lector, la extensión del sistema operativo de las tarjetas, el desarrollo de una interfaz de control para las tarjetas a nivel de *host* y la creación de una librería para manejo de ciertas estructuras algebraicas para trabajar directamente con las operaciones matemáticas de *Rijndael*.

Debemos hacer énfasis en que gracias a nuestra simulación de las tarjetas podemos trabajar indistintamente con las tarjetas físicamente o con la simulación de la arquitectura de manera transparente para las aplicaciones que se comunican con las tarjetas a través de *PC/SC*. Es decir, los programas a nivel de *host* y a nivel de la tarjeta inteligente permanecerán inalterados.

4.1. Selección de la Arquitectura de las Tarjetas Inteligentes

Uno de los pasos más importantes para desarrollar una aplicación en una tarjeta inteligente es escoger la arquitectura adecuada. Todo el proceso de desarrollo posterior será influenciado por nuestra selección. Para tomar una decisión es importante conocer qué arquitecturas y ambientes de desarrollo hay disponibles, cuáles son nuestras necesidades y establecer un criterio para evaluarlas y compararlas.

Arquitecturas Disponibles más Representativas

La mayoría de los microcontroladores de las tarjetas inteligentes están basados en el 6805 de *Motorola* o el 8051 de *Intel*.

La arquitectura de 8 bits 8051 tiene más de 20 años de haber sido introducida por *Intel* y diversas compañías como: *AMD*, *Atmel* y *Phillips* producen variantes del mismo. Posee un conjunto de instrucciones que entre sus puntos fuertes tiene que puede efectuar manipulaciones de los bits individuales eficientemente, aunque ha sido calificado de "bizarro". A pesar de que no fue diseñado especialmente para las tarjetas inteligentes, ha recibido una muy buena aceptación en esta área.

Es difícil establecer especificaciones en cuanto a la cantidad y tipo de memoria, velocidades de reloj, periféricos, etc. debido a que cada variante presenta sus cualidades propias. Gracias al tiempo que tiene en el mercado podemos encontrar muchos ambientes integrados de desarrollo y herramientas para la compilación en Internet. Hay información sobre la eficiencia del *AES* en esta arquitectura por lo que nuestros resultados podrían ser comparados fácilmente. Desgraciadamente no encontramos ninguna tarjeta con este tipo de procesador que nos permitiera modificar su sistema operativo.

La arquitectura 6805 también es de 8 bits y hay código disponible del *AES* optimizado exclusivamente para velocidad. Podría ser una buena elección ya que además hay muchas herramientas y simuladores disponibles. Aunque muchas tarjetas están basadas en este microcontrolador, no se encontró ninguna a la que pudiéramos modificar el sistema operativo e incluir programas en lenguaje ensamblador.

Existen otras arquitecturas más poderosas, como el H8/3113 de *Hitachi* que contiene un coprocesador. Puede realizar algunas operaciones de criptografía asimétrica y se pueden conseguir algunas herramientas de desarrollo. Al parecer este microcontrolador tiene restricciones de exportación. Sabemos de una implantación muy eficiente del *AES* para esta arquitectura, pero no parece sencillo portar el trabajo que pudiéramos realizar en ella a una tarjeta inteligente real.

En el mercado hay una gran variedad de tarjetas que ejecutan programas interpretados. Por razones de eficiencia no fueron considerados. Entre ellas se encuentran las *JavaCards*, y las tarjetas que utilizan *Multos*.

Existe una serie de tarjetas inteligentes con microcontroladores basados en la familia *AVR* de *Atmel*, como el *AT90SC3232C*, que parecen muy prometedores, aunque conseguir las especificaciones y *kits* de desarrollo es algo difícil. Sin embargo es posible utilizar otras tarjetas más accesibles basadas en otros microcontroladores de la familia *AVR*. Dichas tarjetas son las llamadas *funcards* y uno de sus microcontroladores más populares es el *AT90S8515*.

Características Buscadas en la Arquitectura

Para seleccionar la arquitectura adecuada se tuvieron en cuenta los siguientes criterios, en orden de importancia:

- *Capacidad de programar el microcontrolador en bajo nivel*. Con el propósito de obtener la mayor eficiencia necesitamos utilizar directamente las instrucciones en ensamblador.
- *Documentación*. Para trabajar con el microcontrolador y las herramientas de desarrollo es importante contar con documentación disponible y clara.
- *Posibilidad de probar el sistema final en una tarjeta inteligente real*. La mayoría de las pruebas de rendimiento del *AES* en microcontroladores para tarjetas inteligentes se han hecho fuera de la tarjeta, sin embargo en nuestro caso preferimos efectuar las mediciones en el sistema real dentro de la tarjeta que ejecuta los algoritmos.

- *Contar con un ambiente de desarrollo integral preferentemente libre*. Aquí incluimos a compiladores, ensambladores, analizadores de código objeto y depuradores.
- *Existencia de un buen simulador que permita medir los tiempos de ejecución*. La simulación es especialmente importante porque nos permite probar el programa fuera de la tarjeta inteligente y en caso de tropezar con un problema tenemos forma de analizar la memoria y el código para encontrarlo y resolverlo.
- *Bajo costo de los componentes de hardware*. Aquí nos referimos a las tarjetas y al programador en caso de ser necesarios.
- *Existencia de soporte técnico o una comunidad en línea.* Es importante poder recibir ayuda en caso de que se presente alguna dificultad.

Comparación de las Arquitecturas Respecto a las Características Buscadas

De entre todas las arquitecturas exploradas solamente con una podemos cargar código arbitrario programado en bajo nivel y probar el sistema resultante en una tarjeta inteligente real. Dicha arquitectura es la de los microcontroladores *AVR*. Otras opciones en las que es posible ejecutar el código de nuestra predilección están sujetas a las restricciones de un intérprete o máquina virtual por lo que el programa resultante es mucho más lento.

La arquitectura 6805 fue otro muy buen candidato ya que podemos encontrar todo un conjunto de herramientas de desarrollo. Es fácil encontrar simuladores del microcontrolador, desgraciadamente no es sencillo simular la tarjeta inteligente completa porque no se tiene un sistema operativo libre disponible, además es difícil obtener tarjetas programables.

Muchos fabricantes de tarjetas (incluyendo a *Atmel* para la serie *AT90SC*) tienen la política de no proporcionar información detallada sobre sus microcontroladores ni las herramientas abiertamente. En nuestra opinión esta es una mala práctica, pero lo hacen con el propósito de dificultar la ingeniería inversa.

Por esta clase de situaciones muchas de las pruebas de la eficiencia del *AES* en tarjetas inteligentes han sido realizadas no en una tarjeta inteligente real sino en simuladores del microcontrolador, o en el mismo microcontrolador pero fuera de la tarjeta, por lo que varios aspectos quedan sin ser considerados como la transmisión de los datos y el proceso con el cual el sistema operativo interpreta los comandos.

Respecto a las herramientas, los microcontroladores 8051, 6805 y *AVR* tienen ambientes completos de desarrollo libres o de *shareware*, en contraste con el del microcontrolador de *Hitachi* que al parecer se tiene que comprar. En cuanto al soporte no pensamos que pueda haber ningún problema con ninguno de ellos.

El Microcontrolador *AT90S8515*

Después de evaluar las alternativas seleccionamos a los microcontroladores de la familia *AVR* trabajando en las tarjetas *funcards* por ser la única opción que cubre todas las características que estamos buscando. En particular se escogió al microcontrolador *AT90S8515* debido a que es el más fácil de conseguir. De haber tenido la oportunidad hubiéramos utilizado una arquitectura de la familia *AT90SC* pero la información y herramientas solamente son suministradas a empresas certificadas.

Ahora describiremos las características principales del microcontrolador *AT90S8515*. Para encontrar información más detallada se sugiere consultar [2]. Se trata de un microcontrolador *RISC* en el que la

mayoría de las instrucciones tardan en ejecutarse un periodo del reloj externo. Tiene un conjunto de instrucciones muy bien planeado, entre sus virtudes está que los desarrolladores trabajaron auxiliados por programadores y diseñadores de compiladores expertos en lenguaje *C*, por lo que se dice que la compilación de código de este lenguaje es especialmente eficiente con respecto al código en ensamblador y a otros microcontroladores.

Uno de los datos sorprendentes es que tiene 32 registros de propósito general, 6 de los cuales pueden funcionar por parejas como índices. Todos los registros están conectados a la *ALU* directamente. Es posible realizar operaciones con dos registros y almacenar el resultado en un solo ciclo de reloj.

Tiene arquitectura de *Harvard*, esto quiere decir que se tiene un espacio memoria de datos y un espacio de memoria de programa separados. La memoria de programa está formada por 4096 palabras de 2 bytes y la mayoría de las instrucciones ocupan solamente una palabra. Se trata de una memoria *Flash* por lo que podemos borrarla y rescribirla con las señales eléctricas adecuadas.

En cuanto a la memoria de datos se tiene una *SRAM* de 512 bytes y el espacio de memoria permite acceder a los registros de propósito general, a los registros de entradas y salidas, y a la SRAM en sí. El *bus* de datos es de 8 bits y el de direcciones de 16 (permite direccionar hasta 64 KBytes en una memoria *SRAM* externa).

Tenemos una memoria *EEPROM* denominada *EEPROM interna* de 512 bytes. Para accederla se utilizan métodos distintos que para la memoria de datos. El tiempo de escritura es mucho más lento que el de la *SRAM* y está en el orden de los milisegundos (es variable y de entre 2.5 y 4 ms). El microcontrolador da facilidades para agregar una memoria *SRAM* externa.

Los modos de direccionamiento soportados son los siguientes:

- *Directo, con un solo registro*. La instrucción indica el número del registro a utilizar del cual se toma su valor tal cual.
- *Directo, con dos registros*. Dos registros son los operandos y el resultado es almacenado de nuevo en uno de ellos.
- *Directo a los registros de entrada / salida*. Las instrucciones incluyen el número del registro de entrada / salida, el cual va de 0 a 64.
- *Directo de datos*. Se especifica en la instrucción la dirección exacta en memoria de datos del byte que será afectado. Estas instrucciones ocupan 2 palabras y son más lentas.
- *Indirecto de datos*. Los registros X, Y o Z de 16 bits, formados por los pares de registros r27:r26, r29:r28 y r31:r30, son utilizados como apuntadores indicándonos la dirección del operando.
- *Indirecto de datos con desplazamiento*. Este modo de direccionamiento es sumamente poderoso y es similar al modo indirecto de datos pero se añade un desplazamiento fijo de 6 bits y solamente Y y Z pueden funcionar como índices.
- *Direccionamiento de constantes en memoria de programa*. Es posible almacenar datos en la memoria de programa, pero para direccionarlos se requieren instrucciones especiales porque no se encuentran en el espacio de datos.
- *Direccionamiento de programa directo*. Utilizando el registro Z podemos saltar a posiciones arbitrarias.

- *Direccionamiento de programa relativo*. Se especifica un *offset* constante que modifica el contador de programa (saltos relativos).

Además de los modos de direccionamiento anteriores se tienen instrucciones de direccionamiento de datos indirecto que efectúan postincrementos o predecrementos de los apuntadores. Estas instrucciones tienen cierta semejanza con las encontradas en Procesadores Digitales de Señales (*DSPs*[1]).

Ventajas de la Arquitectura Seleccionada

Ahora examinaremos las cualidades principales de este microcontrolador. Primero que nada el fabricante proporciona un ambiente de desarrollo integrado muy práctico y disponible sin costo. Se incluye un simulador que considera también la presencia de una memoria *SRAM* externa y tenemos los manuales correspondientes. Hay una comunidad de desarrolladores muy grande tanto en *Linux* como en *Windows*, la cual ha creado herramientas adicionales libres de muy alta calidad como lo es el compilador *gcc* para los microcontroladores *AVR*, llamado *avr–gcc*. En caso de experimentar algún problema hay listas de discusión donde podemos solicitar ayuda.

Las tarjetas inteligentes con este tipo de circuito integrado son relativamente baratas (de $6 a $10 USD). Es difícil conseguirlas en México y EUA, pero las venden en varias tiendas electrónicas de Europa y pueden enviarlas por correo. Un sistema operativo libre para estas tarjetas y que está programado de forma especialmente clara puede bajarse de Internet.

Hay un simulador en desarrollo y es posible depurar los programas mediante *gdb*, tarea que es prácticamente imposible una vez que los programas están corriendo dentro de la tarjeta.

El equipo para programar la memoria *Flash* de los microcontroladores (para cargar nuestros programas) es de precio reducido (unos $50 USD) y las tarjetas pueden ser reprogramadas prácticamente tantas veces como sea necesario.

El microcontrolador es relativamente eficiente ya que alcanza cerca de 1 *MIPS* por cada *MHz* del reloj externo. Aprender el lenguaje ensamblador no es tan difícil y es posible mezclar la programación en bajo nivel con la programación en otros lenguajes de alto nivel (el soporte para *C++* es muy limitado por ahora, pero es posible que en el futuro mejore).

La cantidad de registros hace posible disminuir el uso de la memoria *SRAM* por lo que los programas pueden funcionar mucho más rápido y permiten utilizar este valioso recurso para otros fines.

4.2. Simulación del Microcontrolador de las Tarjetas Inteligentes

Una vez que un programa está cargado y funcionando dentro de una tarjeta inteligente, no hay ninguna forma sencilla de conocer qué operaciones está realizando, cuál es el valor de los registros, ni cuál es el contenido de la memoria. Este problema se complica si utilizamos *hardware* de generación de números aleatorios o pseudoaleatorios ya que aunque apliquemos las mismas entradas el estado interno será distinto.

La arquitectura queda escondida por los mecanismos de seguridad propios de la tarjeta y por el hecho de que la única vía razonable de comunicación con el exterior es el contacto del puerto serial *half-duplex*.

Como es conocido, difícilmente podemos tener un programa no trivial libre de errores, y aún más es demostrarlo. Lo que sí podemos hacer es reducir significativamente el número de errores hasta que tengamos la confianza de que no ocurrirán fallos graves que ocasionen pérdidas o violaciones a la seguridad. Para ello son útiles algunos tipos de herramientas.

[1] *Digital Signal Processors*.

Hay dos formas principales de conocer el estado interno de una tarjeta inteligente:

- *Mediante un emulador*. Consiste en un *hardware* especial que incluye un microcontrolador junto con cualquier dispositivo auxiliar requerido (como memorias *EEPROM* o *SRAM* externas). Mediante este *hardware* especial tenemos acceso completo al microcontrolador, podemos conocer completamente el contenido de los *buses* y de la memoria. También podemos controlar la ejecución del programa. Además tenemos una buena certeza de que eléctricamente el dispositivo emulado se comporta de la misma forma, por lo que no debería aparecer ningún problema si intercambiamos al emulador por el microcontrolador real. En cuanto a los tiempos, tenemos una gran precisión respecto a la velocidad de respuesta y formas de las señales que produciría el verdadero microcontrolador.

  Existe una serie de emuladores disponibles para la familia *AVR* entre los que se encuentra y sobresale, por ser la versión más nueva, el *ICE 200* (*In-Cicruit Emulator*). Se trata de una tarjeta (no de una tarjeta inteligente) la cual se conecta con la computadora mediante una interfaz serial y es posible depurar los programas que se ejecutan en el microcontrolador. Para mayor información ver [5].

- *Mediante un simulador*. Si no nos interesa tener tanta precisión, en lugar de emular el microcontrolador podemos simularlo en *software*. Bajo ciertas condicione tendremos un comportamiento apegado al real. En cuanto a las consideraciones eléctricas no nos servirá de nada, pero sí desde el punto de vista de la programación.

  Las simulaciones pueden tomar más o menos propiedades de la arquitectura real, así que es importante seleccionar un buen simulador que verdaderamente nos ayude a detectar problemas antes de que se presenten en la implantación real.

  Para los microcontroladores *AVR* podemos escoger uno de entre los tres disponibles:

  - *El depurador de AVRStudio*. *AVRStudio* es una herramienta proporcionada por *ATMEL* que se compone de un ensamblador y un depurador. Puede utilizarse conjuntamente con un compilador externo, el cual es *avr–gcc*. En realidad no nos agradó este simulador, especialmente porque no incluye un compilador y solamente funciona en *Windows*, por lo que para utilizarlo junto con *avr–gcc* debemos instalar también la versión de *avr–gcc* de *Windows*.
  - *IAR C–SPY Simulator*. Esta herramienta forma parte del ambiente de desarrollo integrado denominado *EWA90* (*Embedded Workbench for the ATMEL AVR AT90S*) [4]. Se trata de un simulador muy poderoso, permite seguir la ejecución de los programas tanto en ensamblador como a nivel de sentencia de lenguaje C. Entre sus virtudes está que puede ayudarnos a medir el desempeño de los programas, simula las interrupciones y las entradas y salidas con ayuda de un lenguaje de macros.
  - *Simulavr*. Se trata de un simulador libre (*GPL*) capaz de simular varios de los dispositivos *AT90S* creado por *Theodore A. Roth* y que se encuentra todavía en una etapa inicial pero en constante evolución. A pesar de que no es tan sofisticado como el simulador de *IAR*, tiene todo lo que necesitamos para los propósitos de esta tesis, con la excepción de la memoria *EEPROM* externa, pero este no es un componente esencial y podemos omitirlo. *Simulavr* permite conocer los datos de los registros internos y de la memoria *SRAM* a través de programas externos, los cuales se comunican con él mediante ciertas interfaces.

4.2.1. Selección del Simulador: *Simulavr*

Utilizar un emulador para este trabajo de tesis no era conveniente por que este equipo especializado tiene un costo alto (al rededor de $800 USD), así que debíamos decidirnos por uno de los tres simuladores. El simulador de *AVRStudio* nos pareció algo primitivo y difícilmente podríamos hacer que otras herramientas que necesitamos para simular el protocolo *T=0* trabajaran con él. A pesar de que en términos generales el *IAR C-SPY Simulator* es superior técnicamente a *simulavr*, presenta algunas desventajas. En primer lugar se trata de un programa propietario y funciona solamente en *Windows*, así que no podríamos utilizarlo junto con los lectores de tarjetas, que como se verá más adelante, sólo funcionan en *Linux*. Existe una versión de demostración útil por 30 días pero no es posible renovar este plazo. Por otra parte, ajustar nuestros programas a las interfaces que proporciona este simulador es una tarea que parece difícil y en dado caso dichos programas tendrían que funcionar también en *Windows*. Además en la presente tesis se dio preferencia a las herramientas libres así que decidimos mejor utilizar *simulavr*.

Simulavr ofrece varias ventajas significativas. Entre ellas está que por ser libre podemos mejorar el código o adaptarlo a nuestras necesidades, cosa que efectivamente realizamos. *Simulavr* puede actuar como un objetivo remoto para *gdb*, depurador muy poderoso. Existe un protocolo de comunicación con el cual *simulavr* puede comunicarse con un programa externo que funciona como *display* mostrando el valor de los registros y el contenido de la memoria entre otras cosas. Para obtener el *software* y conocer más del proyecto, recomendamos visitar la página oficial en [36].

Instalación de *Simulavr*

A continuación se describe como instalar *simulavr*. Hay que notar que la última distribución oficial de *simulavr*, que es la 0.1.1, no nos sirve de nada ya que tiene ciertos errores que hacen imposible la simulación de *SOSSE*, el sistema operativo empleado por las tarjetas, y que veremos más adelante. A causa de que es muy común encontrar problemas al tratar de compilar esta herramienta presentaremos la forma de hacerlo paso por paso tal y como lo hicimos en nuestro sistema.

Debido a que *simulavr* está siendo mejorado constantemente es conveniente bajar la última versión disponible directamente del *CVS* en código fuente.

```
[eban@power tesis]$ cvs -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/simulavr login
```

Nos preguntará el *login* a lo cual responderemos simplemente con *enter*.

```
[eban@power tesis]$ cvs -z3 -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/simulavr
  co simulavr
```

Con lo cual obtenemos un directorio llamado `simulavr` con el código en el subdirectorio `src`.

Ahora debemos generar algunos archivos necesarios para la compilación. Hay que notar que las versiones de los siguientes programas no son imprescindibles y podemos utilizar otras siempre que no sean muy viejas.

```
[eban@power tesis]$ cd simulavr
[eban@power simulavr]$ aclocal-1.6 -I config
[eban@power simulavr]$ autoheader-2.53
[eban@power simulavr]$ autoconf-2.53
[eban@power simulavr]$ automake-1.6 --foreign --add-missing --copy
```

Continuemos con el proceso estándar de compilación. Estamos proporcionando ciertos parámetros para que el simulador funcione más eficientemente. Se deshabilitan las pruebas porque primero que nada a estas alturas no tenemos el compilador adecuado y porque algunas de ellas no están diseñadas para trabajar junto con la librería *libc* que vamos a utilizar.

```
[eban@power simulavr]$ my_CFLAGS='-g -O2 -Wall -fomit-frame-pointer
  -finline-functions -march=i686 -mcpu=i686'
[eban@power simulavr]$ CFLAGS=$my_CFLAGS ./configure --disable-tests
[eban@power simulavr]$ make
```

En este punto ya está compilado *simulavr* junto con algunas herramientas, pero debemos instalarlo para que todos los usuarios puedan acceder a él.

```
[eban@power simulavr]$ su root
[root@power simulavr]# make install
```

Lo siguiente facilita el uso del simulador para el microcontrolador *AT90S8515*, permitiéndonos utilizar el simulador sin tener que especificar de qué dispositivo se trata mediante un parámetro en la línea de comandos.

```
[root@power simulavr]# cd /usr/local/bin/
[root@power bin]# ln -s simulavr at90s8515
```

Si el directorio `/usr/local/bin/` está en el *path*, entonces estamos listos para comenzar a utilizar *simulavr*.

El *Display* de *Simulavr*

Gracias a que se tiene un protocolo abierto que establece la forma en que el simulador se comunica con un programa externo, que se encarga de mostrar el estado interno del microcontrolador, estamos en condiciones para escoger o diseñar el tipo de *display* que más nos convenga. Un ejemplo de esto es el *display* creado por *Carsten Beth*, llamado *simulavr–vcd* y que toma la información de las señales o el estado de los registros y la memoria *SRAM* y produce un archivo con formato *VCD*. Este formato puede ser mostrado de forma muy elegante y conveniente por algún programa estándar que muestre archivos *VCD* tal como *gtkwave* [10]. *simulavr–vcd* está incluido en la distribución de *simulavr* y se compila por omisión. El único posible problema es que requiere que generemos un archivo de configuración, pero podemos encontrar un ejemplo de uno de estos archivos en `simulavr/src/disp-vcd/vcd.cfg`.

Veamos una forma de llamar al *display* tradicional de *simulavr*, *simulavr–disp*:

```
[eban@power src]$ simulavr -d at90s8515 -e eedata.bin sosse.bin -P simulavr-disp
Simulating a at90s8515 device.

MESSAGE: file decoder.c: line 3391: generating opcode lookup_table
writing 0x00 to 0x0037
...
```

En este ejemplo le indicamos al simulador que utilice la información del archivo `eedata.bin` para la memoria *EEPROM* interna y la de `sosse.bin` para la memoria de programa. Esta es la forma estándar de simular el *COS*. Se utiliza el coproceso de despliegue *simulavr–disp* y gráficamente podemos apreciarlo en la figura 4.1.

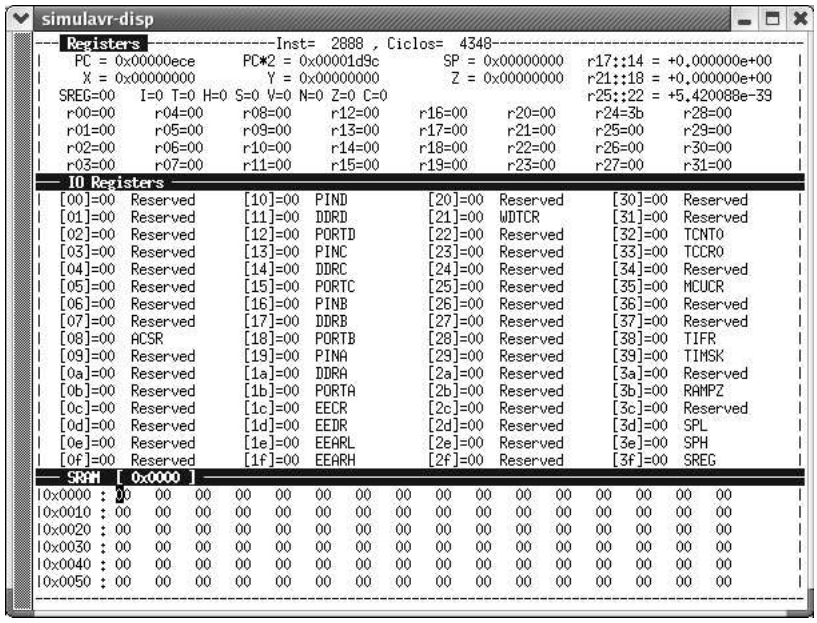

Figura 4.1: Imagen de *simulavr–disp* mostrando la ejecución de *SOSSE*

4.2.2. Crónica de Algunos Problemas Encontrados en el Simulador y sus Soluciones

Cuando se comenzó a utilizar *simulavr* se trabajó con la última versión oficial, la 0.1.1, y la tarea más importante que debíamos hacer con él era simular el sistema operativo *SOSSE*. Debido al tamaño y complejidad de *SOSSE*, teníamos cierta seguridad de que si se lograba simular correctamente entonces podríamos con bastante confianza simular cualquier otro programa.

El primer problema que apareció estaba relacionado conque el simulador indicaba que estábamos accediendo a localidades de la memoria de programa inválidas. No era claro en donde se originaba el error, podía encontrarse en el sistema operativo mismo, en la compilación o en el simulador.

Después de muchas horas de rastrear el error, y que involucró incluso la verificación del código de máquina generado por el compilador, se determinó que el error se encontraba en el simulador. Por ello se reportó el error a la lista de discusión de *simulavr*. El programador de *simulavr*, *Theodore Roth*, contestó a la brevedad y después de explicarle con mayor detalle el problema nos mando un *parche* para que comprobáramos que efectivamente estaba solucionado el problema. Le confirmamos que ya estaba corregido y lo aplicó a la versión del código de desarrollo del *cvs*. Todo este proceso fue de un día para otro. Es difícil pensar que nos puedan dar una respuesta tan rápida con la mayoría de los programas propietarios.

El segundo problema se debió a que de acuerdo a la documentación de *simulavr*, el soporte a la memoria *EEPROM* interna ya estaba listo. Sin embargo la simulación de *SOSSE* nos llevaba a un estado interno de error porque se detectaba un fallo en los datos leídos de la *EEPROM*, donde entre otras cosas se encuentra la *respuesta a reset* (*ATR*).

Después de analizar el código fuente de *simulavr*, quedó claro que si bien prácticamente todas las operaciones sobre la *EEPROM* interna estaban listas, faltaba una muy importante que era la que se encargaba de cargar los datos de un archivo a dicha memoria. También reportamos este mensaje a la lista

de correos y en muy poco tiempo, algo así como un día, estaba lista una nueva versión en el *cvs* con el soporte para la *EEPROM* completo.

Un tercer problema fue que *SOSSE* por omisión hace uso de una memoria *EEPROM* externa, esto es, la que se encuentra fuera del microcontrolador. El mecanismo de acceso a esta memoria es muy diferente al del acceso a la *EEPROM* interna y no le corresponde al simulador soportar esta memoria, eso le correspondería a un simulador de periféricos. Específicamente para acceder a la memoria externa se utiliza un protocolo serial denominado *I2C*. Se comentó esta contrariedad en la lista de discusión, pero la respuesta fue que por el momento no era factible resolver este problema, ya que hacerlo de forma correcta requeriría crear una interfaz con *hardware* externo de propósito general y esa es una tarea difícil a la que todavía no piensan dedicarse.

Afortunadamente pudimos librar este problema compilando el *COS* de forma que no utilizara la memoria *EEPROM* externa. Esto se consigue simplemente modificando el parámetro adecuado en el archivo de configuración. El inconveniente de hacer esto es que estamos limitando la memoria disponible para almacenar el sistema de archivos.

Adaptación de *Simulavr*

Para que *simulavr* nos fuera más provechoso fue necesario hacerle algunas adaptaciones. Con el propósito de realizar comparaciones para esta tesis, es deseable tener información sobre el número de ciclos de reloj que toma cada función y el número de instrucciones ejecutadas (esto último para saber que tan cerca estamos de alcanzar 1 MIPS por cada MHz). Las modificaciones que le hicimos a *simulavr* básicamente se dieron en cuatro direcciones:

- *Incluir una variable interna con el número de instrucciones ejecutadas*. Para ello se modificó la estructura *AVRcore* que contiene el estado del microcontrolador y se encuentra definida en: `simulavr/src/avrcore.h`. Se añadieron las funciones para actualizar y acceder adecuadamente a la nueva variable en `simulavr/src/avrcore.c`.

- *Expandir el protocolo utilizado con el coproceso de despliegue*. *Simulavr* puede utilizar un coproceso externo para desplegar el valor de los registros y la memoria. Se transmite la información necesaria mediante un protocolo, pero no se incluye la información sobre el número de instrucciones ejecutadas y los ciclos del reloj, así que modificamos el código de `simulavr/src/display.c` y `simulavr/src/display.h` para que también se transmitan los datos que necesitamos.

- *Modificar el programa de despliegue simulavr–disp*. El programa de despliegue, *simulavr–disp*, tal cual no muestra el número de ciclos de reloj que han pasado ni el número de instrucciones ejecutadas, por lo que se extendió `simulavr/src/disp/disp.c` para que se acepte la nueva información del protocolo y que la muestre.

- *Agregar nueva funcionalidad al programa de despliegue*. Para analizar más detalladamente nuestra implantación del algoritmo era deseable conocer información tal como: cuánto tiempo se gasta en cada instrucción y cuántas veces se ejecuta cada una. Con esta información podemos conocer qué partes de nuestro programa son más lentas o qué código es más utilizado. De esta forma sabremos que transformaciones valen la pena enfocarnos en optimizar. Esta nueva funcionalidad se agregó a *simulavr–disp* y el resultado puede consultarse en un archivo de salida.

Con las modificaciones hechas se generó un *parche* el cual debe aplicarse al código de *simulavr* después de bajarlo del *cvs* y antes de compilarlo:

```
...
[eban@power tesis]$ cvs -z3 -d:pserver:anoncvs@subversions.gnu.org:/cvsroot/simulavr
  co simulavr
[eban@power tesis]$ patch -p0 < simulavr_cvs_parche
...
```

4.3. Implantación en Lenguaje *C* del *AES* para las Tarjetas

Antes de tratar de desarrollar el *AES* en ensamblador buscamos comenzar a familiarizarnos con el algoritmo desde el punto de vista de la programación como una primera aproximación a lo que será la implantación optimizada del *AES*. Por ahora no se empleará ensamblador ya que no es conveniente hacerlo hasta tener bien clara la forma en que las operaciones transforman el estado y se efectúa la planificación de clave. En vez de ensamblador se utilizó *lenguaje C* debido a que es un lenguaje sumamente poderoso (nos da un nivel de control sobre la arquitectura cercano al que obtendríamos utilizando ensamblador) y a que el microcontrolador utilizado está diseñado especialmente para ofrecer una alta eficiencia al compilar desde lenguaje *C*.

Mediante la implantación de *Rijndael* en lenguaje *C* conoceremos más a fondo la arquitectura y pondremos en práctica los conocimientos adquiridos al estudiar la especificación del algoritmo, además tendremos un punto de comparación para cuando lo programemos en ensamblador. Estudiaremos las herramientas que nos servirán no solamente para la compilación de programas en lenguaje *C*, sino también para manipular el código objeto en general, cualquiera que haya sido el lenguaje original.

4.3.1. El Conjunto de Herramientas de Desarrollo

Existen fundamentalmente dos ambientes de desarrollo. El primero está basado en el compilador de *IAR*, el cual de acuerdo a algunas fuentes es sumamente eficiente, y el segundo tiene como núcleo el compilador *avr–gcc*[2], el cual tiene un muy buen desempeño, aunque al parecer no tanto como el de *IAR*.

Como ya se ha mencionado se optó por *software* libre para el trabajo de esta tesis, así que salvo algunas ocasiones, siempre se utilizó *avr–gcc*. Obviamente el compilador no es la única herramienta que necesitamos, se requieren programas como ensambladores y ligadores, entre otros. Afortunadamente contamos con todo un conjunto de herramientas libres que cubrieron todas nuestras necesidades.

Cabe señalar que para este microcontrolador el desarrollo de *software* en lenguaje *C* es esencial. La mayoría de los programadores lo prefieren sobre ensamblador o cualquier otro lenguaje. Aunque programemos en ensamblador es muy importante que conozcamos sobre el proceso de compilación en lenguaje *C* porque tarde o temprano vamos a tener que interactuar con rutinas y datos definidos en este lenguaje. Para hacerlo correctamente debemos saber las convenciones en cuanto al paso de parámetros, manejo de la pila, etc., además de que hay librerías escritas en lenguaje *C* que nos pueden ser de utilidad.

También es posible programar la arquitectura con el lenguaje *C++*, pero el compilador para dicho lenguaje está en desarrollo y por ahora no ofrece todas las características del lenguaje *C++* estándar. Es natural que no tengamos toda la funcionalidad de *C++* ya que con un microcontrolador como el que estamos utilizando no podemos darnos el lujo de desperdiciar recursos a causa de la sobrecarga en que se incurre normalmente al emplear este lenguaje.

[2] *AVRStudio* carece de compilador y requiere apoyarse en uno externo.

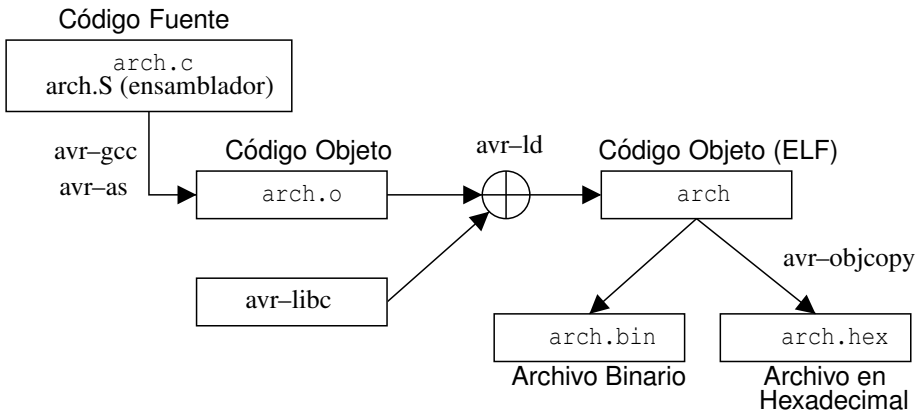

Figura 4.2: Proceso de compilación con la cadena de herramientas de desarrollo de *GNU* para los microcontroladores *AVR*.

Los tres componentes que forman el ambiente de desarrollo *GNU* para los microcontroladores *AVR* (también conocido como la cadena de herramientas de *GNU*) son los siguientes:

- *Binutils*. Paquete que contiene las utilerías con las que podemos manipular y generar código objeto. Los elementos más importantes de este paquete son:
  - `avr-as`. El ensamblador para los microcontroladores de la familia *AVR*.
  - `avr-ld`. El ligador.
  - `avr-objcopy`. Traduce archivos de código objeto de un tipo a otro (*Elf* a binario, etc.).
  - `avr-objdump`. Muestra información sobre los archivos de código objeto.
- *Avr–gcc*. Se trata del compilador *gcc* (colección de compiladores de *GNU*) trabajando como compilador cruzado con el objetivo *AVR*. El código fuente incluye al compilador de lenguaje *C* y al de *C++*, aunque por lo común solamente lo compilamos para producir al compilador de *C*. Los programas principales generados son:
  - `avr-cpp`. El preprocesador de lenguaje *C*.
  - `avr-gcc`. El compilador.
- *Avr–libc*. Librería estándar de lenguaje *C* que contiene muchas funciones y definiciones útiles en la mayoría de los programas, por ejemplo: para acceder a la memoria *EEPROM*, para acceder a los registros de entrada y salida por su nombre, para manipular información en memoria de programa, para trabajar con cadenas, funciones matemáticas, etc.
- *Avr–gdb*. Es el potente depurador de *GNU*, en este caso está compilado para aceptar un objetivo remoto el cual es el simulador *simulavr*.

Instalación del Ambiente de Desarrollo *GNU*

Para poder experimentar con los algoritmos implantados es necesario contar con el ambiente de desarrollo correctamente instalado. El procedimiento de instalación no es del todo estándar, especialmente

para *avr–gcc*, así que lo describiremos brevemente con las mismas versiones que utilizamos. El orden de la instalación de las herramientas sí importa, por lo que debe seguirse el que aquí presentamos.

Se prefirió trabajar con las versiones estables más recientes y se generó todo a partir del código fuente. El primer paso es conseguir las herramientas, las cuales encontramos en:

- *Binutils:* podemos bajarlo del *mirror* de *GNU* en la *UNAM*, `ftp://www.gnu.unam.mx/pub/gnu/software/binutils/`, utilizamos la última versión disponible que es `binutils-2.13.2.1.tar.gz`.
- *Gcc:* también está en el *mirror* de *GNU* en la *UNAM*, `ftp://www.gnu.unam.mx/pub/gnu/software/gcc/`, utilizamos la última versión disponible que es `gcc3.2.2` y que fue liberada el 5 de Febrero de 2003, específicamente hay que bajar el archivo `gcc-3.2.2.tar.gz`.
- *Avr–libc:* el sitio oficial de esta librería es: `http://savannah.nongnu.org/projects/avr-libc/` y podemos encontrar la última versión en: `http://savannah.nongnu.org/download/avr-libc/`, la cual en este momento es `avr-libc-20020203.tar.gz`.
- *Avr–gdb:* podemos ubicar la más reciente versión en `ftp://www.gnu.unam.mx/pub/gnu/software/gdb/` que data de mediados de diciembre del 2002 y viene empaquetada en el archivo `gdb-5.3.tar.gz`.

Ahora antes que nada instalamos *binutils*, el procedimiento es sencillo:

```
[eban@power tesis]$ tar -xzvf binutils-2.13.2.1.tar.gz
[eban@power tesis]$ cd binutils-2.13.2.1
[eban@power binutils-2.13.2.1]$ ./configure --target=avr
[eban@power binutils-2.13.2.1]$ make
[eban@power binutils-2.13.2.1]$ su root
[root@power binutils-2.13.2.1]# make install
```

La siguiente herramienta por instalar es *avr–gcc*:

```
[eban@power tesis]$ tar -xzvf gcc-3.2.2.tar.gz
[eban@power tesis]$ cd gcc-3.2.2
[eban@power gcc-3.2.2]$ cd gcc
[eban@power gcc]$ ./configure --program-prefix="avr-" --target=avr --enable-languages="c"
```

Apareció una dificultad con el proceso de compilación, que involucró el uso de una librería llamada *libiberty*. Intentamos generar dicha librería mediante el código de *libiberty* que se incluye en el directorio `gcc-3.2.2`, pero al parecer hay algún problema en el código, así que tuvimos que reutilizar dicha librería del paquete *binutils*. Este paso no es algo estándar, sino un fallo.

```
[eban@power gcc]$ cp ../../binutils-2.13.2.1/libiberty/libiberty.a ../libiberty
```

Ahora sí podemos compilar correctamente el código, pero hay problemas con la instalación debido a que no se crean los directorios adecuados, por lo que hay que crearlos manualmente:

```
[eban@power gcc]$ make
[eban@power gcc]$ su root
[root@power gcc]# mkdir /usr/local/lib/gcc-lib
[root@power gcc]# mkdir /usr/local/lib/gcc-lib/avr
[root@power gcc]# mkdir /usr/local/lib/gcc-lib/avr/3.2.2
[root@power gcc]# make install
```

Para trabajar con el compilador instalaremos *avr–libc*. Contamos con unos *scripts* muy convenientes:

```
[eban@power tesis]$ tar -xzvf avr-libc-20020203.tar.gz
[eban@power tesis]$ cd avr-libc-20020203
[eban@power avr-libc-20020203]$ ./doc
[eban@power avr-libc-20020203]$ ./doconf
[eban@power avr-libc-20020203]$ ./domake
[eban@power avr-libc-20020203]$ su root
[root@power avr-libc-20020203]# ./domake install
```

Por último tenemos al depurador *gdb* (*avr–gdb*), el cual obviamente no funciona dentro del microcontrolador, pero puede trabajar con un objetivo remoto: *simulavr*. Tecleamos los comandos:

```
[eban@power tesis]$ tar -xzvf gdb-5.3.tar.gz
[eban@power tesis]$ cd gdb-5.3
[eban@power gdb-5.3]$ ./configure --target=avr --program-prefix="avr-"
[eban@power gdb-5.3]$ make
[eban@power gdb-5.3]$ su root
[eban@power gdb-5.3]$ make install
```

Ahora debemos tener instalado todo el ambiente de desarrollo necesario para comenzar a programar y experimentar con nuestros microcontroladores *AVR*.

4.3.2. Restricciones Impuestas a la Implantación del Algoritmo para su Diseño

Para guiarnos en el diseño del algoritmo, previamente a su codificación, debemos establecer algunos criterios o de lo contrario podríamos llegar a una implantación que no nos sería de utilidad. El requerimiento fundamental que debe cumplir la implantación es que debe poder funcionar en la arquitectura seleccionada. No importa que teóricamente lleguemos a la implantación más rápida o más pequeña en tamaño de código si esta implantación no funciona en la arquitectura, por ejemplo, por ser muy grande u ocupar demasiada memoria *SRAM*.

Es también muy deseable que el algoritmo pueda convivir con el sistema operativo, por lo que debemos considerar los recursos que ambos ocupan. En este sentido la limitación más fuerte es la cantidad de memoria de programa disponible. Aprovechando una de las utilerías disponibles sabemos que *SOSSE* prácticamente ocupa toda la memoria de programa:

```
[eban@power src]$ avr-size sosse
   text    data     bss     dec     hex filename
   7796       0      36    7832    1e98 sosse
```

Sería muy difícil llegar a una implantación que cupiera en los $8192 - 7796 = 396$ bytes restantes de memoria de programa, especialmente si recordamos que cada instrucción ocupa por lo menos 2 bytes. Sin embargo, es posible reducir el tamaño de *SOSSE* eliminando aquellas funciones que no son necesarias.

En cuanto a la memoria de *SRAM*, afortunadamente el *COS* hace un uso muy eficiente de ella y la mayoría está disponible. En realidad no es tan fácil determinar la cantidad de memoria libre porque su uso es dinámico y las localidades pueden estar desocupadas en cierto momento y en otro no.

Pensando en nuestra implantación en ensamblador, conviene seguir las recomendaciones de cómo implantar el algoritmo en arquitecturas de 8 bits que se sugieren en [13].

Es importante remarcar que los criterios señalados no nos llevan a una implantación con máxima eficiencia bajo ningún criterio, sino que su objetivo es que con esta implantación establezcamos una base para determinar la factibilidad de la implantación del algoritmo en esta arquitectura y para tener una idea más precisa de los recursos que ocupa.

4.3.3. Implantación Preeliminar del Algoritmo

En nuestra implantación en lenguaje *C* se programó cada una de las transformaciones de forma independiente, aunque sabemos que el algoritmo puede optimizarse si aprovechamos algunas de sus propiedades y mezclamos los pasos. Lo primero que podemos notar en el código producido es que hacemos una transposición de los datos y la clave al inicio y al final, esto se debe a que pareció más sencillo y eficiente manejar los datos en este orden. La disposición de los bytes en la especificación tiende a favorecer a arquitecturas de 32 bits que pueden tomar toda una columna a la vez. Para arquitecturas de 8 bits, al parecer se puede crear código más compacto, en especial para *ShiftRows*, si reordenamos los bytes para tener lo que corresponde con el transpuesto de los originales de acuerdo a la representación matricial.

Para hacer más rápida la transposición de los datos se indicaron explícitamente qué elementos se intercambian con qué elementos.

El paso *AddRoundKey* es muy sencillo de programar y hay que notar que utilizamos operaciones de postincremento. El microcontrolador tiene la ventaja de permitir un direccionamiento indirecto con postincremento así que lo aprovechamos. No desdoblamos el *loop* porque en realidad no se gana mucho con hacerlo, pero podríamos tener código ligeramente más rápido.

En *recorre renglones* (*ShiftRows*) tratamos de reducir el tamaño del código utilizando la operación de asignación con predecremento que también se efectúa en un solo ciclo. Para direccionar la variable renglón esperamos que el compilador utilice el direccionamiento indirecto con *offset*. En realidad la forma en que implantamos esta función está lejos de ser la más compacta porque no es necesario copiar todo el renglón a un arreglo auxiliar para recorrerlo, pero esto lo tomamos en cuenta para nuestra implantación en ensamblador.

La transformación *SubBytes* debe ser implantada con cuidado porque las cajas S son muy utilizadas ($10 \times 16 + 10 \times 4 = 200$ veces en total). Se decidió dejar la tabla respectiva en memoria *SRAM* por ser la más rápida aunque pudo haberse puesto en memoria *Flash* (teniendo cierta sobrecarga). Cualquier otro tipo de memoria sería inadmisible porque el programa resultante sería muy lento.

MixColumns es la operación más compleja, pero debemos recordar que gracias al polinomio escogido cuando se definió esta transformación, puede optimizarse significativamente. No es necesario realizar multiplicaciones entre variables en $GF(2^8)$, sino de una variable por constantes sencillas así que el código se puede simplificar bastante, tal y como se sugiere en [13]. De ellos tomamos la forma de efectuar esta operación. Para la multiplicación por dos (*xtime*) nos aseguramos (desensamblamos el código objeto) de que el tiempo de ejecución fuera constante para evitar ataques por medición de tiempos.

En cuanto a la planificación de ronda, no se mantuvo toda la clave expandida en memoria, sino que se fue transformando y sobrescribiendo en las mismas localidades. Gracias a esto nos ahorramos una muy buena cantidad de memoria *SRAM*. Tratamos de optimizar esta función al menos en el tamaño del código regresándonos a la representación matricial y direccionando cada elemento con dos índices. La especificación del algoritmo nos forzó a trabajar con dos índices, pero esperamos corregir esto en la versión en ensamblador.

Desventajas de la Implantación

Uno de los problemas que se presentaron con esta implantación fue que las funciones declaradas como *inline* permanecían como funciones que eran llamadas de forma normal. Intentamos solucionar esto dando ciertos parámetros de optimización al compilador pero ninguno funcionó. Encontramos que es muy recomendable compilar con la opción *–Os*, de lo contrario se produce código extremadamente grande (del doble de tamaño o más) y las funciones siguen sin ser *inline*.

Comparando el código en ensamblador del programa desensamblado con el del código en lenguaje *C* podemos ver que muchas sentencias son compiladas en unas cuantas instrucciones en ensamblador, pero si queremos optimizar más el algoritmo lo que debemos hacer es transformarlo de fondo. Nosotros tenemos cierta ventaja sobre el compilador porque sabemos el significado de las operaciones que queremos realizar y podemos hacer modificaciones mayores a la lógica del algoritmo.

4.4. Desarrollo y Optimización del *AES* en Bajo Nivel

En esta sección vamos a presentar algunas consideraciones sobre el proceso de implantación del *AES* en ensamblador para nuestra arquitectura. Dicho proceso se benefició de lo aprendido en la implantación en lenguaje *C*, pero busca generar un código más pequeño y rápido con base en nuestro conocimiento del algoritmo y de la arquitectura. Si bien ya se tiene cierta teoría de como implantar el algoritmo en arquitecturas de 8 bits, al momento de hacerlo surgen muchas posibilidades y es difícil decidirse por una de ellas.

La codificación en sí fue un proceso iterativo y en buena parte experimental. En ocasiones no es sencillo estimar con precisión el tiempo en que se efectuará una transformación y la memoria que ocupará si se programa de cierta manera hasta que se ha implantado de esa forma.

4.4.1. Criterio de Optimización

Al igual que hicimos para nuestra implantación del algoritmo en lenguaje *C*, debemos especificar algunos requisitos u objetivos de nuestra implantación en ensamblador. Algunas veces el criterio principal para la implantación del algoritmo es la velocidad de cifrado, pero en nuestro caso, como estamos en un ambiente muy restringido, tenemos otras limitaciones o metas más importantes.

Como ya se mencionó, el espacio en memoria de programa que deja libre *SOSSE* es muy escaso. Podemos incrementarlo en algunos cientos de bytes recortando su funcionalidad, pero solamente hasta un cierto punto. Esto ocasionó que nuestra mayor preocupación fuera el uso de la memoria *Flash*.

En cuanto a la velocidad, también es un factor importante, pero no debemos gastar demasiado esfuerzo en reducirlo porque la implantación en lenguaje *C* ya produjo tiempos de cifrado aceptables. Sin embargo, dado que queremos ocupar al *AES* como componente de alguna primitiva criptográfica más compleja, debemos tratar de minimizarlo siempre y cuando esto no afecte significativamente el número de instrucciones.

Otro punto que debemos considerar es que nuestro algoritmo no funcionará aislado, sino que será llamado desde programas escritos en lenguaje *C* (*SOSSE* está escrito en su mayor parte en lenguaje *C*). Por ello debemos seguir ciertas convenciones respecto al uso de registros y al paso de parámetros.

Al hablar de optimización en general, es necesario establecer un criterio formal que nos permita decir cuándo una solución es óptima y cuándo no. Podríamos intentar la minimización de alguna función dada

en términos de la velocidad de ejecución y el uso de la memoria. Desgraciadamente no parece sencillo traducir esta clase de planteamiento a una implantación. Por ello decidimos trabajar con un criterio más relajado, que es satisfacer las limitaciones que acabamos de presentar en esta sección, con las cuales no obtendremos ningún valor óptimo pero sí una implantación útil y que haga buen uso de los recursos disponibles.

4.4.2. Selección de los Tipos de Memoria Utilizada

De acuerdo a la especificación del *AES*, el algoritmo se compone de transformaciones que afectan al estado y a la clave. Estas operaciones no pueden hacerse directamente sobre la memoria (el conjunto de instrucciones no lo permite) sino que deben traerse los datos involucrados de la memoria a los registros, donde son procesados y luego mandados de regreso.

El uso de la memoria es intensivo, así que debemos hacer una buena elección sobre en donde se localizarán los datos. Debemos recordar que nuestra arquitectura cuenta con muchos tipos de localidades de almacenamiento: memoria de programa (*Flash*), memoria *SRAM*, memoria *EEPROM* interna y memoria *EEPROM* externa. Cada uno de ellos presenta características distintas en cuanto a capacidad, tiempos de lectura, tiempos de escritura y métodos de direccionamiento.

Respecto a nuestra implantación en lenguaje *C*, tenemos una diferencia importante en la localización de la *caja S* (S–box). Nos parece inapropiado gastar la mitad de toda la memoria *SRAM* para guardar esta tabla constante. A pesar de que el acceso a esta memoria es rápido (2 ciclos de reloj) y tenemos mucha flexibilidad para accederla (mediante varios modos de direccionamiento) es inaceptable para la implantación real mantener en todo momento esta información en *SRAM*. Las memorias *EEPROM* son muy lentas: su escritura toma al menos 2.5ms y no se encontró información sobre el tiempo de lectura, pero tampoco es constante.

Decidimos almacenar la caja S en la memoria de programa, con lo cual *no* incrementamos su uso ya que las constantes de la memoria *SRAM* que no son cero siempre se almacenan de todas formas en la *Flash*, de donde son copiadas a la *SRAM*. El tiempo de acceso es más lento (3 ciclos de reloj) y solamente se puede hacer a través del registro para direccionamiento indirecto *Z*, además el dato leído siempre se carga en el registro r0 (otros microcontroladores de la familia sí permiten guardar el dato en cualquier registro). El tiempo extra que requiere la sustitución con la caja S fue compensado gracias a otras transformaciones que sí aceleramos.

El estado se compone de 16 bytes y tenemos 32 en los registros así que suena tentador guardar el estado completo en ellos, al menos temporalmente, para transformarlo y posteriormente devolverlo a la memoria *SRAM*. El problema que aparece es que no hay una verdadera ganancia en hacerlo porque el acceso a los registros solamente es rápido cuando utilizamos el direccionamiento directo a registros, pero este direccionamiento requiere especificar exactamente a que registro queremos acceder, lo cual no nos permitiría crear *loops* para reducir el tamaño de la memoria de programa. Otra forma de acceder a los registros es mediante el espacio de memoria de datos (los registros están mapeados a los primeros 32 bytes) pero todos los métodos de direccionemiento a memoria ocupan el mismo tiempo independientemente de si se trata de registros o memoria *SRAM*, así que la ganancia se disuelve. Por otro lado, al ocupar una mayor cantidad de registros disminuimos el número de registros de trabajo que podríamos utilizar para realizar operaciones más eficientemente. A causa de esto decidimos nunca copiar el estado completo a los registros, solamente se copian los bytes que van siendo necesarios.

En el paso *mezcla columnas* (con una de nuestras versiones del *AES*) utilizamos una técnica intermedia, copiamos toda una columna del estado a los registros, y la procesamos. Cada vez que tenemos un byte del resultado listo lo guardamos en memoria. Esto se hizo para reducir el tamaño del código sin

afectar tanto la velocidad. Conceptualmente esto equivale a pasar de la multiplicación matricial a una convolución circular y de ahí a una convolución lineal. Aprovechando la eliminación de la lógica de la convolución circular pudimos reducir el tiempo de cálculo.

La clave debido a que sufre una transformación especialmente secuencial (cada byte de salida depende de un par de valores previamente calculados), permanece en memoria *SRAM* tanto como es posible, claro teniendo cuidado de no repetir accesos innecesariamente. No se gana nada con copiarla completa o en bloques a los registros.

Gracias a que el número de registros disponibles es alto, podemos guardar todas las variables que controlan el flujo del algoritmo (como el número de la iteración) en ellos. Las constantes de ronda pueden calcularse sin ningún problema en tiempo de ejecución en el orden en que son necesitadas así que no necesitamos mantenerlas en la memoria principal.

4.4.3. Características del Conjunto de Instrucciones

El lenguaje ensamblador del microcontrolador *AT90S8515* es relativamente sencillo de aprender. Primero que nada cabe señalar que el lenguaje ensamblador de *IAR* difiere ligeramente al del ensamblador *avr–as*, que es el que utilizamos, pero son prácticamente iguales.

El microcontrolador puede ejecutar la mayoría de las instrucciones en un ciclo de reloj. La frecuencia del reloj no es dividida sino que se toma tal cual de algún dispositivo externo (en este caso del lector de tarjetas). Internamente tenemos un *pipeline* de 2 etapas, pero éste es completamente transparente para el programador, así que no es necesario hacer ninguna consideración adicional.

Los 32 registros r0–r31 son de propósito general, esto es, pueden ser utilizados la mayoría de las veces con cualquier instrucción. Lo contrario a esto puede ser una arquitectura con registros acumuladores, en la que para realizar una adición necesitamos mover uno de los operandos a alguno de ellos. Con la arquitectura utilizada tenemos bastante libertad en cuanto a para qué utilizamos los registros.

Como los registros están conectados directamente a la *ALU*, podemos ejecutar instrucciones que operan con dos registros y guardar el resultado en uno de ellos en solamente un ciclo de reloj. También hay algunas instrucciones útiles que operan sobre 16 bits simultáneamente.

A continuación veremos como son utilizadas algunas de las instrucciones más representativas que fueron útiles para la implantación del algoritmo:

`eor Rd,Rr`	Hace el or–exclusivo de Rd y Rr guardando el resultado en Rd
`lpm`	Carga en r0 el contenido de la localidad Z=r31:r30 de la memoria de programa
`st Y+,Rr`	Guarda Rr en la localidad de memoria dada por Y=r29:r28 y postincrementa Y
`ldd Rd,Y+q`	Carga en Rd lo que se encuentre en la dirección $Y+q$
`sbiw Rd,K`	Resta a Rd+1:Rd el valor de la constante K

Las instrucciones con postincremento y con predecremento resultaron ser muy útiles para acceder secuencialmente a la memoria. Contamos con tres índices o registros de 16 bits: X, Y y Z, compuestos cada uno por la concatenación de dos registros, mediante los cuales podemos acceder a la memoria con el direccionamiento indirecto. Con los registros Y y Z también podemos utilizar el modo de direccionamiento indirecto con desplazamiento. Este modo resultó ser muy poderoso y práctico. Nos permitió acceder al estado y a la clave con poca sobrecarga gracias a que colocamos a la clave inmediatamente después del estado. Para ilustrar esto digamos que Y apunta al primer byte del estado, entonces $Y+16$ apunta al primer byte de la clave y fácilmente podemos sumar cualquier byte de la clave al estado sin tener que mantener dos índices separados.

De igual forma, si Y apunta al primer byte de cualquier columna del estado, $Y+4$, $Y+8$ y $Y+12$ apuntan a los bytes restantes de la columna y no tenemos que hacer la suma explícita para direccionarlos.

4.4.4. Serialización de Operaciones

De acuerdo a la propuesta de *Rijndael*, es posible combinar las operaciones de la adición de la clave, la sustitución de bytes y el paso recorrer renglones en las arquitecturas de 8 bits. Esto quiere decir que en lugar de aplicar cada una de las transformaciones separadamente sobre todos los bytes del estado, podemos aplicar las tres operaciones sobre un byte a la vez. A esto se le llama serialización de operaciones.

El propósito de la serialización es reducir el número de accesos a memoria, y la carga computacional asociada al movimiento de los apuntadores (índices). En nuestra implantación realizamos la serialización de los tres pasos descritos, y tratamos de reducir sustancialmente el tamaño del código creando los *loops* adecuados tratando de nunca efectuar cálculos que no sean estrictamente necesarios.

La transformación conjunta de adición de clave, sustitución de bytes y recorrer renglones se llama en el código `xor_sust_recorre`. Está programada muy eficientemente, en realidad hicimos dos versiones, la primera no tiene ningún *loop* pero las instrucciones están organizadas de forma en que fácilmente pudieran incluirse y en la segunda sí se tienen ciclos. En la primera implantación nos parece muy difícil que podamos tener otra implantación que utilice la memoria para almacenar el estado y sea más de 5 instrucciones más corta o más de 10 ciclos de reloj más rápida debido a que cada instrucción utilizada tiene un propósito muy específico directamente relacionado con los pasos que tenemos que ejecutar. Es posible reducir el código en 3 instrucciones (y 6 ciclos de reloj) pero eso descompondría la estructura del algoritmo y no la podríamos implantar iterativamente.

Exploramos la posibilidad de también serializar la operación de mezclar columnas, pero no es sencillo hacerlo porque cuando la combinamos con la transformación de recorrer renglones tendríamos que almacenar varias variables temporales e implantar una lógica mucho más compleja, lo que nos alejaría de nuestro objetivo de tener un código compacto.

4.4.5. Incrustación de Tablas de Datos en la Memoria de Programa

El paso de sustituir bytes, como ya se mencionó, se implantó como una consulta a una tabla que se encuentra en la memoria de programa. Debido a que la arquitectura no presenta tanta flexibilidad para acceder a la memoria *Flash* como a la *SRAM*, fácilmente podríamos caer en una disminución de la velocidad de la consulta y un aumento del tamaño del código comparado con el acceso regular a la *SRAM*.

Los microcontroladores de la familia *AVR* varían ligeramente en cuanto al acceso a la memoria de programa. Para el *AT90S8515* la única forma de hacerlo es a través del índice Z, que no puede ser postincrementado ni predecrementado (pero en otros microcontroladores de la familia sí). El resultado siempre se entrega en el registro r0.

El procedimiento para acceder a constantes en memoria de programa, recordando que Z=r31:r30, puede ilustrarse con el siguiente código:

```
ldi     r31,hi8(Srd)    // guarda en ZH la parte alta de la direccion de la S-box
ld      r30,Y           // guarda en ZL el byte que queremos sustituir, al que Y apunta
lpm                     // el resultado de la consulta se encuentra en r0
```

Naturalmente en la implantación no utilizamos los nombres de los registros directamente sino sus alias, para mejorar la legibilidad del código.

Debido a que alineamos la caja S, aquí llamada *Srd* al igual que en el documento de diseño de *Rijndael*, haciendo que su dirección sea múltiplo de 256, podemos transformar cada byte sin tener que rescribir la parte alta de la dirección cada vez ni tener que hacer sumas de 16 bits. Solamente especificamos la parte baja de *Z*.

Las consultas se realizaron de tal forma que únicamente se gastó un ciclo extra comparado con lo que nos hubiera costado acceder a la memoria *SRAM* (3 ciclos en comparación con 2), no se desperdició ninguna instrucción en mover los operandos o el resultado de `lpm` a otros registros.

4.4.6. Transformación *MixColumns*

La transformación mezclar columnas es la que computacionalmente es más costosa. Afortunadamente contamos con una sugerencia de implantación eficiente proporcionada en [13], pero aún siguiendo este consejo ejemplificado en un pseudocódigo de lenguaje *C*, es claro que la transformación requiere más cálculos que las demás.

La implantación inicial del algoritmo de la transformación *MixColumns* dio un buen resultado en cuanto a velocidad. Desgraciadamente dicha implantación ocupó demasiado espacio ya que el acceso los registros fue con su nombre y no fue posible acortar el código mediante ciclos para eliminar código repetido o muy similar. Debido a esta deficiencia se pensó en otras soluciones. Este paso es el que más posibles implantaciones con diferencias importantes puede producir. No es sencillo encontrar formas que sigan siendo rápidas pero que tengan menor tamaño. Después de pensar un largo rato pudimos obtener otras dos alternativas.

La primera nueva solución a la que llegamos utiliza un índice extra *Z* (además de *Y*) para acceder a través del espacio de memoria de datos a los registros (recordar que los registros ocupan los primeros 32 bytes) y cuando fuera conveniente utilizamos direccionamiento directo a registros para minimizar el tiempo de algunos cálculos que se ocupan para cada byte de la columna. Mediante el índice *Z* podemos crear ciclos accediendo a los valores de los registros. La razón por la que no utilizamos *Z* para apuntar a los bytes de la columna directamente en su ubicación original es que solamente podríamos tener un *loop* de 3 ciclos y tendríamos que colocar el equivalente a la cuarta iteración afuera de él porque es ligeramente distinta. Esta solución fue 30 % más pequeña pero 73 % más lenta que la anterior.

La segunda propuesta es un compromiso entre ambas soluciones. En ésta los datos se conservaron en memoria tanto como fue posible y su acceso se hizo mediante el mismo apuntador *Y* que utilizamos para los datos y la clave. Para poder programar eficientemente esta alternativa, se impuso un requisito extra que no habíamos establecido: el estado debe estar en una posición múltiplo de 16 (al igual que la clave puesto que se encuentra inmediatamente después del estado). La razón del nuevo requisito es que para evitar el uso de contadores adicionales se utilizó al mismo apuntador *Y* como contador, y para detectar que la cuenta ha llegado a su punto final empleamos el acarreo medio (*half carry*). Si los datos no estuvieran alineados no funcionaría nuestra lógica con el acarreo medio. Esta segunda solución disminuyó aún más el tamaño del código y en cuanto a la velocidad está en un punto intermedio entre las otras dos soluciones.

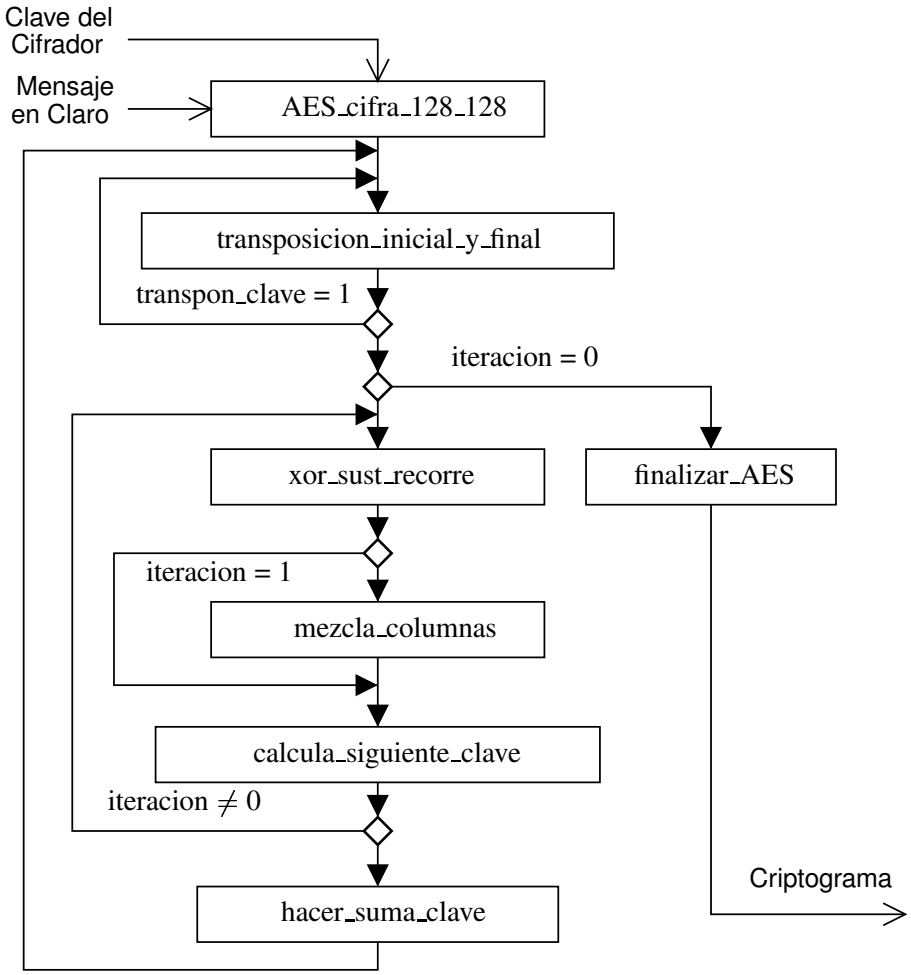

Figura 4.3: Esquema del flujo del programa en ensamblador con los bloques más importantes.

4.4.7. Planificación de Clave

Como ya se mencionó, la clave fue calculada ronda por ronda a medida que era necesitada de manera que no se desperdició memoria *SRAM*. Si bien la especificación del algoritmo no es muy clara en cuanto a cómo se transforma la clave, gracias a las ecuaciones generadas en el capítulo anterior y a la representación gráfica de cómo se deriva la siguiente clave de ronda, pudimos fácilmente implantar esta transformación. Hay cierta asimetría en la forma en que es tratado cada renglón pero con una condición sencilla pudimos meter todo el código de esta transformación en un solo ciclo de 4 iteraciones.

Todavía es posible poner otro ciclo anidado dentro del ciclo antes mencionado, pero no vale la pena porque el contenido del *loop* sería muy escaso y esta parte del código se haría más lenta, prácticamente al doble de tiempo a causa de la sobrecarga que ocasionarían los saltos, además la reducción neta del tamaño del código sería solamente de unos 12 bytes.

EL valor de la constante de ronda (*RC*) no ocupa memoria *SRAM* sino que se mantiene en un registro y es actualizada al final de cada ronda (excepto la última).

4.5. El Sistema Operativo de las Tarjetas: *SOSSE*

El *hardware* de nuestra arquitectura es muy modesto en comparación con el *hardware* de una computadora personal actual, sin embargo es suficiente para albergar a un sistema operativo compacto. El sistema operativo de las tarjetas (*COS*) que utilizamos se llama *SOSSE*[3]. Es desarrollado por *Matthias Brüestle* (sigue en desarrollo) y puede obtenerse gratuitamente en la red.

Una de las razones por las que utilizamos *SOSSE* es porque se trata de un sistema operativo libre, así que tenemos acceso al código fuente y podemos modificarlo.

SOSSE no trabaja con otros programas (no tiene el concepto de proceso), sino que él mismo establece toda la funcionalidad de la tarjeta. Si se desea añadir algún comportamiento a la tarjeta debe modificarse directamente el sistema operativo. En realidad no es difícil añadir nuevos comandos gracias a que ya se tiene una serie de rutinas que se encargan de los aspectos de bajo nivel, como por ejemplo de la transmisión serial.

Los módulos o componentes principales de *SOSSE* son los siguientes:

- *Módulo de Entrada / Salida*. Se trata de las funciones básicas para transmitir y recibir información de acuerdo al protocolo *T=0*. Mediante este módulo es que nos podemos comunicar con el exterior.
- *Acceso a Memoria EEPROM*. Es posible emplear la memoria *EEPROM* interna y externa sin tener que preocuparnos por cuestiones como el tiempo de escritura, o de que tipo de memoria se trata.
- *Sistema de Archivos*. *SOSSE* tiene un sistema de archivos jerárquico y el acceso a los archivos está restringido de acuerdo al nivel de autenticación.
- *Algoritmos Criptográficos*. Se incluye una implantación en ensamblador de un algoritmo criptográfico llamado *TEA*[4] que puede describirse de forma muy sencilla en lenguaje *C*.
- *Transacciones*. En ocasiones es importante mantener un estado válido aún cuando la tarjeta inteligente sea extraída mientras se escribe en la memoria *EEPROM*. Desde el punto de vista de la seguridad, esto es importante para prevenir la manipulación de los contadores internos.
- *Autenticación*. Contamos con una serie de funciones útiles para controlar el acceso a los archivos y para modificar el *PIN*.
- *Interprete de Comandos*. Hay un ciclo infinito en el cual se lee un comando del lector, se determina de qué comando se trata y se ejecuta la acción correspondiente.

Para utilizar el *hardware* no es necesario ni conveniente hacerlo directamente puesto que estaríamos "reinventando la rueda" y el código resultante no sería portable. La manipulación de los recursos de más bajo nivel es encapsulada mediante una *Capa de Abstracción de Hardware* o *HAL*[5]. En caso de que actualicemos la arquitectura todo lo debería seguir funcionando correctamente, excepto por la *HAL* que tendríamos que reprogramar. De acuerdo a los componentes antes descritos, esta capa incluye al módulo de entrada y salida y al de acceso a la memoria *EEPROM*.

Los comandos más importantes que *SOSSE* reconoce son:

[3] *Simple Operating System for Smartcard Education*.
[4] *Tiny Encryption Algorithm*.
[5] *Hardware Abstraction Layer*.

- Autenticarnos mediante un *PIN*
- Autenticarnos mediante una clave (autenticación débil)
- Autenticarnos mediante un protocolo de reto / respuesta (autenticación fuerte)
- Autenticar a la tarjeta mediante un protocolo de reto / respuesta (autenticación fuerte)
- Crear archivos (elementales o dedicados)
- Modificar el contenido de un archivo
- Eliminar un archivo
- Leer el contenido de un archivo
- Cambiar o desbloquear el PIN

Estas funciones son más que suficientes para realizar una aplicación basada en tarjetas inteligentes.

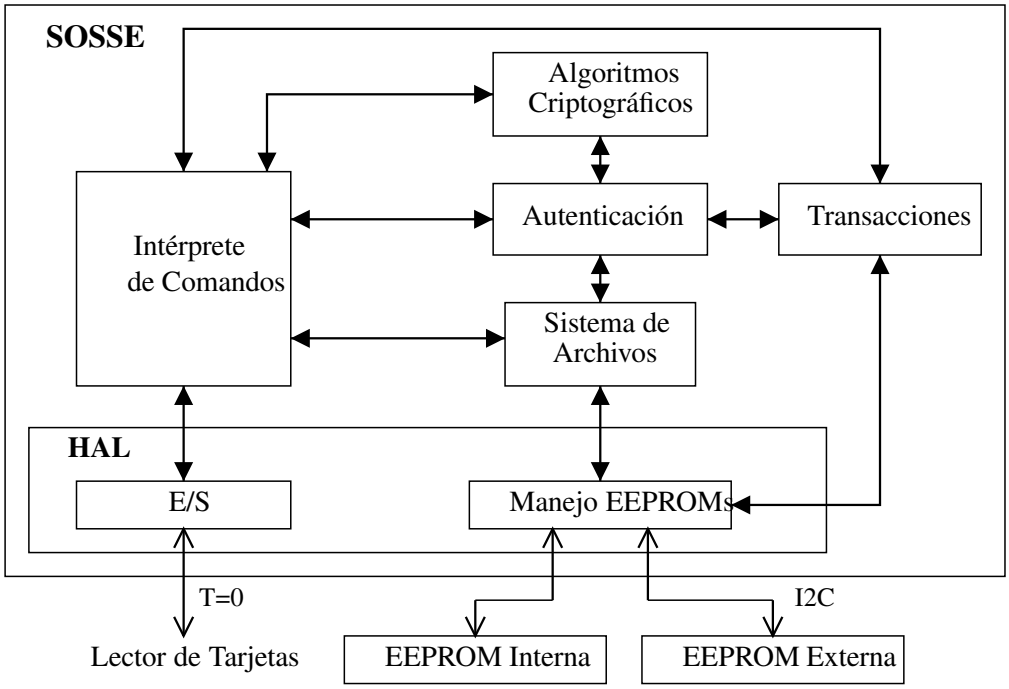

Figura 4.4: Esquema representando los flujos de datos más importantes en *SOSSE*.

Características de *SOSSE*

Ahora daremos un poco más de detalles sobre el *COS*. Prácticamente todo *SOSSE* está programado en lenguaje *C*, con excepción de algunas funciones que requieren un mayor control sobre el microcontrolador como la comunicación. El puerto serial está implantado completamente en *software*. Se necesita leer y escribir los datos en el *pin* o contacto del microcontrolador que sirve de puerto serial en tiempos determinados. Para mantener los tiempos adecuados se sabe con precisión el tiempo que tarda la ejecución de las rutinas de comunicación y se tienen algunos *loops* de retardo.

Otro componente que está escrito en ensamblador es el acceso a la memoria *EEPROM* externa. Para ello hay un protocolo serial denominado *I2C*. Se soportan varios tamaños de memoria externa.

El tercer componente escrito en ensamblador es el algoritmo criptográfico *TEA*. Se proporciona también una implantación en lenguaje *C* (que nos puede servir para comprobar el cifrado). Este algoritmo tiene una descripción sumamente simple, pero presenta las desventajas de que requiere un número alto de rondas, trabaja sobre palabras de 4 bytes y su tamaño de bloque es de 8 bytes.

En cuanto al sistema de archivos se soportan los archivos dedicados (*DF*) y los archivos elementales (*EF*). Los archivos dedicados son el equivalente a lo que conocemos como directorios. En cuanto a los elementales tenemos solamente los transparentes, pero en nuestra opinión esto es suficiente. Los archivos cíclicos y con registros pueden trabajarse utilizando archivos transparentes y una lógica fuera de la tarjeta.

Se incluyen funciones sofisticadas en el sistema de archivos como el manejo de flujos además de las ya mencionadas: creación, modificación, lectura, y eliminación de archivos.

El *PUK* es una clave con la que podemos desbloquear o cambiar el *PIN* en caso de que éste haya sido introducido incorrectamente más veces de lo permitido. Cada vez que el *PIN* o la clave son dados incorrectamente se decrementa el contador respectivo y si llegamos a cero se bloquea la autentificación con *PIN* o clave, de forma que determinarlos por mero ensayo y error es poco probable sencillo porque tenemos un número limitado de intentos.

Instalación de *SOSSE*

Es muy sencillo instalar *SOSSE*. Este proceso está documentado dentro de la misma distribución que contiene el código fuente. De cualquier forma con el fin de la completitud y para ahorrarle tiempo y trabajo al lector se indica aquí como hacerlo.

Primero debemos bajar el código fuente de `ftp://ftp.franken.de/pub/crypt/chipcards/sosse/`, al momento de escribir esta tesis la última versión fue la del 24 de Diciembre del 2002. Luego proseguimos con la instalación:

```
[eban@power tesis]$ tar -xzvf sosse-20021224.tar.gz
[eban@power tesis]$ cd sosse-20021224
[eban@power sosse-20021224]$ make
```

Ahora dentro del subdirectorio `src` debemos tener un archivo *ELF* llamado `sosse`, la versión en binario `sosse.bin` junto con la información de la *EEPROM* interna `eedata.bin` y las versiones en hexadecimal: `sosse.hex` y `eedata.hex`.

Para la simulación es importante producir las versiones binarias, para la depuración con *gdb* se requiere el archivo *ELF* y para cargar el sistema operativo a la tarjeta mediante el software que utilizamos (*Master Burner*) se necesitan los archivos en hexadecimal.

Ventajas de *SOSSE* sobre otros Sistemas Operativos

En realidad el único sistema operativo para tarjetas libre o al menos de código abierto que encontramos fue *SOSSE*, afortunadamente cumplió a la perfección con nuestras expectativas y gracias a que está programado de forma muy limpia pudimos modificarlo para cumplir con nuestros objetivos.

De acuerdo a nuestra experiencia al haber analizado el código fuente podemos decir que la manera en que está programado nos da mucha confianza porque se tiene un excelente control sobre las condiciones de error y sobre el estado en cuanto a la autenticación y al acceso a archivos.

Si bien no hicimos las pruebas extensivas que requiere un sistema de la complejidad de *SOSSE*, al menos por lo que pudimos constatar y por el código fuente creemos que *SOSSE* funciona de forma muy confiable.

Consideraciones sobre los Métodos para Simular *SOSSE*

No solamente es posible compilar a *SOSSE* para generar el sistema operativo en sí, sino también para realizar pruebas haciendo que funcione en la computadora anfitriona. Esto tiene la desventaja de que en realidad no estaremos trabajando con las instrucciones del microcontrolador que nos interesa, sino que solamente se comportará el programa de la misma forma (de la forma en que lo indica el programa en lenguaje *C*). Es decir, se tiene la misma funcionalidad, pero en el fondo se trata de un archivo compilado para el sistema anfitrión. Hacer esto es posible gracias a la capa de abstracción de *hardware*, en donde en lugar de ejecutar las rutinas de bajo nivel se simula la arquitectura. A nosotros nos interesa hacer pruebas con una simulación que replique de forma más precisa el *hardware* real por lo que no utilizamos este modo de compilación.

Otra forma de compilar a *SOSSE* es generando un "lector virtual" utilizando una "tarjeta virtual" con los que podemos interactuar mediante la interfaz *CT-API*. Esta opción se encontró en la versión de Diciembre del 2002 y no pudimos explorarla por razones de tiempo, además sufre el mismo problema de la opción de compilación anterior en la cual el producto de la compilación es para el sistema anfitrión y no para la tarjeta real. Obviamente no nos sirve la información sobre la velocidad de ejecución y el tamaño de código resultante.

En nuestra opinión nuestro enfoque de simulación es superior en este respecto porque nosotros sí simulamos el programa binario real que se cargaría en la tarjeta, así que en teoría obtendremos resultados más cercanos a los resultados que da la arquitectura real. Un ejemplo de esto es si compilamos una versión de *SOSSE* que se excede del tamaño de la memoria de programa, nosotros sí detectaríamos el problema mientras que con el "lector virtual" el sistema operativo incorrectamente sí funcionaría. Esto se debe a que el *hardware* de la PC no tiene la misma limitación en cuanto al tamaño de la memoria de programa.

Más adelante describiremos cada una de las herramientas que creamos para poder simular a *SOSSE* (secciones 4.7, 4.8.2 y 4.9).

4.6. Requerimientos de *Hardware* y *Software*

En esta sección describiremos el *hardware* y *software* que utilizamos y que se necesita para trabajar con el sistema y continuar con su desarrollo. Debemos distinguir entre los principales escenarios que podemos encontrar:

- Implantación de algoritmos criptográficos y extensión del sistema operativo de la arquitectura.
- Desarrollo de una aplicación basada en tarjetas inteligentes que utilice nuestra implantación de los algoritmos.
- Uso de las aplicaciones desarrolladas.

Los tres escenarios pueden realizarse sobre la arquitectura real o sobre la arquitectura simulada. Cuando desarrollemos una aplicación para el usuario final, obviamente tarde o temprano vamos a migrar

a la arquitectura real, pero en el proceso podemos ayudarnos con la simulación sobretodo para facilitar las pruebas y la depuración.

Lectores de Tarjetas

En el mercado hay diversos tipos de lectores con precios muy variados, incluso hay algunos con precios especialmente accesibles para desarrolladores en *Linux* (*Towitoko*). En lugar de gastar en los lectores tuvimos la fortuna de que la *DGSCA*[6] nos donara algunas decenas que tenía sin utilizar y que habían sido adquiridos para un proyecto de automatización del acceso y uso de las salas de cómputo, con una donación de *fundación UNAM*.

Este proyecto por alguna razón desconocida falló pero nosotros tenemos ahora ese equipo y podemos intentar otros proyectos. Los lectores son de marca *Phillips*, modelo *PE112* y se comunican con el *host* por una interfaz serial. Están un poco viejos, fueron importados en el año de 1996, pero han funcionado correctamente.

Se buscaron los *drivers* extensivamente en la red, tanto para *Windows* como para *Linux*, sin embargo al parecer estos lectores ya no tienen ningún soporte del fabricante. El *driver* de *Windows* definitivamente no se encontró, pero el de *Linux* está disponible en la página del proyecto *MUSCLE*[7] (*Movimiento para el Uso de Tarjetas Inteligentes en un Ambiente Linux*).

El *driver* trabaja casi necesariamente en conjunción con *PC/SC* porque implanta las interfaces que ocupa *pcsclite*. Desgraciadamente no ha sido actualizado y no es posible compilarlo directamente para la versión actual de *pcsclite* (que es la 1.1.1). Gracias a que es libre pudimos hacerle las modificaciones que necesitaba para funcionar con la nueva versión y que afortunadamente resultaron menores. Las modificaciones quedaron registradas en el "parche" que creamos y que pronto mandaremos a la lista de mensajes de *MUSCLE*. La instalación y el parchado del *driver* se explicarán en la sección 4.8 sobre *PC/SC*.

Podemos utilizar lectores de tarjetas distintos al que ocupamos nosotros, pero sí es importante contar con el *driver*. Nosotros desarrollamos bajo *Linux* y sugerimos ampliamente que así siga siendo con cualquier trabajo futuro, pero también es posible hacerlo bajo *Windows* ya que varias de las herramientas libres que utilizamos han sido portadas a este sistema operativo.

Sistema Anfitrión

El *hardware* del equipo anfitrión puede ser una PC común, no necesita nada especial, solamente requiere unos cuantos MBytes libres para instalar las herramientas (con unos 150 MBytes es suficiente), un puerto serial disponible (o USB en dado caso) para utilizar el lector o el programador de tarjetas, unos cuantos MBytes de memoria *RAM* y el *hardware* de propósito general. Nosotros utilizamos una *Pentium III* a 800 MHz con 256 MBytes de *RAM*.

Si bien las exigencias de *hardware* son bajas, es preferible tener una máquina rápida, sobretodo si vamos a compilar todas las herramientas o si vamos a simular la arquitectura (aunque una máquina promedio de unos 700 MHz es suficiente). El sistema donde funciona la aplicación final requiere menos recursos puesto que no necesita el ambiente de desarrollo.

El *software* que empleamos viene en la distribución de *Linux* utilizada (*RedHat 8.0*), excepto por las herramientas que se han venido describiendo y los programas desarrollados por nosotros. En realidad

[6] Dirección General de Servicios de Cómputo Académico (de la UNAM).

[7] *Movement for the Use of Smart Cards in a Linux Environment*.

no estamos encadenados a ningún sistema operativo y para la operación normal de la tarjeta podemos utilizar tanto *Linux* como *Windows*.

Figura 4.5: El programador de tarjetas (centro) y los lectores de tarjetas (arriba y abajo).

Programador de Tarjetas

También necesitamos un programador de tarjetas. Hay varios disponibles en `http://www.wafer-shop.de` y su costo aproximado es de 60 Euros. Para la operación normal de las tarjetas (lecturas y escrituras incluidas) es suficiente el lector, pero para programar la memoria *Flash*, es necesario seguir un protocolo especial que está implantado en el programador.

Podemos utilizar cualquiera que soporte las tarjetas *funcards*, pero nosotros empleamos *MasterCRD LCD 2*. Su precio fue de 73 Euros y no se encontró mucha información sobre él, pero al menos hay un documento disponible en `http://www.wafer-shop.de/dlzone/mastercrd.htm` que indica los modos de operación, el voltaje y la polaridad del aparato (saber esto es necesario pues el convertidor de voltaje que lo acompaña funciona con el voltaje de Europa así que hay que comprar otro).

En esta misma dirección electrónica es posible bajar el *software* correspondiente. Hay varios programas pero nosotros utilizamos el llamado *Master Burner*. Desgraciadamente no encontramos un programador que funcione bajo *Linux*, así que la programación de las tarjetas tiene que hacerse bajo *Windows*, afortunadamente la programación en la práctica se hace una sola vez por tarjeta y en general en el proceso de desarrollo es una operación infrecuente.

Master Burner resultó ser un programa muy sencillo de utilizar porque tiene una interfaz altamente intuitiva y las funciones que ofrece son muy simples.

Tarjetas Inteligentes

Podemos trabajar con cualesquiera tarjetas que tengan el procesador *AT90S8515*. *SOSSE* soporta algunos otros microcontroladores de la familia pero nuestra implantación en ensamblador del *AES* está hecha específicamente para el *AT90S8515*. En realidad las modificaciones necesarias para que nuestra nueva

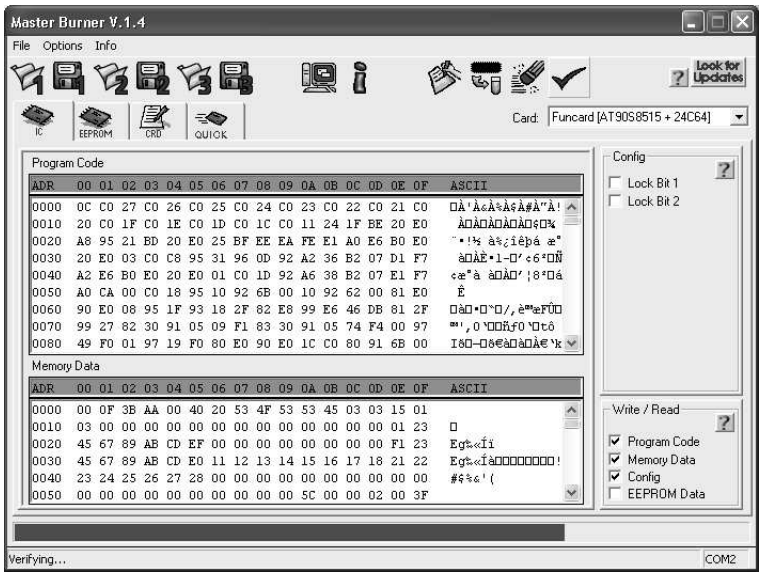

Figura 4.6: *Master Burner* verificando la programación de la tarjeta después de cargar *SOSSE* en ella.

versión del *COS* funcione en otros microcontroladores de la misma familia son mínimas puesto que la tendencia es cada vez utilizar dispositivos más potentes con conjuntos de instrucciones más ricos, así que la mayoría de las instrucciones deben seguir funcionando.

Figura 4.7: Las vistosas tarjetas *Funcard 2* y *Funcard 4*.

Contamos por el momento con 3 tarjetas inteligentes:

- 2 tarjetas “funcard 2” con 64 Kbits de memoria *EEPROM* externa y
- 1 tarjeta “funcard 4” con 256 Kbits de memoria *EEPROM* externa.

El costo aproximado de estas tarjetas es de 7 y 8 Euros. Hay otras tarjetas disponibles en caso de requerir más memoria, como la “funcard 6” que cuesta 15 Euros y tiene memoria externa de 1024 Kbits.

Internamente estas tarjetas inteligentes son equivalentes a un microcontrolador conectado a una memoria *EEPROM* serial y a los contactos de la tarjeta. Incluso es posible encontrar algunas con *leds*, lo cual puede ser útil para conocer el estado interno de la tarjeta al depurar nuestro código.

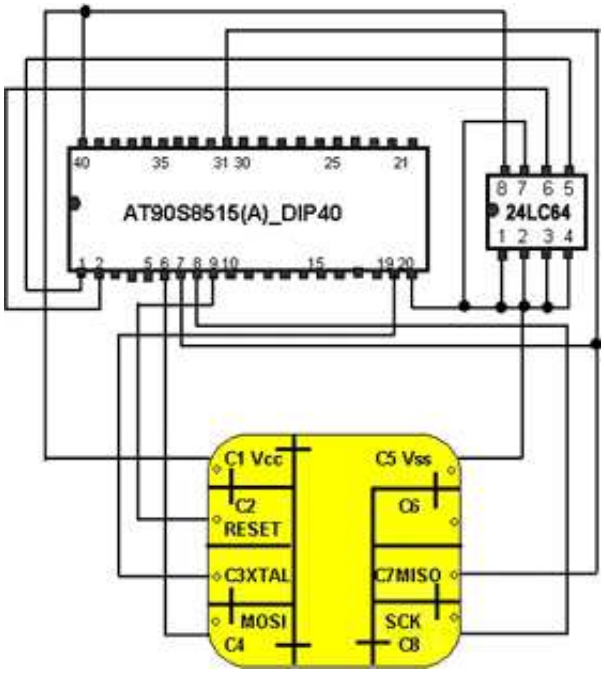

Figura 4.8: Esquema representativo de la interconexión del microcontrolador, la memoria y los contactos para la tarjeta *Funcard 2*. Imagen tomada de [8].

Al parecer las tarjetas basadas en los microcontroladores *Atmega* (que son de la misma familia *AVR* pero más potentes) soportan la programación a través de lectores de tarjeta comunes puesto que tienen un pequeño programa cargado. Al momento de escribir este trabajo no se ha encontrado más información de como hacerlo, pero al parecer es una característica muy útil.

Por las tres tarjetas mencionadas y el programador pagamos en total $1700 pesos, lo cual incluye el costo del envío y la comisión del banco por hacer el depósito.

4.7. Simulación del Protocolo *T=0* y la Comunicación con la Tarjeta

En esta sección hablaremos sobre cómo simulamos al lector e implantamos la comunicación con el sistema operativo de las tarjetas a través del simulador de la arquitectura *simulavr*. Primero que nada hay que señalar que *SOSSE* se comunica con el exterior mediante un contacto o *pin*, el cual consulta para determinar si se está recibiendo un dato (bit de inicio) y posteriormente comenzar a leer la información.

La consulta o escritura de un *pin* no puede ser manejada automáticamente por *simulavr* puesto que ésta concierne a los dispositivos que estén conectados externamente al microcontrolador. Por ello cuando un programa accede a los pines, *simulavr* solicita la información que debe presentar al programa mediante una petición a la salida estándar y lee el dato por la entrada estándar, donde normalmente es tecleada por el usuario. Cuando se escribe en un *pin*, se indica por la salida estándar que así ha sucedido, lo cual normalmente aparece en pantalla.

Para transmitir información al sistema operativo es necesario lidiar tanto con el protocolo de la tarjeta que es *T=0* como con la forma de procesar las entradas y salidas en *simulavr*. Esto puede hacerse de varias formas.

La primera manera involucra la modificación del simulador mediante alguna interfaz nueva, por ejemplo un *pipe* o un *socket*, donde se pueda leer y escribir los datos sobre el estado de los pines.

La segunda opción consiste en utilizar el mecanismo actual de *simulavr* para las entradas y salidas ajustando nuestros programas para interpretar los mensajes que se muestran y que están hechos para interactuar con personas.

Nosotros pensamos que lo más adecuado era la segunda alternativa puesto que la primera opción "ensuciaría" el código fuente original dado que la verdadera solución al problema presentado creemos que es una interfaz de propósito general para simular las conexiones con otro *hardware* y esa solución ha sido tratada en la lista de discusión de *simulavr* pero por su complejidad se ha evitado o pospuesto, aunque es algo que se tiene que hacer. Con la segunda opción nuestras librerías deben funcionar transparentemente con *simulavr* y cualquier nueva versión que aparezca debe trabajar correctamente sin ser parchada.

El primer problema que se presenta con nuestra decisión es que perdemos control sobre el reloj interno que se ocupa en la simulación del microcontrolador, por lo que no podemos seguir los tiempos que se establecen en el protocolo *T=0*, por ejemplo para declarar finalizada la transmisión de un grupo de bytes.

El *software* que desarrollamos para comunicarnos con *SOSSE* (a través de una simulación) lo llamamos *lector_simulador* y se trata de una librería que es el equivalente al lector y a su *driver*. No era conveniente simular al lector con las mismas interfaces que ocupa para comunicarse con *PC/SC* dado que esto era algo más complejo de lo que necesitábamos, nos hubiera tomado más tiempo y al final de cuentas toda esta funcionalidad se desperdiciaría. En lugar de eso creamos algunas rutinas básicas que nos permiten comunicarnos directamente con *SOSSE* a través de *simulavr*. La rutina principal es `transmite_T0` que manda un bloque de datos a *SOSSE* mediante el protocolo *T=0* y lee la respuesta. Claro que internamente se tienen otras rutinas para ocuparse de la iniciación del simulador de la arquitectura y para interpretar las entradas y salidas de *simulavr*.

De esta forma *lector_simulador* se comunica con *simulavr* redireccionando la entrada y salida estándar del segundo mediante unos *pipes*.

Funcionamiento Interno de *lector_simulador*

Básicamente el simulador indica que ha escrito en un *pin* o en un registro de control mediante un mensaje de la forma:
`writing 0xab to 0xcdef`, y solicita un valor mediante:
`Enter a byte of data to read into 0xabcd:`, nuestro programa interpreta estos mensajes y responde de manera adecuada.

SOSSE disminuye los errores por falta de sincronización leyendo tres veces el estado de cada bit en intervalos de tiempo fijos. Hace el promedio de las lecturas y por mayoría decide si se leyó un 1 o un 0. Nosotros tenemos que tomar eso en cuenta y escribir cada bit tres veces.

La paridad utilizada es par, y el bit de inicio es bajo, implantar esto es muy sencillo. Los bytes se mandan con el bit menos significativo primero y al final el más significativo. Si no hubiéramos simulado al sistema operativo, darnos cuenta de estos detalles hubiera sido mucho más difícil.

Lo más interesante de *lector_simulador* es la forma en que iniciamos la comunicación con el simulador:

- Primero se crean 2 *pipes*. De esta forma se tienen 2 vías donde podemos escribir datos y sus respectivas 2 vías para leerlos.
- Después se bifurca el programa (*fork*). Una parte seguirá ejecutando el simulador del lector (proceso padre) y la otra se encargará de cargar el simulador del microcontrolador (proceso hijo).
- El proceso padre cierra una vía de escritura y una de lectura, de forma que ahora solamente el hijo puede escribir y leer en dichas vías. Hay que recordar que los descriptores de archivo de los *pipes* son duplicados para ambos procesos. Además configura su vía de lectura para que no bloquee cuando lea información, esto es importante puesto que no sabemos cuando comenzaremos a recibir datos y no queremos bloquearnos mientras no haya ninguno disponible.
- El proceso hijo reemplaza su entrada y salida estándar por dos de los descriptores de archivo que corresponden a los *pipes* que tiene y cierra los otros dos. De esta forma el hijo tiene un descriptor de archivos exclusivo para escribir datos que leerá el padre (mediante *stdout*) y otro para leer datos (mediante *stdin*) que ha escrito el padre.
- El proceso hijo obtiene una variable de ambiente llamada "LLAMAR_SIMULAVR" que le indica de que manera debe ejecutar el simulador. Esto lo hicimos puesto que no siempre queremos llamar al simulador de la misma forma (algunas veces queremos ejecutar el coproceso de *display*, otras veces queremos utilizar el depurador, etc.). La asignación de esta variable de ambiente puede realizarla también otro programa.
- De acuerdo a la variable de ambiente se crea un arreglo de apuntadores que equivale al "argv" de los programas de *C* y se reemplaza al proceso hijo por el simulador mediante una llamada a la función `excecvp`.
- Por el lado del padre se determinan algunos parámetros de la comunicación mediante las variables de ambiente "TIMEOUT_FIN_BLOQUE" y "TIMEOUT_LIMPIAR_BUFFER" que indican el tiempo que debe esperar para dar por finalizada la recepción de un bloque de datos y el tiempo que debe esperar antes de comenzar a transmitir información.

Ahora toda la entrada y salida estándar de *simulavr* son accedidas perfectamente por el proceso padre, el cual, por supuesto, conserva además su entrada y salida estándar originales. Entonces *lector_simlador* está listo para interpretar la información que le mande *SOSSE* a través de *simulavr*.

El producto de nuestro simulador del lector y su *driver* es una librería, además se proporciona un ejemplo de cómo utilizarla, aunque es preferible ver de que manera la empleamos en el simulador de *PC/SC*, que será descrito más adelante en la sección 4.8.

Instalación de *lector_simulador*

Veamos como se compila e instala *lector_simulador*, esto es muy sencillo gracias al *Makefile* que creamos:

```
[eban@power tesis]$ tar -xzvf lector_simulador.tgz
[eban@power tesis]$ cd lector_simulador
[eban@power lector_simulador]$ make
[eban@power lector_simulador]$ su root
```

```
Password:
[root@power lector_simulador]# make install
```

Después de esto hemos producido e instalado la librería *lector_simulador*, la cual podemos ligar a nuestros programas mediante el argumento a *gcc*: `-llector_simulador`

También podemos generar un ejemplo, para ello llamamos a *make* de la siguiente forma: `make ejemplos`.

El principal problema de nuestra implantación es que debido a que el reloj interno del simulador no se conoce, perdemos la noción del tiempo que tiene el sistema operativo de la tarjeta y en consecuencia no podemos determinar con exactitud cuando se ha terminado una transmisión (existe un *timeout* para el sistema host, pero no sabemos cuanto tiempo el microcontrolador cree que ha pasado).

La solución que implantamos consiste en que el simulador de la tarjeta espera un cierto tiempo que normalmente es suficiente para que la transmisión haya terminado. La desventaja de esto es que siempre tenemos que esperar de más y la simulación se hace un poco más lenta, que de por sí ya es tardada, sobretodo al utilizar el *display* o el depurador.

4.8. La *API PC/SC* y su Simulación

4.8.1. Descripción de *PC/SC*

Hasta hace unos cuantos años no existía una interfaz unificada para comunicarse con los lectores de tarjetas (y por consiguiente con las tarjetas) de diferentes fabricantes. Esto ocasionaba problemas severos, como por ejemplo, si teníamos una aplicación funcionando correctamente y cambiábamos de lector de tarjetas, entonces nuestra aplicación no funcionaría más.

Para impulsar el desarrollo de las tarjetas inteligentes es muy importante establecer alguna interfaz estandarizada de forma que muchos desarrolladores independientes puedan aparecer y entrar en el mercado.

Una primera solución a este problema apareció en Alemania y se llama *Multifunktionales Kartenterminal (MKT)*, también conocida en inglés como *Multifunctional CardTerminal (MCT)*. Entre los puntos que cubre esta especificación se encuentran el del *display* y el del teclado. En realidad para este trabajo de tesis no se exploró esta especificación pero al parecer está en desuso excepto en Alemania.

Más recientemente las principales empresas desarrolladoras de equipo de tarjetas inteligentes se unieron para formar una organización llamada *Pcscworkgroup* y crear una especificación que asegurara interoperabilidad de las tarjetas y lectores a través de la misma interfaz, la cual es independiente de plataforma a pesar de que uno de los miembros de dicha organización es *Microsoft*.

El resultado de este esfuerzo se denominó *PC/SC* (computadora personal a tarjeta inteligente) y lo más importante de un sistema que implemente esta *API* es el gestor de recursos. Las tareas principales del gestor de recursos son: conectarse y transmitir datos a la tarjeta mediante al menos los protocolos *T=0*, *T=1* y sin procesar o crudo (*raw*), controlar el acceso a las terminales por una o más aplicaciones y poder acceder a las funciones particulares de cada lector. No se entrará en más detalles sobre este tema, pero puede obtenerse más información en la página oficial de *Pcscworkgroup* [43]

Movimiento *MUSCLE*

Como ya se mencionó *PC/SC* tiene cierta influencia de *Microsoft* porque dicha empresa es uno de los miembros del grupo. Esto lo podemos constatar al observar las definiciones de la *API* para comunicarnos con el manejador o gestor de recursos, sin embargo *PC/SC* es perfectamente independiente de plataforma,

lo cual podemos comprobar dado que tenemos implantaciones de *PC/SC* para *Machintosh* y *Linux* (o *Unix* en general).

En *Linux* tenemos *PC/SC* gracias al proyecto *MUSCLE*, que proporciona un manejador de recursos de las tarjetas, herramientas y un ambiente de trabajo libre para desarrollar aplicaciones de tarjetas inteligentes (*Musclecard*) entre otras cosas. En el sitio oficial del proyecto *MUSCLE* [32] tenemos una serie muy diversa de aplicaciones libres como un visor de información para las tarjetas de salud de Alemania y un programa para manejar la información de las tarjetas *GSM*.

El administrador de recursos de las tarjetas es proporcionado por *pcsc-lite*, el cual contiene a un demonio llamado *pcscd* que se encarga de coordinar la comunicación con los lectores y tarjetas. En teoría es posible migrar de *Windows* a *Linux* y viceversa porque la *API* no cambia.

Instalación de *PC/SC*

Dado que *PC/SC* es un componente necesario para utilizar las tarjetas inteligentes de acuerdo al *software* desarrollado para este trabajo de tesis, veamos cómo se instala *pcsc–lite*, el *driver* de los lectores disponibles en el laboratorio de seguridad de la información (los *Phillips PE112*) y cómo se configura *pcscd* para trabajar con estos lectores.

Lo primero es bajar el código fuente de *pcsc–lite* y del *driver*. Esto lo podemos hacer directamente de la página de *MUSCLE*. Debemos tener los archivos `pcsc-lite-1.1.1.tar.gz` y `dlr_ifd-drv-0.0.5.tar.gz`. Después continuamos con la instalación de la siguiente forma:

```
[eban@power tesis]$ tar -xzvf pcsc-lite-1.1.1.tar.gz
[eban@power tesis]$ cd pcsc-lite-1.1.1
[eban@power pcsc-lite-1.1.1]$ ./configure
[eban@power pcsc-lite-1.1.1]$ make
[eban@power pcsc-lite-1.1.1]$ su root
[root@power pcsc-lite-1.1.1]# make install
```

Ahora instalamos el driver del lector:

```
[eban@power muscle]$ tar -xzvf dlr_ifd-drv-0.0.5.tar.gz
[eban@power muscle]$ patch -p0 < dlr_ifd-drv-0.0.5-parche
[eban@power muscle]$ cd dlr_ifd
[eban@power dlr_ifd]$ make
```

Como puede observarse previo a la compilación es necesaria la aplicación de un parche debido a que el *driver* disponible es algo viejo (de mediados de 1999) y la interfaz para comunicarse con *PC/SC* ha cambiado ligeramente. Afortunadamente este parche es extremadamente simple. Puede ser necesario ajustar los `include` de acuerdo a donde hayamos instalado *pcsc–lite*.

Si el proceso se ha ejecutado exitosamente debemos tener un archivo nuevo `IFD_Handler_DLR.o`, el cual es el *driver* del lector. Para indicarle a *pcscd* que utilice este *driver* podemos ocupar la herramienta de configuración *installifd* que viene dentro de *pcsc–lite*:

```
[eban@power dlr_ifd]$ su root
[root@power dlr_ifd]# installifd
Please enter a friendly name for your reader (50 char max)
-----> phillips
Please enter the full path of the readers driver (75 char max)
```

```
Example: /usr/local/pcsc/drivers/librdr_generic.so
-----> /home/eban/tesis/dlr_ifd/IFD_Handler_DLR.o
Please select the I/O port from the list below:
-----------------------------------------------
1) COM1 (Serial Port 1)
2) COM2 (Serial Port 2)
3) COM3 (Serial Port 3)
4) COM4 (Serial Port 4)
5) LPT1 (Parallel Port 1)
6) USR1 (Sym Link Defined)
-----------------------------------------------

Enter number (1-6): 2


Now creating new /etc/reader.conf:
```

Esto nos debe producir un archivo de configuración `/etc/reader.conf` que podemos ajustar manualmente en caso de ser necesario. Por supuesto que si el *driver* se encuentra en otra ubicación o si estamos utilizando otro puerto serial distinto a *COM2* debemos proporcionar esta información adecuadamente. Eso es todo, ahora podemos iniciar el manejador de recursos preferiblemente de la siguiente forma: `pcscd -d stdout`, para lo cual necesitamos privilegios de *root*.

4.8.2. Simulación de la *API* de *PC/SC*

Para nosotros era importante comenzar a trabajar con el sistema operativo de la tarjeta antes de que nos llegara la arquitectura real, así que como complemento a nuestra simulación del lector de tarjetas también simulamos la interfaz de *PC/SC*. Mediante nuestro sistema podemos probar muchas de las aplicaciones que utilizan la *API* de *PC/SC* con la ventaja de no tener que gastar en el *hardware* real.

Hay varias formas de simular la *API*. Pudimos haber optado por dar compatibilidad total a nivel de los archivos ejecutables, pero para ello tendríamos que crear nuestro propio *demonio* e implantar su protocolo de comunicación, lo cual sería una tarea algo laboriosa y requeriría investigar más a fondo el funcionamiento interno del *demonio* de *PC/SC*.

En lugar de ello decidimos dar compatibilidad a los programas a nivel del código fuente e incluso a nivel del código objeto antes de ser ligado para generar el ejecutable. Para utilizar la simulación, el programador lo único que tiene que hacer es ligar su código objeto con nuestras librerías en lugar de con las librerías estándares de *PC/SC*. Esto quiere decir que el esfuerzo para utilizar y cambiar de la simulación de la arquitectura a la arquitectura real, es mínimo.

En realidad la simulación tiene algunas limitaciones, pero en la mayoría de las aplicaciones no tendrán importancia. Nuestras librerías fueron probadas por un par de programas diseñados independientemente y el funcionamiento de la simulación mostró ser el correcto.

Para dar la compatibilidad a nivel de código fuente, copiamos las definiciones de las funciones proporcionadas en las librerías de *PC/SC*, tal cuales, a nuestro simulador. Claro que la implantación de estas funciones es lo que hace la diferencia. Tratamos de implantar la lógica de cada una de ellas para regresar los códigos de error o éxito correspondientes e hicimos todo lo necesario para que desde el punto de vista del programa que las utiliza todo fuera transparente e idéntico, tal como si utilizara el *demonio* real.

Para instalar esta librería hicimos también un *Makefile*:

```
[eban@power pcsc_simulador]$ make
[eban@power pcsc_simulador]$ su root
Password:
[root@power pcsc_simulador]# make install
```

Ahora para utilizar la simulación de la *API* de *PC/SC* en lugar de emplear la librería `pcsclite` utilizamos las librerías: `pcsclite_sim` y `lector_simulador` al momento de ligar el programa.

La simulación solamente proporciona un lector, llamado *simulador_avr*. La primera vez que se consulta su estado, tendremos que no hay ninguna tarjeta presente y después la consulta indicará que la tarjeta ha sido insertada. El lector y la tarjeta a su vez están siendo simulados mediante *lector_simulador*.

Para especificar la remoción de la tarjeta se necesita alguna vía alterna de comunicación (de lo contrario estaríamos perdiendo compatibilidad), esta operación requiere que nos comuniquemos directamente con las rutinas del simulador del lector, pero no encontramos una forma sencilla de hacerlo así que la remoción de la tarjeta no está implantada.

En los subdirectorios: `ejemplo1` y `ejemplo2`, tenemos dos aplicaciones que funcionan correctamente utilizando esta simulación y que no fueron desarrollados por nosotros sino que vienen en la distribución estándar de *pcsc–lite*. Solamente creamos los *Makefiles* para modificar el ligado pero todo el código fuente permaneció intacto.

4.9. Integración de las Herramientas de Desarrollo

En esta sección vamos a hablar de las herramientas de desarrollo que podemos ocupar para implantar o mejorar el sistema operativo de las tarjetas inteligentes. Nos enfocaremos especialmente en distinguir las ventajas de utilizar la simulación de la arquitectura mediante herramientas públicas y las desarrolladas para este trabajo de tesis sobre el desarrollo tradicional sin simulación.

Implantación sobre la Arquitectura Real

Después de haber cargado el sistema operativo a la tarjeta, no hay una manera sencilla de determinar qué operaciones se encuentra realizando. No es posible saber el contenido de la memoria ni que lógica está siguiendo el programa. El sistema operativo es compacto, pero rastrear la localización de un posible error con tan solo la salida del puerto serial es sumamente difícil. Además, de acuerdo a nuestra experiencia, los problemas de apuntadores que corrompen algunas áreas de memoria destruyen muy frecuentemente el flujo normal del programa. En realidad no tenemos claro el mecanismo con el que el compilador procesa todas las llamadas a funciones pero al parecer muchas veces el código resultante carga directamente el contador de programa con la dirección deseada a través del índice Z, al cual a su vez se le carga un dato de memoria. Si este dato en memoria se corrompe, el programa hará cosas impredecibles.

El desarrollo del sistema operativo no tiene una verdadera retroalimentación con el programador para que éste lo depure. Conocemos tres formas de retroalimentación muy limitadas que describiremos a continuación:

- *A través de emuladores*. Estos dispositivos de *hardware* son en general caros y no está claro de que forma podríamos utilizarlos en conjunción, por ejemplo, con *PC/SC*.
- *Mediante tarjetas con leds*. La información que nos puede proporcionar el sistema operativo a través de los *leds* es muy limitada y hay dificultades prácticas en su interpretación dada la frecuencia del reloj interno.

- *Ejecutando el programa como si la arquitectura real fuera la PC*. Este es el enfoque utilizado tradicionalmente por *SOSSE*. Nos puede ayudar a depurar la lógica del programa, pero los problemas que pudiéramos encontrar causados por características intrínsecas de la arquitectura no serían detectados.

Herramientas para la Implantación sobre la Arquitectura Simulada

Mediante la simulación de la arquitectura tenemos mucho mayor control sobre el sistema operativo. Primero que nada hay que recalcar que trabajamos exactamente con los mismos archivos binarios que serían cargados en la tarjeta, segundo tenemos la posibilidad de visualizar en todo momento los registros y tenemos la posibilidad de depurar el programa con un depurador estándar.

El depurador *avr–gdb* se comunica con el simulador del microcontrolador y funciona tal como funciona *gdb* común sobre nuestra PC. El usuario puede establecer *breakpoints*, desensamblar la memoria, etc.

Con `lector_simulador` tenemos control de que es lo que exactamente está leyendo y escribiendo el sistema operativo. Con `pcsc_simulador` podemos comunicarnos con la tarjeta inteligente mediante una *API* más conveniente y que también podemos utilizar con la arquitectura real.

Para probar los comandos de *SOSSE_AES* (o *SOSSE_AES_CCM*), que es nuestra versión de *SOSSE* que utiliza al *AES* como algoritmo de cifrado para la autenticación fuerte y para la generación de números pseudoaleatorios, creamos una herramienta llamada *control_sc* con la cual podemos ejecutar la mayoría de las funciones de la tarjeta como lo son las operaciones sobre archivos y la autenticación interna y la externa. Como prueba de la efectividad de nuestra simulación de *PC/SC*, *control_sc* trabaja indistintamente con la arquitectura real o con la simulación, solamente hay que ligarlo con las librerías adecuadas.

Comparación del Sistema Simulado con el Real

Obviamente la simulación también presenta sus desventajas. La más importante es la velocidad de ejecución. Todo el procesamiento de la simulación hace que una operación que normalmente es muy rápida, como la autenticación externa, tome al rededor de un minuto en nuestro equipo. Otra limitación es que no podemos simular la memoria *EEPROM* externa, pero afortunadamente el espacio libre en la memoria interna nos permite probar todas las funciones del sistema de archivos y no deberíamos encontrar diferencias en el comportamiento del sistema cuando probemos la arquitectura real. Otra desventaja es que seguramente versiones posteriores de *SOSSE* tendrán soporte para la generación de números aleatorios con base en los retardos en la comunicación, los cuales tienen cierto componente aleatorio. En la simulación, al menos por ahora, no podemos variar estos tiempos de retardo.

Las desventajas presentadas en nuestra opinión son menores comparadas con las ventajas que nos ofrece la simulación, por lo que la preferimos ampliamente para el desarrollo del sistema.

4.10. Extensión del Sistema Operativo con Nuestra Implantación del *AES*

La distribución estándar de *SOSSE* (al menos hasta la última versión disponible en este momento) solamente soporta un algoritmo criptográfico que en realidad no es muy conocido y se llama *TEA* (*Tiny Encryption Algorithm*). A nosotros nos interesa aplicar el nuevo *AES* a las tarjetas inteligentes, así que tuvimos que adaptar el sistema operativo para que ahora utilice nuestra implantación.

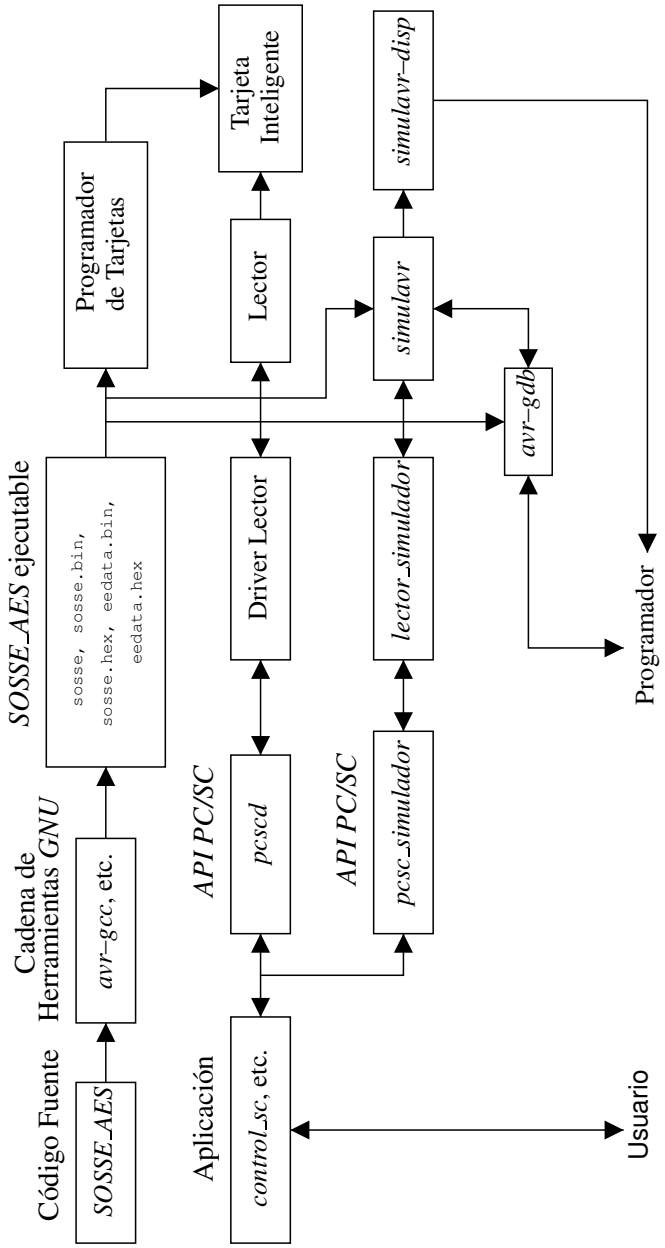

Figura 4.9: Herramientas para trabajar con la arquitectura real y con la arquitectura simulada.

Deseamos mantener compatibilidad con el código fuente original, pero éste no tiene contemplada la posibilidad de utilizar otros algoritmos de cifrado, así que decidimos modificarlo y hacerlo más general de forma que la siguiente persona que desee integrar otro algoritmo criptográfico pueda hacerlo más fácilmente.

Con el código fuente resultante es posible generar los códigos objeto que utilicen el *AES* o bien el *TEA* con simplemente llamar al *Makefile* adecuado. No hay necesidad de utilizar ambos simultáneamente y además no sería esto posible por la cantidad de memoria de programa que queda libre.

La integración del *AES* en *SOSSE* requirió superar las diferencias que presentan la función de cifrado del *AES* y la de *TEA* por lo que este proceso no fue tan directo como habíamos pensado. En primer lugar *TEA* tiene un tamaño de bloque de 8 bytes mientras que en el *AES* es de 16 bytes. Además en *TEA* la clave y el texto en claro / criptograma se interpretan como palabras de 32 bits y en el *AES* se tienen arreglos de 16 bytes. Otra diferencia está en la forma en que se trata la clave y el mensaje en nuestra implantación del *AES*. Por razones de eficiencia la clave debe seguir inmediatamente al texto en claro y ambos deben tener una alineamiento de 16 bytes. En la implantación de *TEA* la posición de la clave y el mensaje son independientes y no tienen restricciones.

La descripción de *TEA* es muy simple, pero ocupa operaciones con palabras de 32 bits y el número de iteraciones es relativamente alto (16). Comparando ambas implantaciones, empíricamente y sin hacer mediciones precisas, podemos decir que nuestra implantación es más rápida pero ocupa más memoria de programa.

Los algoritmos de cifrado de bloques son utilizados por el *COS* con tres propósitos fundamentales:

- *Autenticación externa*. La tarjeta comprueba que el *host* ha cifrado el reto adecuadamente (demostrando que tiene el secreto).
- *Autenticación interna*. El reto del *host* es cifrado por la tarjeta para demostrar su identidad con la clave secreta destinada para este fin.
- *Generación de números pseudoaleatorios*. Se mantiene un estado el cual es alterado mediante el cifrador de bloques. Estos números son utilizados para generar el reto de la autenticación externa.

Nuestra nueva versión de *SOSSE* puede compilarse de tres formas distintas.

- *Versión Tradicional*. Equivale a la versión original de *SOSSE* (antes de que lo modificáramos).
- *Versión AES*. Utiliza nuestra implantación del *AES* para realizar la autenticación interna y externa.
- *Versión AES+CCM*. Además de la funcionalidad de la versión anterior añadimos dos comandos que permiten hacer lecturas asegurando la integridad y la confidencialidad de la información entre la tarjeta y el *host*.

Para compilar la versión *AES* de *SOSSE* ejecutamos el siguiente comando:

```
[eban@power src]$ make -f Makefile.AES
```

La versión *AES+CCM* puede obtenerse mediante:

```
[eban@power src]$ make -f Makefile.AES_CCM
```

Estos *Makefiles* se encargan de compilar y ligar el código correspondiente a nuestra implantación. Los archivos resultantes `sosse.hex` y `eedata.hex` están listos para ser cargados a la tarjeta, aunque todavía pueden ser modificados para establecer claves diferentes.

Cabe señalar que para que la compilación sea exitosa necesitamos definir los valores adecuados en el archivo de configuración `config.h`. Para facilitar esto se proporciona un archivo de configuración predefinido para cada una de las versiones.

Gracias al conocimiento que ganamos del sistema operativo pudimos implantar nuevos comandos pensados en incrementar la flexibilidad y seguridad del sistema. El esquema tradicional de transferencia de información entre el *host* y la tarjeta de acuerdo a *T=0* y a la versión original de *SOSSE* pasa toda la información en claro y no hay ninguna medida que asegure su confidencialidad ni su integridad. Decidimos modificar la función que se encarga de leer los datos de un archivo en la tarjeta y de transmitirlos al

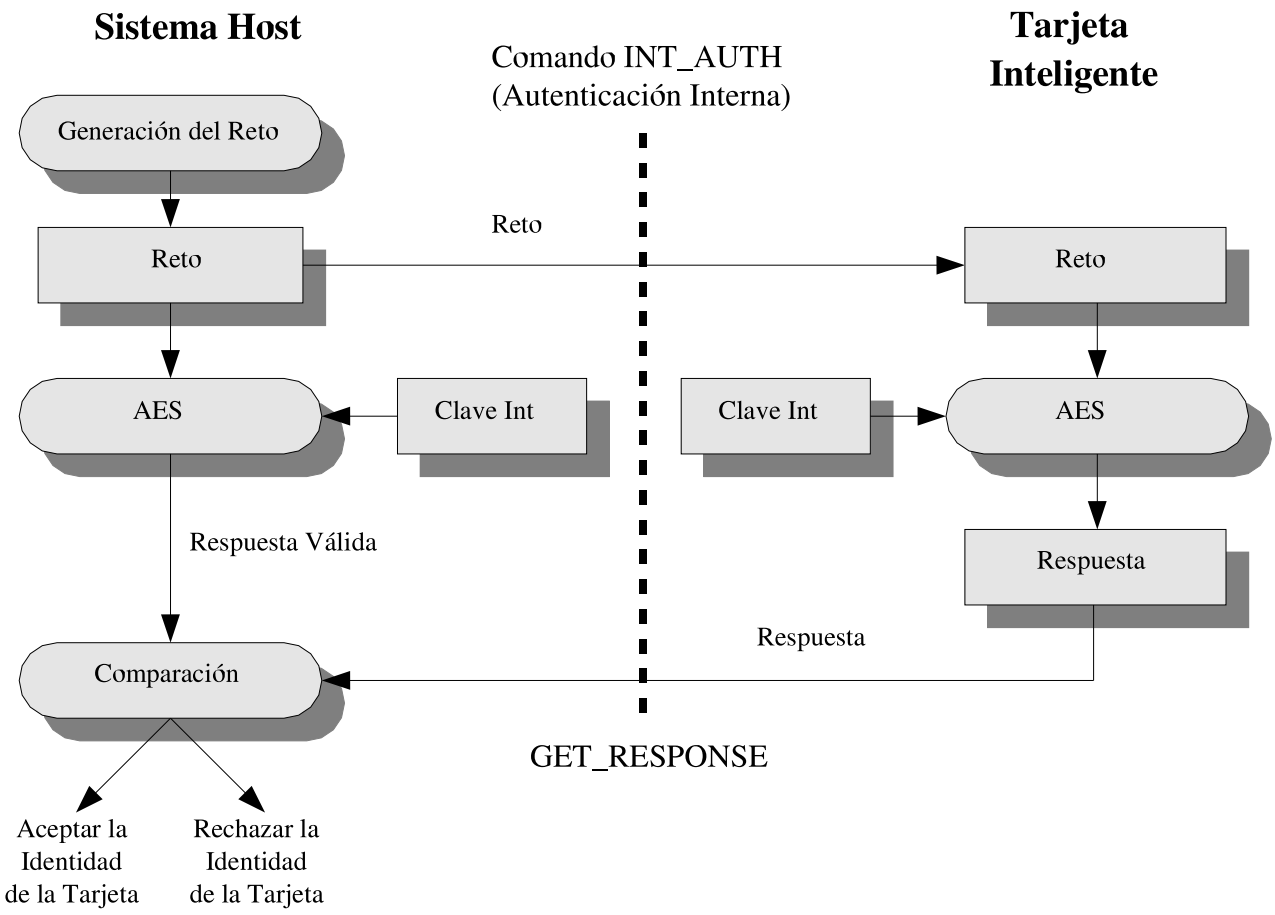

Figura 4.10: Diagrama que muestra el protocolo de autenticación de la tarjeta con el *host*.

sistema *host* a través del lector. Nuestro objetivo era aplicar un algoritmo de cifrado de alguna forma que no se había utilizado en nuestras tarjetas con miras en mejorar la seguridad.

Para lograr este objetivo escogimos el modo de operación *CCM* debido a que conjunta un código de autenticación de mensajes con el cifrado de los datos. En lugar de implantar dos modos de operación independientes, decidimos implantar un modo que está pensado para realizar ambas tareas con poca sobrecarga.

Los mensajes cortos típicos de las tarjetas inteligentes no se prestan para utilizar los códigos de autenticación de mensajes más comunes ya que éstos típicamente aumentan el tamaño del criptograma en al menos un bloque (16 bytes), lo cual es mucha sobrecarga para la vía de comunicación. El modo *CCM* permite seleccionar la cantidad de bytes que ocupará el *MAC*. En nuestro caso decidimos utilizar 4 bytes, pero de ser necesario este número puede incrementarse fácilmente hasta 16 bytes tomando cada uno de los valores pares intermedios.

La comunicación cifrada nos puede proporcionar confidencialidad sobretodo cuando no confiamos en el sistema *host* o el lector y deseamos transferir información a otro sistema. El algoritmo de cifrado "garantiza" que para descifrar los paquetes transmitidos se necesita poseer la clave correspondiente y otra información llamada *nonce*, que es un arreglo de bytes que nunca debe reutilizarse con la misma clave.

El modo *CCM*[8] consiste a grandes rasgos en la unión del modo de operación contador y el *CBC–MAC*.

Mediante el modo contador (ver sección 2.2.1, página 42) se logra la confidencialidad eficientemente, pero este modo por sí solo es muy susceptible a la pérdida de la integridad puesto que un error *e* en el criptograma se traduce directamente en un error *e* en el texto en claro recuperado. El modo *CBC–MAC* (ver sección 2.3.3, página 52) cubre las limitaciones del modo contador.

Para el funcionamiento correcto del modo *CCM* es necesario preestablecer ciertos parámetros. El primero es la longitud del campo de autenticación *M* que como ya mencionamos es de 4 bytes, el segundo es la cantidad de bytes *L* con los que se expresa la longitud de un mensaje. En nuestro caso $L = 2$ bytes. La especificación concreta de este algoritmo la podemos encontrar en [42].

Las características de este modo de operación son excelentes en cuanto a la sobrecarga de la transmisión de datos, pero creemos que es posible reducir la sobrecarga de cálculos ya que para cifrar y autenticar los mensajes se realiza una doble "pasada", es decir se efectúa al rededor de dos veces el algoritmo de cifrado por cada bloque de texto en claro. Sabemos que existen algoritmos que proporcionan integridad y confidencialidad con una sola "pasada" pero no encontramos ninguna especificación abierta disponible que pudiéramos utilizar.

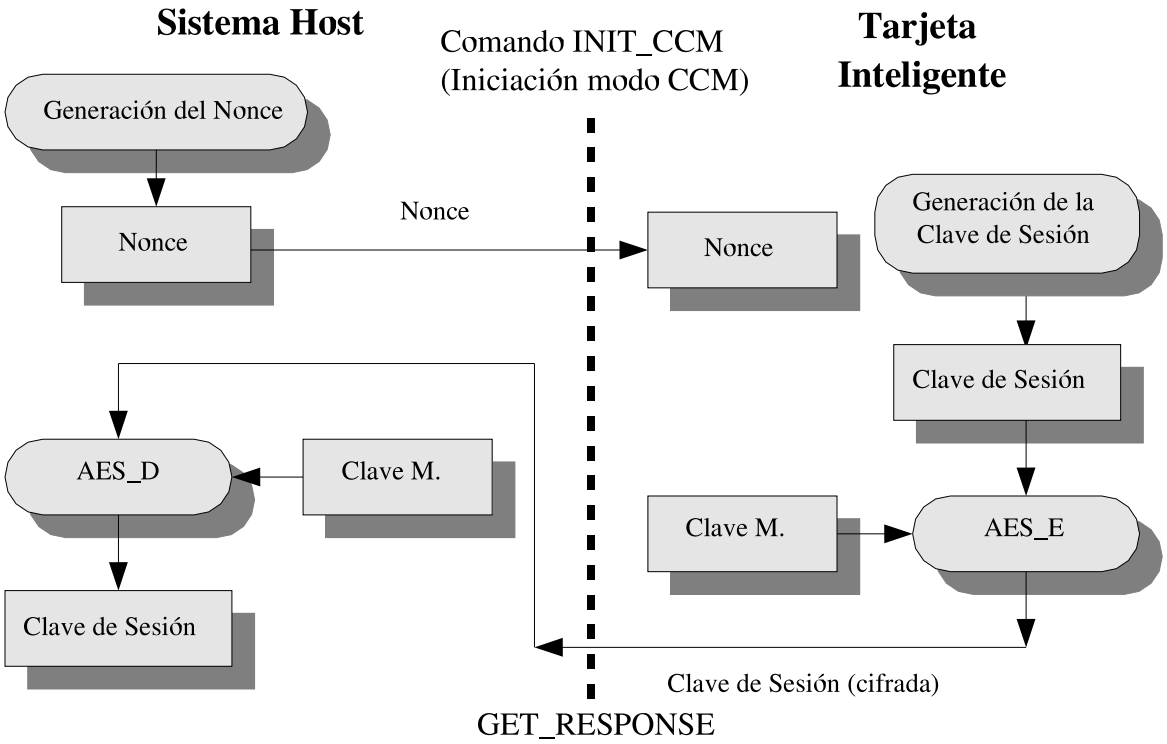

Figura 4.11: Generación e intercambio de la clave de sesión y el *nonce* mediante el comando INIT_CCM.

La lectura de información de la tarjeta a través del modo *CCM* fue implantada en su totalidad en lenguaje *C* y llama a nuestra implantación del *AES* en ensamblador cuando es necesario. No tendría mucho caso programar esta rutina en ensamblador porque la ganancia en velocidad y uso de las memorias

[8] *Counter* with *CBC–MAC*.

no sería significativa, además se podría perder legibilidad en el código y esto podría propiciar la presencia de errores de programación.

Antes de iniciar la transferencia de datos es necesario establecer una clave y un *nonce*. Para ello creamos el comando `INS_INIT_CCM`. La tarjeta se encarga de generar mediante su *PRNG* una clave de sesión (16 bytes) y el *host* genera el *nonce* de 13 bytes. La clave de sesión es transmitida mediante un protocolo de transporte de clave punto a punto de una pasada [30, p. 497] que consiste en cifrar la clave de sesión r_A con otra clave k previamente acordada. La clave k es programada directamente en la tarjeta y debe proporcionarse mediante una vía de comunicación segura a los sistemas *host* involucrados. El *nonce* no necesita ser secreto así que se transmite en claro.

Los datos de un archivo de la tarjeta pueden ser leídos mediante el comando `INS_READ_BINARY_CCM` que reemplaza a `INS_READ_BINARY`.

Es muy importante señalar que si bien mediante nuestra implantación del modo *CCM* podemos asegurar la confidencialidad y la integridad de la información leída de un archivo en la tarjeta hacia el *host*, en realidad no estamos incrementando la seguridad de la tarjeta en lo absoluto (pero tampoco la estamos disminuyendo), puesto que los mecanismos de acceso a los archivos siguen presentes. Un adversario que tenga acceso a la vía de comunicación entre el *host* y la tarjeta puede simplemente escribir cualquier información que desee en al archivo antes de que lo leamos. Puede alterar toda la información en claro y si queremos añadir confidencialidad e integridad a la comunicación es necesario modificar el esquema de seguridad de la tarjeta completo y asegurar la transmisión en ambas direcciones. Esto puede lograrse implantando el comando *envelope*, pero para ello requerimos de tarjetas más potentes puesto que las utilizadas ya no tienen memoria de programa suficiente.

4.11. Sistema de Control y Administración de las Tarjetas

En secciones anteriores se describió el trabajo encaminado al enriquecimiento del sistema operativo de las tarjetas con nuestra implantación de un algoritmo criptográfico nuevo. La gran mayoría de los sistemas basados en tarjetas inteligentes se componen, además de por las tarjetas inteligentes y equipos lectores, por un sistema *host* o anfitrión.

Para que el sistema quede completo es necesario implantar también los sistemas de *software* del sistema anfitrión y en particular sería muy conveniente contar con una herramienta con la cual podamos probar nuestro trabajo en la tarjeta. Nos interesa poder realizar las tareas que soporta el sistema operativo enriquecido y poder realizar ciertas tareas de administración, como por ejemplo: cambiar el PIN y crear archivos.

Deseamos tener un control bastante amplio sobre la mayoría de las funciones disponibles en la tarjeta y la libertad de manipular como nos sea más conveniente la información de la misma. Con esta herramienta podremos facilitar el proceso de personalización de las tarjetas y sobretodo, podemos establecer un conjunto de funciones base para el *host*, con las cuales el realizar una aplicación práctica pudiera ser más sencillo.

Funcionalidad Requerida

Concretamente queremos implantar al menos las 7 funciones principales: autenticación externa, autenticación interna, identificación del usuario mediante un PIN, desbloqueo y cambio del PIN, selección, creación y modificación de archivos tanto elementales como dedicados.

Los comandos mencionados son todo lo que se necesita para implantar una aplicación real, en [22] se presenta el desarrollo de algunas aplicaciones en las que todo el trabajo por el lado de la tarjeta se

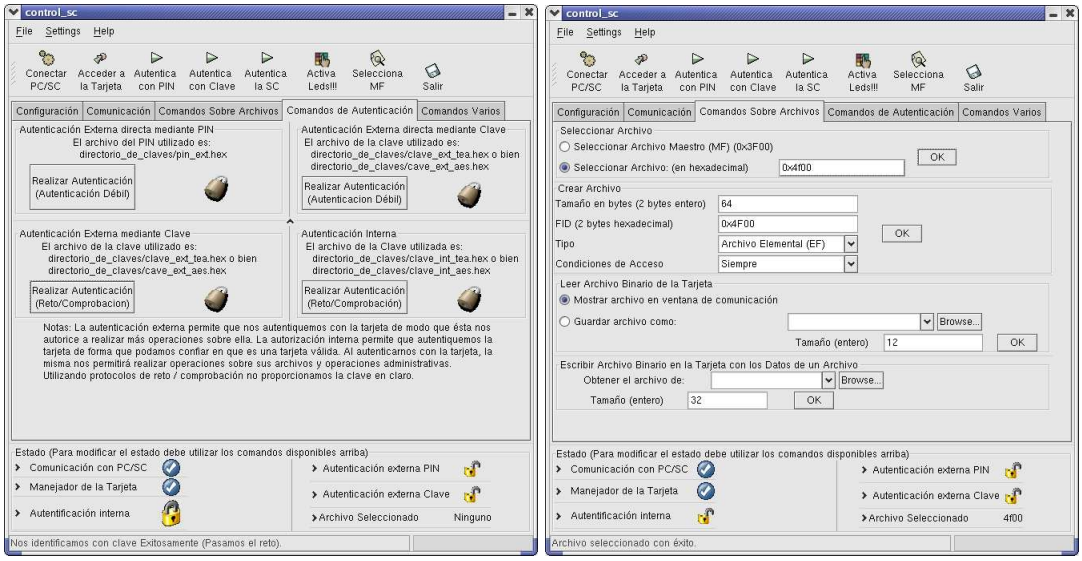

Figura 4.12: La herramienta de control y administración de las tarjetas: *control_sc*.

limita a crear ciertos archivos con un contenido y permisos adecuados. Por el lado del *host* se tienen las funciones que manipulan correctamente dicha información. Claro que aplicaciones más específicas requieren de otros comandos.

Es normal que los usuarios olviden su PIN, y la tarjeta se bloqueará parcialmente cuando el número de intentos fallidos al tratar de proporcionarlo sea alcanzado. No será posible identificarnos de nuevo y la tarjeta quedará prácticamente inutilizable, por ello es necesario que podamos desbloquear y cambiar el PIN pero esta tarea debe poder hacerse solamente con una clave especial llamada PUK y que normalmente permanece secreta para los usuarios.

Uno de nuestros objetivos al implantar este sistema de control es el de mantener compatibilidad con el sistema operativo original que utiliza el algoritmo de cifrado *TEA* mientras que también lo podamos utilizar con el sistema operativo *SOSSE_AES* y *SOSSE_AES+CCM*.

Desarrollo de la Interfaz en *GTK* Utilizando *Glade*

No entraremos en detalles sobre como implantamos el sistema de control *control_sc*, pero al menos mencionaremos algunas partes del proceso. Al programar fuera de la tarjeta inteligente, no tenemos por qué limitarnos tanto en los recursos que ocupan nuestras aplicaciones. Con el fin de facilitar el uso de los comandos y tener un ambiente más ergonómico (desde el punto de vista de *software*) decidimos utilizar *GTK+* para el diseño de la interfaz gráfica. No es sencillo programar directamente con *GTK+* porque hay que recompilar para ver los resultados, por ello utilizamos *glade* que es una herramienta gráfica que genera automáticamente el código en *GTK+* de nuestra aplicación. Gracias a esta herramienta acortamos el tiempo que nos hubiera llevado implantar la interfaz gráfica a mano considerablemente y el resultado final es mucho más agradable puesto que podemos perfeccionarlo a medida que creamos la interfaz.

Glade genera un esqueleto de la aplicación que no hace nada, pero tiene la interfaz gráfica que diseñamos y a la cual podemos añadirle la funcionalidad correspondiente.

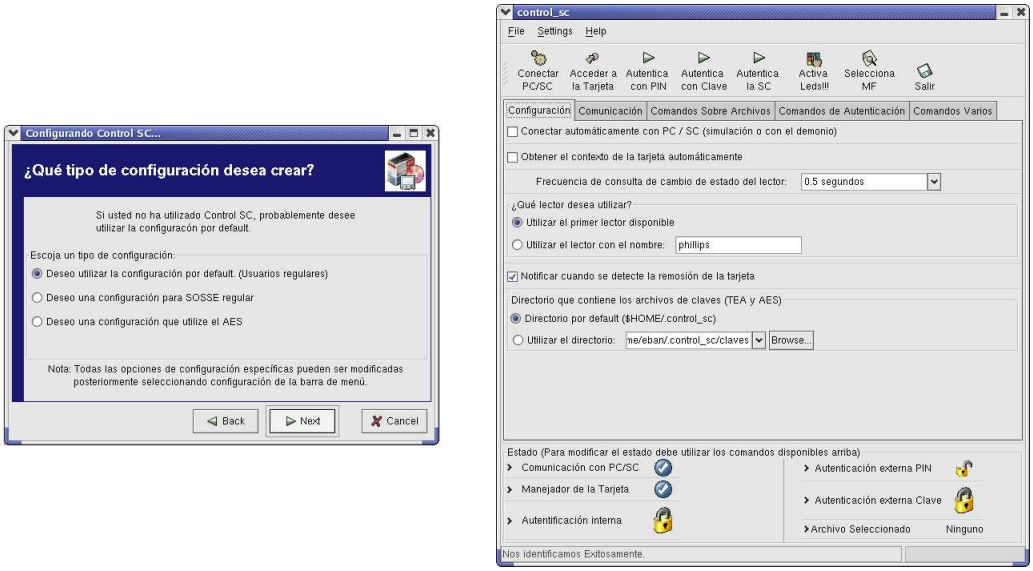

Figura 4.13: Configuración inicial e identificación mediante PIN con *control_sc*.

Implantación de la Aplicación

Se crearon varios módulos formados por un archivo de cabecera con las declaraciones de las funciones y un archivo con su definición. Cada uno tiene un propósito específico y trabajando juntos forman la aplicación de control y administración de las tarjetas:

- *pcsc_funciones*. Son las funciones básicas para establecer la comunicación con el demonio de *PC/SC*, obtener la lista de lectores disponibles, conectarnos con un lector, consultar su estado y mandar información a la tarjeta.

  En teoría una aplicación cualquiera podría utilizar este módulo para acceder de forma más sencilla a *PC/SC*.

- *tajeta_funciones*. Este módulo guarda el estado de la comunicación con la tarjeta y nos evita preocuparnos por detalles como qué algoritmo criptográfico estamos utilizando, saber el formato de los comandos que se mandan a la tarjeta e interpretar las respuestas de la misma.

  Las funciones son de propósito general y nos permiten efectuar las operaciones tradicionales de la tarjeta como seleccionar archivos y autenticarnos. En teoría este módulo debe ser reutilizable.

- *pcsc_interfaz*. Aquí tenemos algunas funciones específicas a nuestra aplicación con las cuales podemos guardar o recuperar la configuración del programa y reflejar esta información en el ambiente gráfico.

- *pon_estatus*. Se trata de funciones que cada aplicación que utiliza *pcsc_funciones* debe rescribir puesto que con ellas se da retroalimentación al usuario.

  Para esta aplicación estas funciones modifican el mensaje de la *barra de aplicación* en la parte de abajo de la ventana principal y muestran la información transmitida desde y hacia la tarjeta en el marco correspondiente.

- *callbacks*. *Glade* crea este archivo para manejar las señales generadas por los *widgets* cuando ocurre algún evento. Nosotros implantamos las rutinas con las que queremos que el programa responda a estas señales.

- *interface*. Este módulo es generado automáticamente por *Glade* y se encarga de crear las interfaces gráficas y conectar las señales de acuerdo al diseño que hemos hecho previamente.

- *Módulos criptográficos*. Contienen las rutinas de los algoritmos de cifrado *TEA* y el *AES*. Utilizamos la implantación de *TEA* que está incluida en el código original de *SOSSE*. Nuestra implantación del *AES* obviamente no funciona para la *PC*, así que utilizamos una implantación optimizada hecha por *Vincent Rijmen, Antoon Bosselaers y Paulo Barreto*, la cual puede conseguirse en la página principal de *Rijndael*.

4.12. Creación y Empleo de una Biblioteca para Manejar las Estructuras Algebraicas de *Rijndael*

Al trabajar con el *AES*, la especificación y el libro del diseño de *Rijndael* son muy útiles para implantar eficientemente el algoritmo, sin embargo para su estudio es conveniente contar con herramientas que nos permitan realizar las transformaciones del algoritmo de forma similar a como han sido descritas matemáticamente.

Todas las operaciones en *Rijndael* involucran algún campo de *Galois* de característica 2, ya sea *GF(2)* o *GF(256)*. Las distintas transformaciones pueden representarse como operaciones modulares con polinomios sobre alguno de estos campos o bien como una inversión en *GF(256)*.

A nosotros nos pareció interesante, el como a partir de una estructura tan simple como *GF(2)* puede construirse un campo extendido *GF(256)* (mediante polinomios con coeficientes sobre *GF(2)*). De ahí podemos obtener polinomios con coeficientes en *GF(256)*. De esta forma una transformación como *MixColumns* involucra distintas estructuras algebraicas.

Hay información disponible sobre los pasos inversos, y hay tablas en las que se muestra como se debe realizar el mapeo para la transformaciones *affine* y la inversión en *GF(256)*, pero nos gustaría poder corroborar esta información con nuestros propios resultados.

Intentamos experimentar con las transformaciones de *Rijndael* con *Matlab* pero el soporte para operaciones sobre campos que encontramos era limitado. En particular tuvimos dificultad para expresar los polinomios sobre campos y su aritmética modular por lo que decidimos crear nuestras propias herramientas para experimentar con las operaciones matemáticas de *Rijndael*.

Estructura Algebraica Base

Todas las estructuras utilizadas en nuestra herramienta para trabajar con las estructuras de *Rijndael* (campos y anillos conmutativos) poseen algunas características comunes. Además para implantar la aritmética de polinomios es necesario establecer también una aritmética de sus coeficientes, que requiere tener ciertas operaciones disponibles como la multiplicación y la adición.

Debido a lo anterior todas las estructuras algebraicas de nuestro conjunto de herramientas se derivarán de una clase base virtual que nos indica que operaciones deben implantar.

Entre los métodos que deben implantar las clases derivadas están la adición, substracción, multiplicación y división. También deben proporcionar información sobre su orden (número total de elementos que integran la estructura) y una representación de cada elemento como entero.

Si bien es posible *instanciar* esta clase, no debe hacerse puesto que no se tiene todavía ninguna funcionalidad útil.

Implantación de Polinomios con Coeficientes sobre la Estructura Base

Los polinomios son muy utilizados en *Rijndael* y hay de diversos tipos que se distinguen por tener coeficientes sobre distintos campos y efectuar aritmética con módulos diferentes. En lugar de crear una estructura algebraica particular para cada caso decidimos crear una estructura más general capaz de utilizar coeficientes y módulos prácticamente arbitrarios. Los coeficientes del polinomio y del módulo pueden ser cualquier clase derivada de la estructura base incluyendo a los polinomios mismos.

Las operaciones con polinomios se calculan tal y como estamos acostumbrados, pero al momento de realizar las operaciones entre los coeficientes, se ejecutan las correspondientes dependiendo de la clase de la que se trate (se emplea polimorfismo).

Se agregaron algunas variables porque nos interesa conocer el grado del polinomio y el módulo (que se utiliza al hacer las multiplicaciones). También son útiles las rutinas que nos permiten desplegar los polinomios mediante sus coeficientes y mediante una representación entera.

El Algoritmo Extendido de *Euclides*

El algoritmo extendido de *Euclides* nos permite resolver ecuaciones de la forma: $a \cdot x + b \cdot y = gcd(a,b)$ donde x, y y $gcd(a,b)$ (el máximo común divisor de a y b) son desconocidos. El algoritmo puede ser empleado directamente con números enteros pero también es posible aplicarlo a otras estructuras. Hay ciertas condiciones que se deben cumplir para que sea aplicable pero daremos por hecho que se cumplen. La razón por la que implantamos este algoritmo es porque nos permite invertir elementos (calcular inversos multiplicativos) con aritmética modular. No entraremos en detalles de como puede realizarse esta operación, pero puede observarse en el código fuente.

Implantación de *GF(2)* y *GF(256)*

La estructura más simple que nos es de utilidad es *GF(2)*. Hay que recordar que en este campo tenemos solamente dos elementos, el uno y el cero (hay métodos para obtener estos elementos). En la práctica la multiplicación equivale a una función lógica AND y la adición, como es módulo 2, a una XOR.

Esta estructura base nos sirve como pilar para construir otras más complejas.

La estructura $GF(256)$ es extremadamente importante para *Rijndael*. Como ya se mencionó en otro capítulo, todos los campos de *Galois* del mismo orden son isomórficos. Una forma de representar este campo es mediante polinomios de grado menor o igual a 7 con coeficientes sobre *GF(2)* y con aritmética módulo un cierto polinomio de grado 8. En *Rijndael* dicho polinomio es $m(x) = x^8 + x^4 + x^3 + x + 1$, que puede representarse en hexadecimal como 0x11b, o en decimal como el número 283.

Para dar una mayor generalidad nosotros dejamos flexible el módulo, aunque es común que trabajemos con 283. Esta estructura difiere de la generada con polinomios sobre *GF(2)* en que la división fue implantada como la multiplicación por el inverso multiplicativo, mientras que en el caso de los polinomios, la división es la división tradicional de polinomios que puede producir un residuo.

El conjunto de herramientas diseñado consta de una librería que está acompañada de varios programas que ejemplifican su uso haciendo alguna tarea útil. La instalación de la librería se hace de la siguiente forma:

```
[eban@power estructuras]$ make
[eban@power estructuras]$ su root
Password:
[root@power estructuras]# make install
[root@power estructuras]# exit
[eban@power estructuras]$ make ejemplos
```

Programas de ejemplo

Los programas de ejemplo que creamos son los siguientes:

- *invGF256*. Genera una tabla con los inversos multiplicativos en *GF(256)*.
- *euclides_enteros*. Aplica nuestra implantación del algoritmo de *Euclides* a los números enteros.
- *invAffine*. Obtiene a partir del polinomio factor y el polinomio constante de la transformación *affine* directa (con coeficientes sobre *GF(2)*), los polinomios inversos, factor y constante, de la transformación inversa.
- *invMixColumns*. Dado el polinomio con coeficientes sobre *GF(256)* utilizado en la transformación *MixColumns*, obtiene el polinomio inverso correspondiente que invierte la transformación.
- *pasosRijndael*. Muestra como realizar cada uno de los pasos de *Rijndael* que alteran al estado tanto para el cifrado directo como para el inverso, excepto que no se muestra como realizar la suma de la clave de ronda ni la planificación de la clave.
- *lagrange*. Este programa produce una representación polinomial de la caja S. Se trata de un polinomio con coeficientes sobre *GF(256)* y que se obtiene mediante la interpolación de *Lagrange*. Concretamente coincidimos con el resultado dado en [13, pp. 212]
- *invLagrange*. Genera la representación mediante la interpolación de *Lagrange* de la caja S inversa.
- *raicesMixColumns*. Obtiene las raíces del polinomio $c(x)$ del paso *MixColumns*.

Para ilustrar lo conveniente que es utilizar nuestra librería, a continuación presentamos un fragmento del código de `invGF256.cpp` que calcula y muestra los inversos de todos los elementos en el campo $GF(256)$ utilizado en *Rijndael*, obsérvese lo compacto del código y que el módulo (283) pudo haber sido modificado fácilmente:

```
cout<<"Calculando los inversos multiplicativos en GF(256) de Rijndael"<<endl;
for(i=0;i<256; i++) {
  GF256 algo(283,i);
  algo.Imprime();
  cout << " -> ";
  algo.Inverso();
  algo.Imprime();
  cout << endl;
}
```

Representación de Cada Paso del *AES* de Acuerdo a la Librería

El paso *SubBytes* está constituido por la composición de una inversión en $GF(256)$ y una transformación *affine* de la forma $A \cdot \vec{x} + \vec{b}$, donde $\vec{x}$ y $\vec{b}$ son bytes representados como vectores y M_1 es una matriz binaria de 8x8. La inversión en $GF(256)$ fue ilustrada en el párrafo anterior y es su propia inversa (es una involución).

Para la transformación *affine* la representación común de cada byte del estado es una multiplicación y una suma matricial, pero nosotros los representamos como polinomios de grado menor o igual a 7 con coeficientes sobre $GF(2)$ y la transformación consiste en multiplicar cada polinomio del estado por otro polinomio $h(x) = x^4 + x^3 + x^2 + x^1 + 1$ utilizando el módulo $x^8 + 1$ y después sumar un polinomio constante $c(x) = x^6 + x^5 + x + 1$. Para la transformación *affine* inversa, simplemente multiplicamos y sumamos los polinomios correspondientes: $h^{-1}(x) = x^6 + x^3 + x$ y $c^{-1}(x) = x^2 + 1$. Éstos fueron determinados con el programa *invAffine*.

En cuanto al paso *recorre renglones*, normalmente se especifica mediante un corrimiento distinto para cada renglón del estado. Nosotros con el fin de utilizar nuestra librería lo expresamos mediante una multiplicación, donde cada renglón del estado representa a un polinomio de grado menor o igual a tres y el corrimiento se logra multiplicando a estos polinomios por 1, x, x^2 o x^3 con un módulo $x^4 + 1$. La transformación inversa ocupa los mismos polinomios pero en otro orden: 1, x^3, x^2 y x.

El paso *MixColumns* opera independientemente en cada columna del estado interpretando a esta columna como un polinomio con coeficientes sobre $GF(256)$, multiplicándolo por el polinomio constante $c(x) = 3 \cdot x^3 + x^2 + x + 2$, respecto al módulo $x^4 + 1$, que está especificado en el algoritmo. Para implantar este paso nosotros seguimos exactamente esta representación. El paso inverso involucra el polinomio constante inverso $c(x) = 11 \cdot x^3 + 13 \cdot x^2 + 9 \cdot x^1 + 14$ (expresado con coeficientes en decimal) que podemos obtener mediante el programa *invMixColumns*.

Representación de *SubBytes* Mediante la Interpolación de *Lagrange*

Un resultado que nos pareció interesante fue la forma de expresar el mapeo de la transformación *SubBytes* mediante un polinomio de grado 254 con coeficientes sobre $GF(256)$. Esta representación se obtiene mediante la interpolación de *Lagrange* y llegamos al mismo resultado dado en la bibliografía con el programa *lagrange*.

Gracias al programa anterior fácilmente pudimos expresar la caja S inversa de la misma forma. Para ello escribimos el programa *invLagrange*.

Ambos programas ocupan aproximadamente unos 750 MBytes de memoria *RAM* (aquí ayuda la memoria virtual y el *swap*) y tardan en ejecutarse alrededor de un minuto.

Capítulo 5

Análisis de Resultados

En este capítulo describiremos y explicaremos el producto de este trabajo de tesis desde un punto de vista cualitativo y cuando sea pertinente cuantitativo.

Primero evaluaremos todos los sistemas de *software* que desarrollamos haciendo énfasis en nuestra implantación del algoritmo *AES* para la arquitectura de las tarjetas. Examinaremos otras implantaciones en arquitecturas similares y las compararemos con la nuestra con un enfoque numérico respecto a los recursos computacionales que requieren y aclararemos algunos aspectos técnicos.

El segundo punto en que nos concentraremos será en la nueva funcionalidad con la que enriquecimos el sistema operativo (*COS*). Cotejaremos las características de los comandos mejorados, especialmente en cuanto a tiempo de procesamiento, con los que previamente teníamos disponibles. También determinaremos qué tan bueno es el desempeño de los comandos con relación a lo reportado en la literatura para otras arquitecturas y algoritmos de cifrado (por ejemplo: el *DES*).

Después brevemente mencionaremos las características e importancia de las herramientas auxiliares que creamos para facilitar el proceso de depuración y pruebas. Específicamente comentaremos sobre los simuladores que desarrollamos. También hablaremos sobre nuestra aplicación de control y administración de las tarjetas haciendo hincapié en la funcionalidad que proporciona.

Por otro lado hablaremos sobre nuestra librería para el manejo de las estructuras algebraicas de *Rijndael*, que aunque no está directamente relacionado con la implantación del algoritmo, es una herramienta conceptual para comprender su operación, y en qué consiste cada transformación de acuerdo a su especificación.

Finalmente equipararemos los resultados obtenidos con los objetivos propuestos inicialmente.

5.1. Implantación del *AES*

La necesidad de implantar el algoritmo *AES* nos llevó a tres versiones diferentes. La primera tuvo ante todo el propósito de ayudarnos a comprender el algoritmo desde una perspectiva de la programación y fue codificada en lenguaje *C*. Creemos que cumplió su objetivo ya que fue el primer acercamiento a la optimización del paso *MixColumns* y con ella pudimos derivar las claves de ronda "al vuelo", es decir, al momento de requerirlas. *MixColumns* desde la especificación puede apreciarse como la transformación que es más intensiva computacionalmente y es clave para el desempeño de todo el cifrado. Por otro lado las restricciones de la arquitectura no permiten, o al menos no sería razonable, expandir la clave entera en memoria, así que para meter el algoritmo en la tarjeta era prácticamente obligatorio hacer la planificación de clave de esta forma. Debido a que su principal utilidad no fue brindar un buen desempeño, esta

implantación no se discutirá más.

La segunda versión estaba encaminada a expresar al *AES* de manera eficiente en lenguaje ensamblador. Aquí debemos recordar que el criterio de diseño contemplaba ante todo la posibilidad de convivir con el sistema operativo, luego la reducción del tamaño del código, y después venía la velocidad y el uso de memoria *RAM*. En realidad esta implantación inicialmente era solamente un paso previo a la versión que mejor satisface los requerimientos que establecimos. A causa de sus propiedades, especialmente en cuanto a la velocidad, decidimos reportarla y analizarla con mayor detalle.

La tercera implantación del algoritmo puede considerarse como la mejora de la versión anterior, puesto que se redujo considerablemente el tamaño del código incluyendo algunos ciclos y se reprogramó completamente el paso *mezclar columnas*. Aunque algunas de las transformaciones se compactaron sustancialmente se buscó que la velocidad de cifrado no se perjudicara significativamente.

Antes de presentar información más explícita conviene hacer una reflexión. En la literatura es muy frecuente encontrar implantaciones de cifradores de bloque pero el único parámetro que se busca es la velocidad. Desde un punto de vista meramente académico esto puede ser adecuado, pero desde un enfoque de la ingeniería este factor pasa a segundo plano. Lo más importante es contar con rutinas de cifrado adaptadas a la arquitectura. Aunque un algoritmo funcione en un miembro de la familia de microcontroladores, si no funciona en el que verdaderamente vamos a utilizar cualquier información sobre el desempeño teórico, por muy bueno que sea, se vuelve intrascendente.

No debemos suponer que el algoritmo criptográfico será el único componente que va a trabajar en la tarjeta puesto que hay otras tareas que también necesitan los recursos del sistema. Aunque tengamos suficiente memoria de programa y memoria *RAM* debemos administrarlos juiciosamente, de lo contrario podríamos no contar con aplicaciones o rutinas que hagan uso de nuestra implantación.

El tamaño de clave utilizado fue exclusivamente el de 128 bits, las otras longitudes fueron dejadas de lado puesto que normalmente no se necesitan en las aplicaciones de tarjetas inteligentes. Ahora veamos más minuciosamente las dos últimas versiones mencionadas. Cabe señalar que es posible intercambiar fragmentos de código entre ellas, de manera que podemos crear más versiones y hacer un mejor balance entre el tamaño y el rendimiento. De hecho es sencillo determinar la combinación que ofrece la mayor velocidad de cifrado dado un número máximo de instrucciones. En adelante nos referiremos a la versión 2 como la versión “rápida” y a la versión 3 como la versión “pequeña”.

La información que se presenta a continuación fue obtenida gracias al simulador (que mejoramos) y al depurador. Directamente examinamos el flujo del programa para determinarla. Es preferible tratar los siguientes valores como aproximaciones pero son datos muy precisos. Para entenderla es conveniente apreciar el flujo de los programas en ensamblador que ilustramos en la página 103.

La transposición modifica el estado y la clave realizando lo que sería el equivalente a una transposición matricial del estado y de la clave del cifrador. La razón por la que efectuamos este paso es porque con los datos en esta disposición es más sencillo y eficiente direccionarlos en el orden en que los requerimos. Por ejemplo al recorrer un byte de un renglón a la derecha en una unidad, podemos aprovechar las instrucciones con postincremento para reducir la manipulación de los índices. Para el caso de la transformación *MixColumns* en que los datos se acceden “verticalmente” no incurrimos en ninguna sobrecarga o ésta es mínima. En realidad es difícil asegurar que tenemos un mejor desempeño que el que hubiéramos obtenido si trabajáramos con el orden original de los datos hasta no haber implantado ambas posibilidades, pero creemos que es así.

Los pasos *AddRoundKey*, *SubBytes* y *ShiftRows* se aglutinaron en el mapeo *xor_sust_recorre* que

	AES Rápido		**AES Pequeño**	
Transformación	Ciclos	Tam. Código	Ciclos	Tam. Código
transposición	$3 \times 57 = 171$	76	$3 \times 57 = 171$	76
xor_sust_recorre	$10 \times 168 = 1680$	170	$10 \times 207 = 2070$	108
mezcla_columnas	$9 \times 206 = 1854$	90	$9 \times 300 = 2700$	32
calcula_siguiente_clave	$10 \times 102 = 1020$	118	$10 \times 139 = 1390$	74
resto_ultima_ronda	129	46	142	32

Cuadro 5.1: Desempeño de cada transformación del *AES* para nuestras dos versiones en ensamblador. El tamaño del código está dado en bytes y los ciclos corresponden a los del reloj externo.

reduce el movimiento de datos de los registros y de la memoria mediante la serialización de operaciones. Tomando en cuenta que se trata de tres transformaciones en una y que ocupamos memoria de programa en lugar de memoria *RAM* para la *caja S* creemos que tenemos un buen desempeño.

En donde es más evidente el compromiso entre tamaño y velocidad es en el paso *MixColumns*. Esta operación fue la que más esfuerzo requirió para optimizarla. Recordemos que uno de los principales requerimientos era minimizar el uso de la memoria de programa. Logramos reducirlo en aproximadamente un 64 % pero a costa de un incremento en el tiempo de procesamiento del 46 %. También tenemos una tercera implantación de este paso pero es superada por la versión pequeña en ambos sentidos. La diferencia de velocidades es poco significativa, así que difícilmente tendremos aplicaciones que forzosamente necesiten la versión rápida.

El cálculo de la siguiente clave de ronda se hizo "al vuelo". Gracias a que no implantamos la derivación de claves con el mismo algoritmo del pseudocódigo de la especificación es que pudimos reducir sustancialmente el tiempo de ejecución. No gastamos además ningún byte extra de memoria primaria, que en términos de la superficie de silicio es un recurso costoso.

En total, considerando toda la lógica adicional de las rutinas de cifrado y las cajas S que están en memoria de programa, tenemos que:

	AES Rápido	**AES Pequeño**
Ciclos de reloj	4911	6553
Tamaño del código (bytes)	772	598
Memoria RAM (bytes)	32	32
Tiempo de ejecución (ms)	1.29	1.65

Cuadro 5.2: Desempeño global de dos implantaciones del *AES*.

Los valores de tiempo de ejecución fueron determinados experimentalmente con las tarjetas inteligentes reales y nuestro equipo. Para ello se crearon algunos comandos auxiliares, los cuales llaman a nuestras rutinas de cifrado una cantidad dada de veces. En realidad estos tiempos empíricamente resultaron menores a los que teóricamente pronosticamos suponiendo una frecuencia del reloj de 3.57 MHz, al parecer se debe a que el reloj del lector funciona a 3.8 MHz.

En cuanto al uso de memoria *RAM* ocupamos solamente 32 bytes. 16 bytes corresponden al texto en claro que es transformado en el criptograma y los otros 16 a la clave del cifrador que es transformada en cada una de las claves de ronda.

Hay que remarcar que la versión rápida **no** fue optimizada para velocidad y la versión pequeña

solamente reduce el tamaño del código cuando el sacrificio en velocidad es módico. Si nuestra meta hubiera sido exclusivamente aminorar el tiempo de cifrado o el número de instrucciones hubiéramos llegado a otros resultados.

Para que la información presentada pueda ser analizada mejor, vamos a comparar nuestra implantación con otras que hemos encontrado en la literatura. La comparación de implantaciones es complicada porque existe una gran diversidad de arquitecturas, se tienen diferentes objetivos en sus diseños y las mediciones en ocasiones no son totalmente claras o compatibles.

5.1.1. Comparación del *AES* con Otra Implantación

Existe otra implantación del *AES* para un microcontrolador de la familia *AVR*. Fue hecha por *Kim Sung–ha* de la universidad *UCLA*. Es muy importante que quede claro que dicha implantación **no** está dirigida a las tarjetas inteligentes y su propósito exclusivamente fue la velocidad.

Aunque está diseñado para un microcontrolador similar al que utilizamos nosotros, portar este algoritmo a una tarjeta inteligente sería prácticamente imposible sin hacerle modificaciones mayores. Esto se debe a que emplea una memoria *SRAM* externa, ocupa los registros con total libertad, no es posible llamar a la función de cifrado desde lenguaje *C* y el código es muy extenso. En pocas palabras se trata de una implantación que no está hecha más que para cifrar una serie de datos fijos, tal como está ni siquiera puede ser llamada como una rutina en ensamblador.

Nos enteramos de la existencia de esta implantación del algoritmo hasta después de que habíamos completado las nuestras, así que no influyó en lo más mínimo en cómo realizamos las transformaciones. Nos parece muy grato haberla encontrado puesto que así tenemos un punto de comparación más confiable.

Las decisiones de diseño de este algoritmo son radicalmente diferentes a las nuestras. Esto es natural puesto que los criterios de optimización son de cierta forma contrarios. Primero que nada aquí sí se genera toda la clave expandida y se guarda en la memoria *RAM*. El estado no se mantiene en memoria sino en los registros. Las *cajas S* no se almacenan en memoria de programa sino en la *SRAM*. El direccionamiento para el estado es directo, esto es, mediante el nombre de cada registro, así que son imposibles la mayoría de los ciclos que podrían reducir el tamaño del código. No se utiliza la serialización de operaciones puesto que en este caso la manipulación de índices es menor y no hay sobrecarga por el acceso a los datos. El cifrador no devuelve el criptograma en la memoria sino que se queda en los registros.

Después de haber examinado el código podemos decir que, si bien las decisiones sobre el uso de los recursos son adecuadas para maximizar la velocidad, en realidad no nos parece suficiente el nivel de optimización de las transformaciones y creemos que todavía puede mejorarse substancialmente el desempeño. Independientemente de eso nos pareció que el código fuente está bastante "sucio" ya que el acceso a los registros se hizo en ocasiones por su nombre y en otras mediante un alias, hay comentarios que no corresponden con el código y hay mucho código que se eliminó metiéndolo en comentarios.

Los datos siguientes fueron obtenidos mediante el simulador de *AVRStudio* y de nuevo es preferible tratarlos como aproximaciones. El código fuente está pensado para ser ensamblado por dicho *software* así que no pudimos emplear el mismo compilador que el nuestro.

Como puede observarse especialmente en los pasos *MixColumns* y en la derivación de las claves de ronda, que fueron las transformaciones que más estudiamos, es donde tenemos un mejor desempeño en cuanto a la velocidad respecto a esta implantación. No contamos el tiempo que necesita la versión de *Kim Sung–ha* para pasar la caja S a *SRAM* que es de 768 ciclos.

	AES de *Kim, Sung–ha*	
	Generación Caja S	**Expansión de Clave**
Número de Ciclos	768	1784
Tamaño del Código (bytes)	768	120
	Suma Clave Ronda	**Sustituye Bytes**
Número de Ciclos	50 (una sola vez)	49 (una sola vez)
Tamaño Código (bytes)	68	66
	Recorre Renglones	**Mezcla Columnas**
Número de Ciclos	16 (una sola vez)	288 (una sola vez)
Tamaño del Código	32	134

Cuadro 5.3: Desempeño de cada transformación en la implantación del *AES* de *Kim Sung–ha*.

	AES de *Kim, Sung–ha*	**AES Rápido**	**AES Chico**
Cifrado Completo (Ciclos)	5599	4911	6553
Cifrado Reutilizando la Clave Expandida (Ciclos)	3815	—	—
Tamaño del Código (bytes)	1422	772	598
Uso total de *SRAM* (bytes)	432	32	32

Cuadro 5.4: Comparación general de la implantación de *Kim Sung–ha* con las nuestras.

Esta información quiere decir que cuando necesitamos cambiar de clave constantemente la implantación más veloz es *AES Rápido*. Por otro lado si rara vez cambiamos de clave y contamos con los recursos de *hardware* suficientes (seguramente no se trata de una tarjeta inteligente), entonces la versión de *Kim Sung–ha* es la más rápida.

También hay que señalar que en realidad esta implantación incurriría en una sobrecarga adicional ya que normalmente hay que mover el mensaje en claro y el criptograma hacia, o desde, la memoria a los registros. Además es frecuente que otras partes del código utilicen los registros con otros fines, por lo que seguramente nos veremos forzados a copiarlos temporalmente a memoria antes del cifrado y restituirlos a su término.

5.1.2. Comparación con el Algoritmo de Cifrado Original de *SOSSE*

El sistema operativo de las tarjetas inicialmente contaba con el algoritmo de cifrado *TEA* programado en ensamblador. No sabemos bajo que objetivos se desarrolló el código, pero dado que nosotros lo reemplazamos por el del *AES*, conviene determinar de que manera se ve afectado el rendimiento del sistema. Hay que recalcar que a diferencia de la implantación del *AES* de *Kim Sung–ha*, ésta sí funciona en nuestra arquitectura.

Intentamos inspeccionar el flujo del programa mediante el depurador *avr–gdb* pero encontramos un fallo al pasar sobre cierta instrucción y el proceso se muere con una violación de segmento. Por el momento no podemos aislar el error y reportarlo a las listas de desarrollo, pero lo haremos posteriormente.

Después de observar el código fuente superficialmente podemos decir que es difícil entender que está haciendo el programa puesto que no se utilizaron alias para los registros, los comentarios son sumamente escasos, las etiquetas son ininteligibles y se reutilizó código para el cifrado y el descifrado. Pareciera que fue generado por un compilador y muy ligeramente adaptado a mano.

Tenemos un código extremadamente compacto en parte debido a que no se utilizan tablas de sustitución. Se gastan solamente 318 bytes para el cifrado, haciéndole un buen honor a su nombre: *Tiny Encryption Algorithm*. Debemos tomar en cuenta que el tamaño de los bloques es de 8 bytes a diferencia del *AES*, por lo que para ser justos deberíamos duplicar los tiempos de ejecución por cada cifrado reportados para *TEA*. Como ya se mencionó nos tropezamos con un problema del depurador, así que no pudimos determinar el número exacto de ciclos que toma la ejecución. Afortunadamente tenemos la capacidad de probar la velocidad de funcionamiento real del algoritmo en una tarjeta[1] y que podemos apreciar en la siguiente tabla.

	Cifrado Mediante TEA (8 bytes)
Tiempo de ejecución real (ms)	3.41
Memoria de programa (bytes)	318

Cuadro 5.5: Desempeño de la implantación en ensamblador de *TEA* que utilizaba *SOSSE*.

En la sección 5.2 mostraremos de que manera se modifica el rendimiento de las funciones de autenticación al utilizar nuestras implantaciones de los algoritmos de cifrado.

5.1.3. Comparación con Implantaciones del Algoritmo para Otras Arquitecturas

No es sencillo comparar implantaciones de un algoritmo en distintos microcontroladores porque el desempeño está influenciado directamente por la arquitectura y frecuentemente los principios de diseño son diferentes. De todas formas es conveniente conocer al menos someramente cual es el rendimiento de nuestra implantación con respecto a otras para microcontroladores de tarjetas inteligentes. La información que vamos a presentar a continuación debe ser tratada con reservas.

Es muy importante remarcar que nuestros datos han sido corroborados experimentalmente, en contraste con los que vamos a presentar a continuación y que han sido obtenidos mediante simuladores.

En [26] *Geoffrey Keating* reporta una implantación de *Rijndael* para el 6805 que ocupa 879 bytes de *ROM*, 50 bytes de *RAM* y toma 9464 ciclos del reloj (la frecuencia tradicional de las tarjetas inteligentes es de 3.57 MHz). Se indica que la transformación *xtime* (multiplicación por x) de manera directa no toma tiempo constante, así que hay que desperdiciar varios ciclos.

El mismo autor proporciona información sobre el *DES*, el cual logró implantar con 136 bytes de *RAM*, 1036 bytes de *ROM* y un rendimiento de 29778 ciclos por bloque.

Por otro lado en [23] se evalúa el desempeño de cuatro ex–candidatos para convertirse en el *AES* en dos microcontroladores: el 8051 y el *ARM* (*Advanced Risc Machine*). Es importante notar que la arquitectura *ARM* es más poderosa que las tradicionales para tarjetas inteligentes. Es una arquitectura RISC de 32 bits, con *pipeline* de tres etapas y soporta frecuencias de reloj elevadas.

De todas las implantaciones que encontramos del *AES* para tarjetas inteligentes estas dos son probablemente la más centradas, ya que sí cuidan el uso de los recursos y reconocen que el algoritmo de cifrado no es el único componente de la lógica de la tarjeta.

Para la arquitectura 8051 se muestran dos versiones, pero nos parece muy difícil hacer comparaciones en cuanto a la velocidad de cifrado. El menor tamaño de código es de 512 bytes, con una tabla de 256 bytes y utilizando 64 bytes de *RAM*. El menor tiempo es de 3168 instrucciones pero cada una toma un cierto número de oscilaciones del reloj externo, que de acuerdo al documento es de 12 o 6.

[1] Creamos un comando específico para medir esta información.

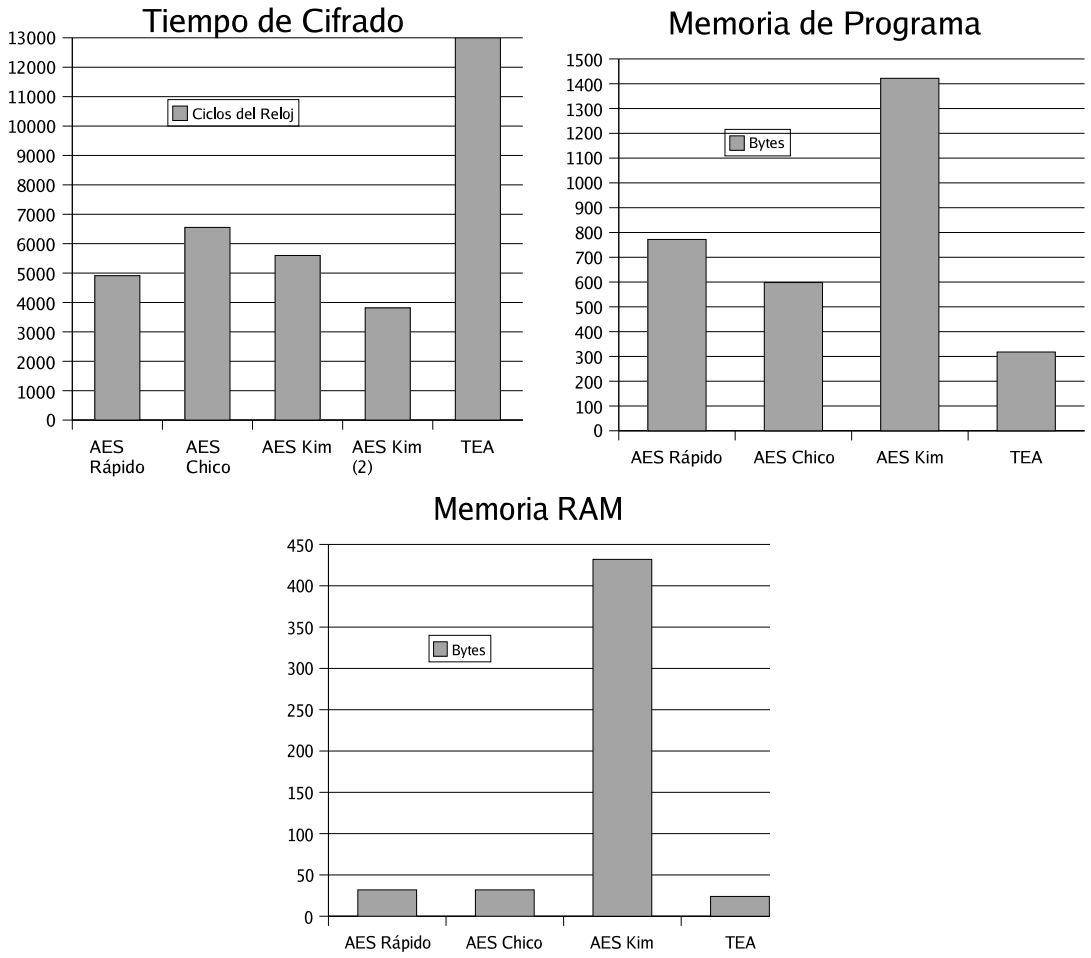

Figura 5.1: Gráficas comparativas para las implantaciones que funcionan sobre un microcontrolador de la familia *AVR*. Para *AES Kim (2)* no contamos la expansión de clave.

También se presentan dos implantaciones para la arquitectura *ARM*, ocupando 1148 y 2620 bytes de código, 256 y 1280 bytes de tablas, 16 y 32 bytes de *RAM* y tardando 2889 y 1467 ciclos de reloj para completar su operación respectivamente. Aquí cabe señalar que la frecuencia tradicional del reloj externo es 3.57 MHz, pero podría multiplicarse por 8, obteniendo una frecuencia neta de 28.56 MHz y que elevaría el rendimiento considerablemente.

Los resultados para distintas arquitecturas de tarjetas inteligentes indican que nuestro microcontrolador es capaz de ofrecer un mejor desempeño en cuanto a velocidad, uso de memoria *RAM* y tamaño del código que los que están basados en la tecnología CISC de 8 bits. Esto se debe en buena parte a que nuestro microcontrolador ejecuta alrededor de una instrucción por ciclo de reloj y tenemos una gran cantidad de registros de trabajo.

Por otro lado podemos apreciar que el rendimiento en una arquitectura de 32 bits fue muy superior en cuanto a velocidad, aunque el uso de tablas se incrementó significativamente. La justificación de esto es que el algoritmo favorece a las arquitecturas de 32 bits, el microcontrolador presentado es mucho más sofisticado, además que cuando tenemos palabras de 32 bits podemos realizar una optimización muy

efectiva pero que requiere al menos una tabla de 1kByte.

Debemos recordar que por lo general el cuello de botella es la lenta comunicación serial, así que a menos que estemos empleando una velocidad mayor a la estándar, todos estos microcontroladores dan tiempos de cifrado muy aceptables.

5.2. Mejora del Sistema Operativo de las Tarjetas

El *COS* de nuestras tarjetas, *SOSSE*, antes de que lo modificáramos ya contaba con mecanismos de autenticación y de control de acceso al sistema de archivos. La autenticación se consigue básicamente mediante los comandos *Internal Authenticate* y *External Authenticate* que trabajan en conjunción con *Get Response* y *Get Challenge* respectivamente. Mediante la autenticación externa el lector o el *host* se identifican frente a la tarjeta, lo recíproco ocurre con la interna.

De acuerdo al nivel de autenticación del usuario (mediante el PIN) y del *host* es que el sistema operativo permite o niega las operaciones sobre los archivos.

Existen otros comandos de autenticación, tales como *Verify PIN* y *Verify Key*, sin embargo están relacionados con protocolos de autenticación débil, y como no utilizan ningún algoritmo criptográfico no son de nuestro interés.

Los protocolos de autenticación fuerte, explicados en la página 57, requieren de la aplicación de un algoritmo de cifrado de bloques. En la versión original de *SOSSE* se empleaba *TEA*, pero nosotros nos encargamos de reemplazar los comandos involucrados para que utilizaran nuestras implantaciones del *AES*.

En realidad no podemos decir que *TEA* sea menos seguro que el *AES* (aunque sabemos de un ataque por claves relacionadas contra *TEA*), pero sí que el *AES* ha sido más estudiado por la comunidad abierta de criptoanalistas y por lo tanto tenemos más confianza en él.

Uno de los factores más importantes para evaluar el rendimiento de un comando es la velocidad con que se ejecuta. Aquí es muy importante distinguir entre los dos componentes del tiempo total de ejecución:

- *Tiempo de Transferencia de Datos*. Se refiere al tiempo en que el comando es transmitido hacia la tarjeta y en el que ésta da la respuesta. Aquí también estamos considerando cualquier sobrecarga producida por la comunicación de la aplicación con el demonio de *PC/SC*, el *driver* del lector y el lector mismo, aunque estos retrasos no son significativos si los comparamos con la velocidad estándar de transmisión del protocolo *T=0*.

- *Tiempo de Procesamiento*. Es el tiempo que le toma a la tarjeta interpretar y ejecutar el comando correspondiente. Aquí se incluye el retardo ocasionado por los algoritmos criptográficos, la lectura o escritura a memoria *EEPROM*, la lógica de autenticación, etc.

Puesto que el tamaño de bloque del *AES* es de 16 bytes y el de *TEA* de 8 bytes, parece conveniente aprovechar la mayor longitud para elevar la resistencia de nuestros protocolos criptográficos. Es posible modificar los retos y respuestas de los protocolos de autenticación para que solamente utilicen 8 bytes de los 16, pero esto sería en nuestra opinión un desperdicio del nivel de seguridad que podría obtenerse empleando el bloque completo, y solamente recomendaríamos hacerlo cuando minimizar la velocidad de transferencia de datos sea primordial.

5.2.1. Autenticación Externa

Para efectuar la autenticación externa, el *host* primero que nada obtiene el reto mediante el comando *Get Challenge*, lo cifra mediante la clave secreta destinada para este fin generarando la respuesta y la manda a la tarjeta con el comando *Extern Authenticate*.

Los bytes transmitidos en ambos casos son: 5 (para la cabecera del *APDU*), 8 o 16 para el reto o la respuesta y 2 para la palabra de estado (que indica el éxito o fracaso de la operación). En total tenemos 15 o 23 bytes dependiendo del tamaño de bloque.

El reto es creado pseudoaleatoriamente por la tarjeta, lo cual consigue mediante la función `hal_rnd_getBlock`, que a su vez llama al algoritmo de cifrado utilizando como parámetros el número serial de la tarjeta y un estado interno que se va actualizando.

A continuación veamos cómo se modificó el rendimiento real de esta operación, lo cual medimos en nuestro propio sistema, y naturalmente está sujeto a variaciones, especialmente causadas por el lector.

Comando	**Tiempo Total**	**Tiempo de Transferencia**	**Tiempo de Procesamiento**
Get Challenge con TEA	130 ms	58 ms	72 ms
Get Chalenge con el AES (Chico)	145 ms	75 ms	70 ms
Get Challenge Típico	55 ms	29 ms	26 ms

Cuadro 5.6: Desempeño del comando *Get Challenge*.

La instrucción *Get Challenge* modificada, la cual emplea nuestras implantaciones del *AES*, por un lado redujo ligeramente el tiempo de procesamiento, pero por otro lado aumentó el de transferencia de datos[2].

Aquí hay que resaltar que la duración del tiempo de procesamiento está influida directamente por el tiempo de lectura y escritura a memoria *EEPROM*. El tiempo de transferencia depende principalmente de la cantidad de bytes transmitidos. La disminución en el tiempo de procesamiento se debe a que nuestra implantación del *AES* es más rápida que la de *TEA*, y es acentuada porque internamente se llama dos veces al cifrador de bloques.

No estamos seguros de por qué nuestros tiempos de transferencia son mayores a los estándares, pero esto está muy relacionado con el lector y no hay nada que podamos hacer desde el punto de vista de la programación para elevar la tasa de transferencia de datos. En cuanto al tiempo de procesamiento, un elevado porcentaje se debe al acceso a la memoria *EEPROM*. En caso de requerir una velocidad de respuesta más elevada, podríamos incrementarla modificando el algoritmo de generación de números pseudoaleatorios.

Con el comando *Extern Authentication* se proporciona a la tarjeta la respuesta al reto del *host* para ser verificada. Esta comprobación implica llamar una vez al cifrador de bloques.

En los primeros dos casos (con el algoritmo *TEA* y el *AES* rápido) establecimos los tiempos a través de nuestro programa *control_sc* el cual adaptamos para tomar el tiempo en que se ejecuta cada instrucción. No presentamos el tiempo de procesamiento porque tenemos demasiada variancia en las mediciones, pero sabemos que es muy corto y debe estar abajo de los 5 ms empleando el algoritmo *TEA* y aún menor con el *AES*.

[2]Los valores típicos se dan de acuerdo a lo reportado en la literatura [35].

Comando	Tiempo Total
Ext. Authenticate con TEA	72 ms
Ext. Authenticate con el AES (Rápido)	84 ms
Ext. Authenticate Típico.	65 ms

Cuadro 5.7: Desempeño del comando *External Authenticate*.

5.2.2. Autenticación Interna

Mediante la autenticación interna el sistema *host* determina la validez de la tarjeta inteligente. Esto lo consigue con el comando *Internal Authenticate*, con el cual manda un reto a la tarjeta. Después solicita la respuesta mediante *Get Response*. Aquí es importante notar que las claves utilizadas son diferentes para los dos tipos de autenticación.

Si queremos tener un mayor control sobre nuestras tarjetas debemos asignarle una clave diferente a cada una, aunque esto podría traer consigo un problema de manejo de claves.

El principal factor que determina el tiempo de ejecución es la transferencia de datos. Para la versión mejorada del comando tenemos que el reto y la respuesta pasan de 8 a 16 bytes, veamos de que forma afecta esto el desempeño de los comandos mencionados.

Comando	Tiempo Total
Internal Authenticate con TEA (8 bytes)	62 ms
Internal Authenticate con el AES (16 bytes)	75 ms
Internal Authenticate Típico (8 bytes)	65 ms

Cuadro 5.8: Desempeño aproximado del comando *Internal Authenticate*.

Intentamos obtener la información sobre el tiempo de procesamiento, pero debido a que es corto (menos de 5 ms) y a que el tiempo de transmisión tiene una desviación estándar relativamente alta, preferimos omitirla ya que es poco confiable y nos podría llevar a deducciones erróneas. Lo que sí podemos afirmar es que nuestros tiempos de procesamiento son menores que los señalados como típicos (26 ms). Es probable que empleando un lector de mayor calidad se incremente la velocidad.

Después de que la tarjeta ha cifrado el reto la respuesta está disponible, pero el sistema *host* tiene que solicitarla mediante el comando *Get Response*. En este caso el procesamiento es despreciable y la carga se concentra en la transmisión.

Comando	Tiempo Total Promedio	Tiempo Total Mínimo
Get Response para TEA (8 bytes)	70 ms	48 ms
Get Response para el AES (16 bytes)	80 ms	66 ms
Get Response Típico (8 bytes)	33 ms	—

Cuadro 5.9: Desempeño aproximado del comando *Get Response*.

Las mediciones indicaron que algún componente introduce retardos esporádicamente que duplican el tiempo total. Por otro lado en ocasiones obtuvimos tiempos especialmente bajos lo cual sugiere que la tarjeta, al menos por sí sola, no es el factor que nos está alentando el proceso (puesto que su tiempo de respuesta teóricamente es prácticamente constante). Estamos bastante seguros de que la causa de estos problemas es el lector y para dar resultados más de acuerdo a la capacidad de las tarjetas eliminamos

manualmente aquellas mediciones que evidentemente son mucho más lentas que el promedio.

Con el propósito de explorar otra aplicación de nuestras implantaciones del *AES*, decidimos reemplazar la función tradicional de lectura de archivos *Read Binary* por nuestro propio comando *Read Binary CCM*. El punto importante es que la información que pasa de la tarjeta al *host* ahora está cifrada e incluye un código de detección de modificación que asegura su origen.

Por el momento el nivel de seguridad permanece constante ya que para mejorarlo nos hace falta también asegurar el flujo de datos hacia la tarjeta. De cualquier forma, la implantación de esta instrucción es importante porque es un paso previo a un comando *Envelope*. Existe un gran paralelismo entre la transmisión y la recepción de datos con el modo *CCM*[3], así que, si técnicamente fue posible implantar lo primero, también debe ser posible implantar lo segundo. La única razón que evitó que lo hiciéramos es que ya no contamos con la memoria de programa suficiente.

El algoritmo utilizado requiere, antes de cualquier procesamiento de los datos, establecer un *nonce* y una clave. Esto lo realizamos mediante el comando *Init CCM*. Básicamente se encarga de proporcionar el *nonce* en claro a la tarjeta y de solicitar una clave de sesión cifrada. La tarjeta transmite esta clave cuando le es indicado utilizando el comando *Get Response*.

La mayor parte del tiempo para estos dos comandos se dedica a la transferencia de datos y es de alrededor de 80 ms, veamos ahora una comparación del tiempo de lectura original con el nuevo. Aquí es importante recordar que la única sobrecarga en la transmisión ocurre para mandar el MAC, que tiene una longitud de 4 bytes.

Comando	**Tiempo Total**
Read (Original)	64.6 ms
Read CCM (Utilizando el AES)	78.9 ms
Read Típico[4]	35.9 ms

Cuadro 5.10: Desempeño aproximado para una lectura de 12 bytes.

La sobrecarga computacional del modo *CCM* es en este caso de 4 cifrados con el *AES*, y como puede observarse tiene un efecto adverso muy poco significativo. Con esta información podemos decir que las ventajas que podemos obtener al utilizar este modo de operación son muy superiores a las desventajas.

5.2.3. Otros Comandos

Además de los comandos ya mencionados, decidimos crear otros 5 comandos, que si bien no añaden funcionalidad extra al sistema final, son importantes puesto que nos permiten hacer mediciones controladas de los algoritmos de cifrado y de la transmisión de datos. De esta forma podemos experimentalmente determinar con mayor precisión información como: el tiempo que toma la comunicación y el tiempo que toma el procesamiento del comando.

Los primeros dos comandos permiten determinar la velocidad de cifrado, se llaman *Speed TEA* y *Speed AES*. En uno de los bytes de la cabecera del *APDU* el sistema *host* indica cuantas veces quiere que la tarjeta llame al cifrador de bloques. La tarjeta ejecuta dicha acción e inmediatamente devuelve una palabra de estado indicando que la operación se completó exitosamente.

Como el intercambio de datos es mínimo la velocidad de ejecución de estas instrucciones es bastante constante. El tiempo total varía de forma lineal al número de cifrados que le indicamos que realice la

[3]La recepción implica utilizar un poco más de memoria *RAM* puesto que antes de comenzar a escribir la información al archivo hay que comprobar su autenticidad, lo cual no se puede hacer hasta tener todo el paquete de datos.

tarjeta. La pendiente de esta función nos indica el tiempo que toma cada ejecución del algoritmo de cifrado. Al comparar esta información con la que pronosticamos que obtendríamos podemos decir que la frecuencia de reloj que le proporciona el lector a la tarjeta no es exactamente la más común y para la cual fue diseñado *SOSSE*, sino que es ligeramente más rápida. Esto afecta positivamente algunos tiempos de procesamiento, pero es posible que sea lo causante de que la velocidad de transferencia que obtuvimos sea menor a la documentada como típica.

Los otros tres comandos se ocupan de determinar los tiempos de transferencia de datos en que incurrimos al ejecutar cualquier comando. Esto lo logramos con instrucciones vacías pero que reciben o regresan la misma cantidad de bytes que sus contrapartes originales.

Veamos una descripción de cada uno de ellos:

- *Speed To Card.* Manda una cantidad dada de bytes a la tarjeta.
- *Speed Create Response.* Crea una respuesta de una cierta longitud y que está disponible con el comando *Get Response*.
- *Speed Read.* Lee de la tarjeta un bloque de bytes de tamaño arbitrario (máximo 255 bytes).

Para efectuar la medición en sí se necesita algún programa del lado del *host*. Los programas *control_sc* y *formaticc* (con nuestra modificación) pueden servir para este propósito.

5.3. Herramientas Desarrolladas

En esta sección describiremos los sistemas de *software* que desarrollamos con el fin de auxiliarnos en la integración del *AES* al sistema operativo de las tarjetas.

En un principio estas herramientas estaban enfocadas a simular la arquitectura y no habíamos pensado en que nos servirían para trabajar con las tarjetas reales. Los objetivos podrían cubrirse evaluando el desempeño de nuestras implantaciones solamente en la simulación, pero afortunadamente también pudimos hacerlo con las tarjetas.

Posteriormente las librerías y programas que creamos nos sirvieron para depurar los comandos que se ejecutan dentro de la tarjeta y que de otra forma serían extremadamente difíciles de corregir. De hecho la mayor parte del tiempo utilizamos la simulación, excepto para comprobar el funcionamiento de las funciones de la tarjeta y para realizar las pruebas de desempeño.

5.3.1. *Lector_simulador*

La primera herramienta que creamos, *lector_simulador*, nos permitió transmitir información con el sistema operativo el cual es simulado mediante *simulavr*. Simulamos el protocolo *T=0* y lidiamos con la manera en que *simulavr* procesa las lecturas y escrituras de los pines del microcontrolador.

El producto final fue una librería que nos proporciona algunas funciones muy fáciles de utilizar y con las cuales podemos comunicarnos directamente con el sistema operativo de las tarjetas. Con la librería podemos iniciar en el *background* al simulador y posteriormente nos ahorra manejar el protocolo que emplea la tarjeta, la manera específica en que accede a los contactos y la forma en que *simulavr* procesa las entradas y salidas del microcontrolador.

Ejemplificamos muy elementalmente esta librería con el programa *controlador*, que a diferencia de lo que su nombre indica, solamente lee la respuesta a *reset* de la tarjeta. Por nuestra propia experiencia

podemos decir que esta librería, si bien no es muy vistosa y todo lo que hace está oculto para el usuario, resultó ser muy útil. Otro sistema de *software* que hace un uso más completo de la librería se presenta a continuación.

5.3.2. *Pcsc_simulador*

No solamente es importante que nos comuniquemos con la tarjeta sino también es hacerlo de alguna forma establecida, puesto que queremos probarlas con programas que no sean de juguete sino que verdaderamente puedan trabajar con las tarjetas reales. Por ello escogimos la interfaz *PC/SC*, la cual es multiplataforma y que en principio era aplicable a nuestros lectores y tarjetas. De hecho cuando parchamos el *driver* todavía no teníamos las tarjetas para comprobar que los lectores funcionaban, así que en realidad no sabíamos si alguna vez podríamos utilizar al demonio de *PC/SC* real, pero supusimos que todo saldría bien y afortunadamente así fue.

Es importante notar que nuestro objetivo en este caso no era crear un simulador perfecto del demonio de *PC/SC* sino solamente proporcionar las funciones más comunes. Por ejemplo, no estaba claro de que forma nos podría servir simular varios lectores simultáneamente, otros protocolos de comunicación, o el acceso en modo compartido al lector.

Desde el punto de vista de las aplicaciones tenemos compatibilidad total a nivel del código fuente en la funcionalidad soportada, e incluso a nivel de código objeto antes de ligarse. En realidad solamente necesitamos ligar los programas con nuestras librerías en lugar de con la librería estándar de *pcsclite* (ligar medainte `-llector_simulador -lpscslite_sim` en lugar de con `-lpcsclite`).

Las aplicaciones al invocar las funciones de *PC/SC* no se dan cuenta de que se trata de una simulación sino que ven un lector disponible llamado *simulador_avr* y pueden ejecutar todas las operaciones tradicionales como: obtener el contexto del lector, conectarse con él, consultar su estado y transmitir comandos a la tarjeta.

Los programas que probamos y que pueden trabajar indistintamente con la simulación o con la arquitectura real son *pcsc_scan* y *formaticc*, los cuales forman parte de la distribución estándar de *pcsclite*, además de la aplicación de control de las tarjetas que creamos.

5.3.3. *Ccm*

Se trata de un programa que implanta el modo de operación *CCM* tanto para la transmisión como para la recepción de mensajes proporcionándonos cada uno de los resultados intermedios generados. Esta información fue útil para depurar nuestra aplicación de control *control_sc* que funciona en el sistema anfitrión y nuestra rutina de transmisión de datos que se ejecuta en la tarjeta.

Este *software* está programado en lenguaje *C* y está orientado a la comunicación serial. Es decir se transmite y recibe un byte a la vez. De esta forma pueden portarse fácilmente las funciones que implantamos para el sistema *host* a la tarjeta. Es muy importante tener presente la naturaleza de la comunicación puesto que esto nos permite disminuir el uso de la memoria *RAM* y proporcionar tiempos de respuesta más rápidos.

En realidad este programa es experimental y solamente puede ser necesario volver a emplearlo en caso de que implantemos la transferencia de datos con el modo *CCM* en ambas direcciones. Fue muy útil para depurar este algoritmo corriendo dentro de la tarjeta puesto que teníamos un problema que se reflejaba en un código de autenticación de mensajes calculado por la PC que discrepaba del generado por la tarjeta, pero gracias a este programa lo corregimos.

5.3.4. *Simulavr* y *Simulavr–disp*

Estos programas no fueron desarrollados por nosotros pero sí los modificamos e incrementamos su funcionalidad. Primero que nada detectamos un error en la simulación de cierto tipo de saltos relativos (*rjmp*) el cual reportamos a la lista de correos para que fuera corregido, de otra forma no sería posible simular el sistema operativo.

También solicitamos que se completara la funcionalidad de la memoria *EEPROM* interna y que es necesaria para *SOSSE* puesto que ahí guarda las claves, la respuesta a *reset*, el estado interno del generador de números pseudoaleatorios y parte del sistema de archivos.

Las siguientes modificaciones sí las realizamos nosotros. Hicimos lo requerido para poder tener control sobre el número de instrucciones ejecutadas modificando la estructura *AVRcore* del simulador y agregando las funciones correspondientes para actualizar su valor. Expandimos el protocolo con el cual *simulavr* se comunica con el coproceso de despliegue para incluir información sobre el reloj y las instrucciones ejecutadas. Modificamos *simulavr–disp* para que muestre dicha información.

Además le agregamos una nueva función a *simulavr–disp*. Ahora tiene la capacidad de llevar un control sobre cuántas veces se ejecuta cada instrucción y cuánto tiempo toma su ejecución. Con esta información podemos analizar más cuidadosamente nuestros algoritmos de cifrado y determinar en qué partes del programa se están gastando más ciclos del reloj, qué código se ejecuta más veces y qué saltos están añadiendo más sobrecarga. Esta información es muy importante porque nos puede permitir identificar las zonas que vale la pena optimizar más. Al final el programa de despliegue crea un archivo donde pone los resultados de la ejecución de la simulación.

5.3.5. Otros Componentes que modificamos

Para poder utilizar los lectores aparte del equipo mismo se necesita el *driver* correspondiente. El único *driver* que pudimos conseguir necesita ser actualizado porque está hecho para una versión vieja de la *API* de *pcsclite*. Nosotros nos encargamos de modificar el código fuente para que trabajara con la última versión, lo cual afortunadamente fue una tarea sencilla. Creemos que es posible hacer algunas otras mejoras al *driver* puesto que por ahora "se come" la respuesta a *reset*, la cual puede ser útil para identificar que tipo de tarjeta insertamos.

También modificamos una aplicación de nombre *formaticc* que es una herramienta que acompaña al demonio gestor de recursos de *PC/SC* en el paquete *pcsclite*. Originalmente esta herramienta servía para mandar una serie de comandos almacenados en una especie de *script* semiautomáticamente a la tarjeta. Nosotros agregamos la funcionalidad de que muestre el tiempo que tomó realizar cada uno de estos comandos. Esto nos fue muy útil para determinar, por ejemplo en conjunción con nuestra instrucción *Speed AES*, cuánto toma en la tarjeta efectuar cada cifrado con el *AES*. Fácilmente pudimos crear un *script* con el cual se llamó decenas de veces a esta instrucción y por consiguiente obtuvimos decenas de mediciones con las cuales pudimos calcular tiempos promedios más exactos.

5.4. El Sistema de Control y Administración de las Tarjetas

Se trata de una herramienta gráfica programada en lenguaje *C* que desarrollamos para probar e interactuar con todos los comandos importantes de la tarjeta.

La primera parte del sistema completo se encuentra en las tarjetas inteligentes, pero la segunda parte que es igualmente importante se ejecuta en el sistema *host* y es la que se encarga de realizar las operaciones específicas que hacen diferente a cada aplicación.

Es mediante la lógica del sistema *host* que se le da cierto significado a los datos contenidos en la tarjeta y que se les manipula para efectuar alguna tarea. Es decir, aunque desde el punto de vista de la tarjeta tengamos simplemente archivos, desde la perspectiva de la computadora anfitriona pueden significar cosas totalmente distintas como: el saldo, la cantidad de puntos acumulados, información sobre el tenedor de tarjeta, descuentos especiales, etc.

Con nuestro sistema de control *control_sc* completamos lo que se requiere para tener una aplicación basada en tarjetas inteligentes.

Es importante notar que se trata de una sola herramienta con la cual podemos trabajar con todas las versiones de *SOSSE* que desarrollamos además de con la versión original. La lógica del programa nos permite cambiar fácilmente de algoritmo de cifrado (*TEA* o el *AES*) y hacer lecturas en claro tradicionales o a través del modo *CCM*.

Creamos una interfaz gráfica sencilla de utilizar. Tenemos una serie de "pestañas", iconos y botones con los cuales podemos acceder a los comandos y solicitar su ejecución. Podemos probar cada una de las funciones de autenticación y trabajar con el sistema de archivos. No sacrificamos funcionalidad por facilidad de uso, puesto que en todo momento podemos mandar comandos arbitrarios a la tarjeta, aunque para ello requerimos conocer la codificación de las instrucciones de acuerdo al protocolo a nivel de aplicación.

Nuestro sistema de control está orientado a ser empleado por un programador que quiera experimentar con los comandos de las tarjetas, observar los flujos de datos, hacer mediciones de tiempos de respuesta, etc. Sin embargo, desde el momento en que diseñamos esta aplicación, pensamos en proporcionar un conjunto de módulos y funciones que pudieran servir como base para crear alguna aplicación destinada al usuario común inexperto.

En realidad ya estamos realizando las mismas tareas que efectuaría por ejemplo, una aplicación de cliente frecuente, solamente que por ahora tenemos que indicar manualmente qué comandos queremos realizar. Además debemos interpretar y manipular nosotros mismos el contenido de los archivos. Considerando el mismo ejemplo del programa de cliente frecuente necesitamos solamente automatizar las operaciones de autenticación, selección y lectura de archivos, modificar la información que contienen (como puntos acumulados, fecha de última compra y algunos datos personales) y rescribir la nueva información a la tarjeta.

Creemos que reutilizando los módulos que creamos podríamos extender y mejorar la funcionalidad de nuestra herramienta para llegar a una aplicación adecuada, de acuerdo al usuario final que tengamos en mente. Sabemos que hay algunos puntos que ciertamente pueden mejorarse. El primero es que la aplicación corre con un solo hilo, pero sería deseable crear otros hilos de manera que podamos realizar varias tareas "simultáneamente", esto sería útil en caso de que deseemos que el usuario sea notificado automáticamente cuando la tarjeta sea insertada o removida del lector.

Otra funcionalidad que resultó ser muy útil fue que agregamos una medición del tiempo que toma la ejecución de los comandos. Gracias a esta funcionalidad pudimos determinar la velocidad de algunas instrucciones que no habíamos podido medir previamente con *formaticc*. Este era el caso de los comandos de autenticación utilizando protocolos de reto / respuesta, puesto que es necesario computar la respuesta para cada reto que presenta la tarjeta y que en principio nunca se repite.

5.5. Librería para Manejar las Estructuras Algebraicas de *Rijndael*

A causa de la necesidad de utilizar las operaciones matemáticas de *Rijndael* desarrollamos una librería que nos permitió trabajar directamente con las estructuras algebraicas con que se definió el algoritmo. El producto resultante es la librería llamada: *estructuras*, que contiene una serie de clases con las cuales, si las empleamos adecuadamente, puede efectuarse cada una de las transformaciones del cifrado directo y del inverso, entre otras cosas.

Debido a que un punto importante que queríamos mostrar es cómo podemos construir una estructura más compleja a partir de otra más sencilla, por ejemplo $GF(256)$ a partir de $GF(2)$ empleando anillos polinomiales, y para poder aplicar la misma implantación del algoritmo extendido de *Euclides* a estructuras arbitrarias, decidimos programarla en *C++*.

La base para comprender nuestra librería es la clase *ElementoAbstracto*, la cual es una clase virtual en la que especificamos una serie de métodos que las clases derivada deben implantar. Está pensada para soportar estructuras algebraicas con dos operaciones donde una de ellas es una suma y la otra una multiplicación.

La primera clase que derivamos de *ElementoAbstracto* fue *GF2* que corresponde a $GF(2)$. Se trata de un campo sumamente simple pero importante. Es muy fácil realizar operaciones con este campo y aparentemente no hay mucha ganancia en crear una clase específica para trabajar con él, pero lo que hicimos fue crear un bloque que nos sirvió como base para generar otras estructuras.

Las operaciones algebraicas del *AES* están muy relacionadas con la aritmética módular de polinomios y de hecho cada una de las operaciones puede representarse como alguna operación entre ellos. Como podemos tener muchos tipos diferentes pero todos realizan las mismas operaciones básicas, decidimos crear una clase que encapsula el comportamiento de los polinomios con cualquier tipo de coeficientes, siempre que estos se deriven de la clase base *ElementoAbstracto*. Esto quiere decir que podemos manejar polinomios cuyos coeficientes son: polinomios, campos, los enteros, etc. Esta construcción es importante porque gracias a ella es que pasamos de $GF(2)$ a los anillos conmutativos con coeficientes en $GF(256)$.

Otra estructura que derivamos de *ElementoAbstracto* fue la de los enteros, *Z*. No se utiliza en realidad para las transformaciones de *Rijndael*, pero la creamos para mostrar como puede aplicarse a ella el algoritmo extendido de *Euclides*.

Con la clase llamada *GF256* realizamos las operaciones de $GF(256)$ representando a cada elemento del campo como un polinomio de grado menor o igual a 7 sobre $GF(2)$ y empleando aritmética con un módulo arbitrario especificado al momento de instanciar cada objeto. Fácilmente podemos realizar con esta clase las operaciones de este campo, y en conjunción con la clase *polinomio* definimos los polinomios de grado menor o igual a 3 que se utilizan para la transformación *MixColumns*

Para implantar *Rijndael* es suficiente contar con la especificación y seguirla al pié de la letra, pero si lo que queremos es aprender sobre las operaciones matemáticas que efectúa, entonces es más conveniente emplear una librería como la nuestra con la cual se puede trabajar con los datos en el mismo nivel conceptual con que fueron definidos.

Utilizando esta librería pudimos representar *todos* los pasos de las rondas del *AES* del cifrado y descifrado como operaciones relacionadas con polinomios. Esto lo materializamos con el programa *pasosRijndael*. Solamente dejamos de lado las transformaciones que involucran la clave, pero no habría ninguna dificultad en añadirlas.

Determinamos cada uno de los pasos inversos, para lo cual utilizamos el algoritmo extendido de *Euclides* aplicado a estructuras diversas. Para la inversión de la transformación *affine* primero representamos una multiplicación entre una matriz circulante y un vector como una multiplicación entre polinomios, y luego la suma entre vectores como una suma de polinomios (puede observarse como lo hicimos mediante

el programa *invAffine*). Invertimos la transformación en este dominio y fácilmente nos pudimos regresar al matricial. No repetimos aquí este resultado porque lo podemos encontrar en la especificación del algoritmo.

El paso *ShiftRows* y su inverso involucran un corrimiento cíclico diferente para cada renglón del estado. Podemos obtener este mismo efecto tratando a los renglones como polinomios con coeficientes sobre $GF(256)$ y multiplicándolos por 1, x, x^2 o x^3 con el módulo x^4+1.

En cuanto a la transformación *MixColumns* calculamos el paso inverso mediante el programa *invMixColumns* y concordamos con el polinomio dado en la especificación. Además utilizando el programa *raicesMixColumns* determinamos las raíces del polinomio $c(x)$ que es la base de este paso: 0xBC, 0xBD y 0xF7. Es importante que este polinomio no tenga un factor $x+1$ puesto que eso haría que la transformación no fuera una biyección.

En la literatura se indica que la caja S puede representarse como un polinomio de grado 254 utilizando la interpolación de Lagrange. Se menciona que se buscó una combinación de transformaciones que tuviera una representación algebraica compleja. Escribimos un programa (*lagrange*) que realiza dicha interpolación y comprobamos nuestro resultado. Repetimos el mismo procedimiento pero ahora con la caja S inversa (con *invLagrange*) y llegamos a una representación aparentemente mucho más compleja. Se trata de un polinomio de grado 254 con todos los coeficientes diferentes de cero (excepto el 255). Este resultado contrasta con la transformación directa que puede representarse de esta forma empleando tan solo 9 coeficientes diferentes de cero.

5.6. Resultados Generales

En esta sección vamos a señalar cuáles fueron nuestros resultados desde un punto de vista global y con relación a los objetivos que planteamos al inicio de la tesis.

Los resultados más importantes a los que dio lugar el trabajo de esta tesis son:

- Exploramos la tecnología de las tarjetas inteligentes abarcando una gran cantidad de temas que van desde los enfocados a su uso práctico hasta su funcionamiento interno incluyendo: sus aplicaciones más exitosas, diversas clasificaciones, los protocolos utilizados, sus sistemas operativos, el sistema de archivos y los comandos más útiles.

- Comprendimos los mecanismos de seguridad que implementan las tarjetas inteligentes basados en métodos simétricos.

- Seleccionamos una arquitectura de tarjetas inteligentes, examinamos sus características, su conjunto de instrucciones y aprendimos a utilizar las herramientas de desarrollo.

- Simulamos toda la arquitectura de la tarjeta (excepto la memoria *EEPROM* externa) utilizando un simulador del microcontrolador y nuestras propias herramientas, comprendiendo al sistema operativo, el microcontrolador, los protocolos de comunicación de la tarjeta y una interfaz de programación para interactuar con la tarjeta mediante aplicaciones del sistema *host*.

- Analizamos el funcionamiento de un sistema operativo para tarjetas inteligentes con el propósito principal de entender la forma en que procesa los comandos y efectúa las funciones de autenticación.

- Estudiamos los principios básicos de la criptografía simétrica, con atención especial en las primitivas criptográficas que hacen posible como: las funciones *hash*, los códigos de autenticación de mensajes y los cifradores de bloques.
- Asimilamos diversos protocolos fundamentados en las técnicas simétricas: algunos protocolos de autenticación, un protocolo de establecimiento de clave y los modos de operación más comunes, tratando brevemente sus características más importantes.
- Comprendimos la especificación del algoritmo *AES* analizando cada una de sus transformaciones para el cifrado directo e inverso desde la perspectiva de las operaciones matemáticas con las que se definió y tomando en cuenta la forma de implementarlas eficientemente para nuestra arquitectura. En este punto ayudó la librería que creamos para el manejo de las estructuras algebraicas de *Rijndael* y sus aplicaciones.
- Desarrollamos dos implantaciones del *AES* optimizadas en bajo nivel para nuestra arquitectura, las cuales cumplieron un criterio de diseño gracias a lo cual tienen la capacidad de funcionar en la arquitectura real haciendo un uso cuidadoso de los recursos y dando un buen desempeño.
- Integramos nuestros algoritmos de cifrado en el sistema operativo de las tarjetas mejorando las funciones de autenticación interna y externa.
- Estudiamos e implantamos un modo de operación creando un nuevo comando de lectura de archivos con el fin, al menos para esta operación, de añadir el cifrado de los datos y un código de autenticación de mensajes.
- Desarrollamos una aplicación para utilizar las funciones más útiles que soporta la tarjeta, en cualquiera de las versiones del sistema operativo.
- Medimos el desempeño de nuestro algoritmo desde un punto de vista teórico (con ayuda del simulador) y práctico (con nuestro programa de control y otras herramientas).
- Comparamos nuestra implantación con otra diseñada para otro microcontrolador de la misma familia y optimizado para velocidad, determinamos que para el cifrado completo una de nuestras implantaciones es más rápida y utiliza muchos menos recursos.
- Evaluamos el desempeño real de los comandos nuevos o modificados y los comparamos con los anteriores, con los tiempos típicos y los reportados para otras arquitecturas.

Consideramos que con estos resultados se cumplieron ampliamente los objetivos fijados al inicio de este trabajo de tesis.

Capítulo 6

Conclusiones

A través del presente trabajo de tesis hemos demostrado que la implantación del algoritmo *AES* para una arquitectura de tarjetas inteligentes, la cual no fue evaluada durante el proceso de evaluación y selección del *AES*, la arquitectura RISC de 8 bits *AT90S8515* de la familia de microcontroladores *AVR* de *Atmel*, es totalmente factible. Además mostramos que puede ser utilizado como una primitiva básica para construir o mejorar otros algoritmos o protocolos criptográficos que funcionan dentro de las tarjetas. De hecho mostramos que su desempeño es superior, en cuanto a la velocidad de cifrado y uso de memoria de programa y de datos, respecto al desempeño que ofrecen los dos microcontroladores más utilizados para las tarjetas inteligentes: el 6805 y el 8051, que tienen tecnología CISC de 8 bits. Esto se debe principalmente a: la capacidad del microcontrolador seleccionado de ejecutar la mayoría de las instrucciones en un solo ciclo de reloj (alcanza cerca de 1 MIPS por cada 1 MHz del reloj externo), contar con modos de direccionamiento eficientes y tener disponibles una gran cantidad de registros de trabajo. Puede obtenerse un rendimiento aún mejor si se está dispuesto a costear una potente arquitectura de 32 bits, tal como la *ARM*. Sin embargo el verdadero "cuello de botella" se encuentra en la lenta comunicación serial que muy frecuentemente es de 9600 bits/s.

Los resultados indican que nuestra implantación del *AES* tiene un mucho mejor desempeño que el reportado como típico utilizando el anterior algoritmo *DES* con otras arquitecturas de tarjetas inteligentes. Esto concuerda con otros resultados encontrados en la literatura y significa que, al menos desde el punto de vista de los recursos computacionales, es ampliamente recomendable reemplazar al veterano *DES* por el nuevo *AES* en las tarjetas inteligentes.

El microcontrolador *AT90S8515* exhibió excelentes cualidades. Podemos afirmar que su lenguaje ensamblador es claro y fácil de aprender, el código resultante es compacto, tiene un mapeo muy directo desde lenguaje *C*, y contamos con un eficiente conjunto de instrucciones. Tal es el caso de aquellas que permiten el direccionamiento indirecto con desplazamiento y las que efectúan un postincremento o predecremento del índice de 16 bits. Estas fueron las instrucciones clave con las que logramos optimizar el algoritmo.

La serialización de operaciones permitió obtener una velocidad cercana a la que ofrece el trabajar con el estado almacenado en los registros, y mucho más elevada que si efectuáramos cada transformación por separado con el estado en la memoria. Gracias a la minimización del número de accesos a los datos y a la reducción de la manipulación de apuntadores, es que el deterioro en la velocidad con la primera técnica respecto a la segunda es pequeño. Sin embargo la serialización de operaciones nos permitió compactar significativamente el código, lo cual es fundamental para cumplir con nuestro criterio de optimización, y es importante puesto que la memoria de programa, al igual que los demás recursos de las tarjetas

inteligentes, es en general escasa. Esta información nos permite concluir que los resultados de *Kim Sung–ha*, referentes a una supuesta reducción del tamaño del código y del tiempo de ejecución de un 40 % al emplear los registros para almacenar el estado, presentan deficiencias importantes y su validez es muy cuestionable.

La transformación *MixColumns*, a pesar de haberse optimizado considerablemente, es el paso más complejo computacionalmente, por lo que afecta substancialmente el desempeño global del algoritmo. Es posible implantar cada una de las transformaciones, y en especial ésta, de muchas formas distintas, lo cual da lugar a diferentes desempeños. Esto es muy útil puesto que podemos hacer un compromiso entre la velocidad y el tamaño del código de acuerdo a los requerimientos de nuestras aplicaciones.

La sugerencia de optimización presentada en la especificación del algoritmo permite reducir el número de operaciones que toma este paso desde el punto de vista matemático, pero es solamente un primer adelanto para reducir el tiempo de ejecución y el número de instrucciones. El conocimiento de la arquitectura y del algoritmo resultó ser fundamental para optimizar esta transformación al punto de que nuestra implantación *AES rápido* venció en el tiempo de cifrado completo a otra implantación, cuyo objetivo fue la velocidad y calificada por su autor como “altamente optimizada”.

La especificación de algunos de los pasos de *Rijndael* mediante pseudocódigo similar al lenguaje *C* no es conveniente para hacer una implantación directa utilizando esta arquitectura, sino que solamente debe servirnos para comprender las operaciones que sufren los datos. Es más provechoso entender el algoritmo, determinar cómo se modifican los datos y transformarlo en uno nuevo. Esto lo conseguimos replanteando cuidadosamente la lógica, estableciendo una secuencia de operaciones que posteriormente pudieron aprovechar más eficientemente las instrucciones del microcontrolador.

Los resultados muestran que mediante el algoritmo *AES* podemos disminuir significativamente el tiempo de procesamiento de los comandos respecto al que teníamos previamente empleando *TEA* y respecto a los tiempos documentados como típicos utilizando el *DES*. Esto quiere decir que con nuestras implantaciones del algoritmo tenemos una mayor flexibilidad en cuanto a los mecanismos de seguridad que podemos implantar en las tarjetas inteligentes, por lo cual podemos, por ejemplo, cifrar todos los datos transmitidos con alguno de entre los varios modos de operación existentes basados en el *AES*. Respecto a otras arquitecturas y algoritmos en que el cifrado es más lento, es más difícil pensar que esta clase de técnicas pudiera mejorar la seguridad, dado que la penalidad en el tiempo de procesamiento en que se incurriría, se reflejaría directamente en el tiempo total de ejecución de los comandos que hagan uso intensivo del cifrador de bloques.

El modo de operación *CCM* resultó ser especialmente apropiado para utilizarse en las tarjetas inteligentes con el fin de cifrar y autenticar el origen de los datos transmitidos. Presenta varias características muy favorables que permiten minimizar los tiempos totales de ejecución de los comandos de las tarjetas, y los recursos que ocupa son razonables de acuerdo a los que tenemos disponibles. El número de veces que se invoca al cifrador de bloques es relativamente bajo, de forma que tenemos una sobrecarga computacional muy poco significativa comparada con los tiempos de transmisión de datos. Otra ventaja muy relacionada con las tarjetas inteligentes es que ni el cifrado ni el descifrado de la información a través de este modo emplean el cifrador inverso, por lo que se reduce el uso de la escasa memoria de programa. Si bien desde la tarjeta solamente implantamos la transmisión de datos, no existe ninguna dificultad técnica esencial que impida que hagamos el proceso inverso.

Tenemos la libertad de especificar de antemano el tamaño del código de autenticación de mensajes, lo cual significa que podemos hacer un balance entre el nivel de seguridad y la cantidad de bytes extra

que transmitimos. Otra característica fundamental de este modo de operación es que, gracias a que uno de sus constituyentes es el modo contador, no hay ningún incremento en la cantidad de datos que son recibidos o transmitidos, excepto por el mismo *MAC* que no puede ser eliminado. No cabe duda de que el modo *CCM* es perfectamente aplicable a las tarjetas inteligentes y puede ser la base para implantar un comando tipo *envelope* sumamente eficiente.

A partir de un simulador del microcontrolador escogido, utilizando nuestras propias herramientas logramos crear un simulador de las tarjetas. Gracias al presente trabajo ahora contamos con un ambiente completo de simulación de las tarjetas que se apoya en la cadena de herramientas de desarrollo *GNU*.

Con la simulación de las tarjetas, por un lado podemos depurar los algoritmos y comandos que implantemos, y por otro lado tenemos la capacidad de hacer pruebas conjuntamente con las aplicaciones que corren a nivel de *host*. Esto significa que podemos ejecutar y depurar los programas de la computadora, simultáneamente con el sistema operativo de las tarjetas, sin siquiera contar con el *hardware* que dichos programas consideran que están utilizando. Esperamos que nuestro trabajo al crear este ambiente de simulación de las tarjetas pueda ser reutilizado con el fin de crear más fácilmente otros comandos más sofisticados.

6.1. Trabajo Futuro

En esta sección vamos a indicar algunos puntos en los que sugerimos se continúe trabajando:

- Aumentar la flexibilidad de *SOSSE*: Sería muy conveniente convertir al *COS* de un sistema operativo multifuncional a un sistema operativo multiaplicación. Esto involucra crear claves distintas para cada archivo dedicado (directorios) y condicionar el acceso a los archivos que contiene. De esta forma cada aplicación tendría datos aislados e independientes, por lo que varias aplicaciones podrían convivir en la misma tarjeta sin disminuir la seguridad de las demás.

- Separar las condiciones de acceso a los archivos: Varias aplicaciones se beneficiarían de poder establecerse condiciones diferentes para permitir la lectura y la escritura de los archivos. Un ejemplo en que esto sería útil, es para proporcionar información personal (nombre, tipo de sangre, etc.). Nos interesaría que muchas partes puedan tener acceso de sólo lectura a dicha información, pero queremos que su modificación esté más restringida.

- Portar nuestro trabajo a una arquitectura con 16 Kbytes de memoria de programa, tal como la de las tarjetas basadas en el microcontrolador *ATmega161*, también de la familia *AVR*. Con esto tendremos un aumento en el espacio disponible para implantar nuevos algoritmos. No es nada complicado realizar este paso puesto que el conjunto de instrucciones que utilizamos en nuestras implantaciones es un subconjunto de las que soporta este microcontrolador, y en realidad son prácticamente las mismas. Lo que queda por investigar, es si es posible hacer uso de las pocas instrucciones extra disponibles para optimizar aún más el algoritmo.

- Después de realizar el trabajo del punto anterior, tendremos la posibilidad de implantar un comando *envelope* reutilizando nuestro comando *Read Binary CCM*, pero ahora cifrando y autenticando todos los datos transmitidos entre la tarjeta y el sistema *host*. La protección de la información ahora deberá incluir a los comandos mismos y a sus respuestas.

- Puesto que el *hardware* de las tarjetas aparentemente no fue hecho con la seguridad en mente, es necesario determinar qué tan confiable es. Para probar la resistencia de las tarjetas a ataques físicos hay varias tareas que podemos realizar.

  En cuanto a los ataques de *side channel*, sabemos que no hay ningún problema con los ataques por medición de tiempos utilizando nuestra implantación del algoritmo, pues toma un tiempo de cifrado constante. Por lo menos falta estudiar la factibilidad de los ataques por medición de las emisiones electromagnéticas y mediante el análisis diferencial de potencia (DPA).

  Respecto al DPA, *Brice Canvel* comenzó a estudiar la resistencia de la arquitectura, para lo cual utilizó las versiones de *SOSSE* y del *AES* que desarrollamos para esta tesis. Desgraciadamente su trabajo quedó inconcluso, y al menos hasta donde se quedó, nos informó que no pudo realizar dicho ataque con éxito.

  Por otro lado, también hace falta determinar la resistencia del circuito integrado para revelar la información almacenada directamente en las celdas *EEPROM* y *Flash*. Estos datos podrían accederse retirando la superficie que cubre al circuito integrado y examinando el estado de las celdas.

  En el peor de los casos, si la seguridad que proporciona el *hardware* fuera muy baja y las aplicaciones lo requirieran, nos veríamos forzados a migrar a una arquitectura de la misma familia, pero aún más potente: la *AT90SC*. Estos microcontroladores sí fueron diseñados específicamente con la seguridad en mente, pero desgraciadamente no se trata de una arquitectura abierta.

- Cifrar el contenido de las memorias *EEPROM* interna y externa. A causa del buen desempeño de nuestras implantaciones del *AES*, es totalmente posible mejorar la seguridad de las memorias *EEPROM*, especialmente de la externa que es muy vulnerable. Aquí lo que queremos evitar son los ataques físicos que podrían recuperar el contenido de la memoria, y que revelarían, por ejemplo: claves secretas y otra información valiosa. Claro que la clave utilizada debe almacenarse en alguna parte. Para dificultar los ataques que directamente recuperen dicha clave, puede especificarse una función sencilla que la determine, por ejemplo un or–exclusivo entre un bloque de datos que se encuentre en la memora de programa y otro que se encuentre en la de datos interna. De esta forma el adversario tendría que recuperar la información de dos tipos distintos de memoria. En teoría el circuito integrado, una vez que hemos quemado ciertos fusibles irreversiblemente, no permitirá la lectura de dichas memorias. El cifrado de la memoria deberá utilizar algún modo de operación que permita el acceso aleatorio a los datos, tal como el modo contador, y lo debemos conjuntar con algún mecanismo de autenticación para evitar manipulaciones.

- Documentación general de las herramientas de simulación desarrolladas. La primer tarea verdaderamente importante que queda pendiente es escribir los manuales para los usuarios y para quienes estén interesados en modificar nuestras herramientas.

- Mejora de la librería *lector_simulador*. Antes que nada sería conveniente añadir algún mecanismo, por ejemplo a través de dos *pipes* nombrados, que nos permita indicarle al sistema cuando hemos removido o insertado la tarjeta, y conocer qué tiempo cree el microcontrolador que ha transcurrido en la simulación. Esto principalmente para eliminar algunos tiempos de espera en la librería que estrictamente no son necesarios.

  Por ahora *SOSSE* solamente trabaja con números pseudoaleatorios, pero en el código fuente podemos apreciar que ya hay una implantación preeliminar de un *TRNG* (*True Random Number Generator*) que aparentemente ya casi está listo. Para soportar esta funcionalidad en la simulación,

sería necesario añadir retardos aleatorios en los tiempos de espera que registra el sistema operativo para la comunicación, los cuales son normales y aparecen al transferir datos entre la tarjeta y el lector. Estos retardos son la base para crear el *TRNG*.

- Mejorar la librería para manejar las estructuras algebraicas de *Rijndael* y los programas de ejemplo que la utilizan. Hace falta dar una limpieza general al código, crear los destructores de las clases y sobrecargar algunos operadores para hacer más sencillo el uso de la librería. Otra tarea pendiente es completar el cifrado completo con *Rijndael*, lo cual implica implantar las transformaciones que involucran la clave, lo cual sería sencillo.

  También podría ser útil, crear un programa con el cual seamos capaces de interactivamente realizar el cifrado y descifrado, y modificar algunos parámetros de los pasos del algoritmo, tales como la transformación *affine* de la caja S, los corrimientos del paso *ShiftRows* y el polinomio del paso *MixColumns*.

- Mejorar nuestro sistema de control y administración de las tarjetas, *control_sc*. Hace falta dar una retroalimentación al usuario más explícita sobre las condiciones de error encontradas. También sería útil añadir un hilo que esté pendiente de los cambios de estado de la tarjeta (por ejemplo, que nos avise automáticamente si fue retirada o insertada).

  Por otro lado, en realidad no encontramos ningún simulador de tarjetas inteligentes libre y de calidad que proporcione una interfaz de comunicación compatible (al menos parcialmente) con *PC/SC*. Los que hemos observado se limitan a imitar la funcionalidad de la tarjeta y no es posible que las aplicaciones reales los utilicen como sustituto del *hardware*. Sería una buena idea pulir y hacer disponible al público esta herramienta (junto con las demás de la simulación) para facilitar a otras personas el comenzar a trabajar con las tarjetas *funcards* y con *SOSSE*.

Apéndice A

Información Adicional sobre las Tarjetas Inteligentes

A.1. Origen y Evolución de las TIs

Las tarjetas como mecanismo de identificación personal se han utilizado desde hace muchos años. El desarrollo de las tarjetas plásticas comenzó en 1950 cuando se logró producir tarjetas de PVC durables y de bajo costo. *Diners Club* creó una tarjeta de pago de bienes en general. En ese momento el número de tarjetas era reducido, solamente los individuos con la mayor solvencia económica podían adquirirlas. Era por ello un símbolo de estatus.

En 1960 el *Banco de América* (EUA) emitió la primera tarjeta de crédito. Estas tarjetas carecían de mecanismos de seguridad robustos. Tenían escrito el nombre del tenedor en relieve y su firma. Debido a que no eran muy comunes y se les consideraba muy valiosas los vendedores ponían bastante atención en que la firma de la tarjeta y la que daba el cliente correspondieran. Estas medidas de seguridad eran adecuadas en ese entonces, pero eso no duró mucho tiempo. Sabemos que ahora las personas que aceptan las tarjetas no tienen mucho cuidado en corroborar las firmas. Además éste es un mecanismo que tiene limitaciones a causa de la variabilidad de las firmas (se puede rechazar una firma válida) y la posibilidad de falsificarlas (se puede aceptar una firma inválida).

El equipo para imprimir las tarjetas de crédito y escribir en ellas en relieve se hizo más accesible. Gracias a la entrada de *Visa* y *Mastercard*, las tarjetas de crédito pasaron de ser algo exclusivo a ser un artículo de masas, los mecanismos de seguridad descritos fueron insuficientes y se prestaban a fraudes frecuentes, los cuales año con año iban aumentando.

Estos problemas impulsaron el desarrollo de algunas otras tecnologías más difíciles de falsificar como la microimpresión y los hologramas. También se empleó tinta visible solamente bajo la luz ultravioleta. Otra tecnología que surgió fueron las fotografías impresas en las tarjetas. Desgraciadamente las tecnologías mencionadas sufren de problemas similares, de origen humano y difíciles de evitar. En ocasiones las personas que deben verificar que estos mecanismos de seguridad se encuentren presentes y sean correctos, no lo hacen o no ponen la atención suficiente como para detectar cualquier anomalía.

Las tarjetas magnéticas surgieron como un mecanismo que permitió la automatización en el manejo de las tarjetas. Mediante un campo magnético las partículas de la banda son alineadas de acuerdo al dato que contienen. Esta tecnología resultó ser sumamente útil pero desgraciadamente no es lo suficientemente avanzada como para brindar un alto nivel de seguridad. Debido a que la banda magnética es totalmente

accesible y a que los dispositivos empleados para leerla y escribirla son relativamente sencillos (el principio es fundamentalmente el mismo que en las grabadoras comunes de *cassettes*), es fácil realizar ataques a las tarjetas magnéticas. Un ataque muy común es llamado *skimming*[1] y consiste en leer la información de una tarjeta y escribirla en otra, pudiendo utilizar la segunda para fines ilegales sin la autorización ni conocimiento del propietario de la tarjeta original.

Otro de los problemas que presentan las tarjetas magnéticas es que a causa del uso diario o de estar en contacto con campos magnéticos externos, la información contenida en la banda puede corromperse ocasionando pérdida de la información. Las tarjetas de alta cohercitividad surgieron como solución a este problema. Estas tarjetas requieren un campo magnético mucho más fuerte, aunque difícil de producir. Como efecto colateral se incrementó la seguridad ya que los equipos para escribir en estas tarjetas eran más difíciles de conseguir.

Las tarjetas ópticas son una alternativa a las tarjetas magnéticas y a las tarjetas inteligentes. Mediante un procedimiento similar al del lector de Discos Compactos (CD's), un rayo láser lee la información escrita en la superficie de la tarjeta. La ventaja principal que ofrecen las tarjetas ópticas sobre las tarjetas magnéticas es su gran capacidad de almacenamiento. La confidencialidad de la información no se asegura, a menos de que la información sea guardada cifrada, pero esto introduce otros problemas difíciles de resolver como la transferencia y administración de claves.

Los avances en microelectrónica y la posibilidad de integrar memorias y lógica en un solo circuito integrado junto con la idea de utilizar este circuito integrado con fines de identificación personal fueron los factores necesarios para el desarrollo de la tecnología de las tarjetas Inteligentes. En 1968 dos inventores Alemanes llamados *Jürgen Dethloff* y *Helmut Grötrupp* obtuvieron la primera patente de un circuito integrado incrustado en una tarjeta de identificación.

En 1970 *Kunitaka Arimura*, un inventor Japonés, obtuvo una patente similar a la patente anterior, pero solamente para Japón y limitada por algunas características técnicas.

De 1974 a 1976 se dio un progreso muy importante en el desarrollo de las tarjetas inteligentes gracias a las patentes que obtuvo *Roland Moréno* en Francia. Todavía no era posible producir las tarjetas en masa y a bajo costo debido a dificultades técnicas, las cuales año con año han ido desapareciendo. Estas patentes expiraron en 1996, lo cual provocó un incremento en el número de compañías que se desenvuelven en esta área ya que los problemas de licencias con los que se enfrentaban anteriormente disminuyeron.

Roland Moréno tuvo un gran acierto al darse cuenta de que lo más importante era proporcionar un medio de almacenamiento que estuviera protegida por una lógica de control de acceso. Desde entonces el desarrollo de las tarjetas inteligentes ha sido lento pero con paso firme.

Otro momento brillante de las tarjetas inteligentes se dio en 1984, en Francia, cuando se realizó una prueba de campo en la aplicación específica de la telefonía pública fija. De 1984 a 1985 se probaron diversas tecnologías entre las que se encontraban las tarjetas magnéticas, las tarjetas ópticas y las tarjetas inteligentes para su uso como tarjetas telefónicas. El resultado fue que las tarjetas inteligentes mostraron tener un mayor nivel de flexibilidad, confiabilidad y seguridad.

En 1995 el número de tarjetas vendidas al rededor del mundo superó los cuatrocientos millones. En el año 2000 las ventas superaron los mil seiscientos millones.

Muchas veces se puede obtener una mayor flexibilidad y seguridad cuando se emplea más de una

[1] Esta técnica se conoce comúnmente como clonación de tarjetas. Hay dispositivos muy sencillos de venta en Internet que permiten hacerlo sin la sospecha de los clientes.

tecnología en las tarjetas. Las *tarjetas híbridas* combinan dos o más de los mecanismos descritos. Es posible utilizar tarjetas inteligentes que además posean una banda magnética, una fotografía, la firma y el nombre de la persona a la que pertenece. Cuando se desea un nivel de seguridad más elevado, muchas veces se emplean además mecanismos de identificación basados en información biométrica[2].

La evolución de las tarjetas inteligentes se vio muy influenciada por los estándares existentes. En particular se respetaron las características físicas y de la escritura en relieve que habían sido establecidas en los estándares para tarjetas plásticas.

A.2. Componentes de una Aplicación Real Basada en Tarjetas Inteligentes

Hasta ahora se ha hablado de las tarjetas inteligentes sin considerar explícitamente el ambiente en el que se desenvuelven. Las tarjetas inteligentes no pueden operar independientemente, sino que requieren de algún sistema externo que les proporcione la alimentación y la señal de reloj y con el cual puedan comunicarse y recibir comandos o transmitir información.

Las tarjetas inteligentes son solamente un componente de un sistema más grande y complejo. Es importante conocer con que otros componentes interactúa la tarjeta y saber que protocolos se utilizan con el propósito de tomar las medidas necesarias para preservar los atributos de seguridad de la información, de lo contrario se corre el riesgo de abrir una brecha de seguridad. Las especificaciones de seguridad de las aplicaciones en que se empleen las tarjetas inteligentes deben hacerse a priori, tomando al sistema completo y no se debe pensar que con el simple hecho de añadirlas al sistema éste incrementará automáticamente su nivel de seguridad.

Los componentes principales de un sistema basado en Tarjetas inteligentes –sin incluir a éstas– son (no todos los componentes son requeridos):

- *El Lector*
- *El Tenedor de Tarjeta*
- *El Emisor de Tarjetas*
- *El Sistema Anfitrión*
- *La Red*

[2]Existen tres métodos principales para autenticar (o identificar) a una persona. La primera forma es con base en algo que se sabe (por ejemplo una clave o password). La segunda forma se basa en algo que se tiene (por ejemplo una tarjeta o una llave). La tercera forma de identificar una persona es con base en lo que se es. Esta última forma se consigue gracias a las comprobaciones biométricas, las cuales consisten en asegurarnos que una persona es quien dice ser con base en sus características intrínsecas. Existen muchos tipos de comprobaciones biométricas entre las que se encuentran técnicas fisiométricas (huellas digitales, geometría de la mano, patrones en la retina y el iris, reconocimiento de rostros, etc.) y las técnicas conductistas (verificación de firma, forma de escribir en un teclado, reconocimiento de voz, etc.).

El uso de PINs en tarjetas magnéticas y tarjetas inteligentes es muy común, sin embargo el nivel de seguridad que proporciona esta forma de autenticación es muy bajo. Por otro lado es posible utilizar los mecanismos de autenticación biométricos en conjunción con las tarjetas inteligentes y el resultado es un sistema mucho más confiable y seguro. La tarjeta inteligente puede almacenar algunos cientos de bytes en los que se incluye la información de algunos datos biométricos sencillos con lo que los errores de autenticación pueden reducirse considerablemente.

El Lector

Las tarjetas inteligentes requieren antes que nada de algún tipo de interfaz eléctrica que les permita obtener las señales que necesitan para su operación. El dispositivo que se encarga de proporcionarles estas señales y de interactuar física y eléctricamente con ellas es llamado *lector de tarjetas*. Otros nombres con los que se les conoce son *Terminal*, *Dispositivo de Interfaz (Interface Device o IFD)*, *Dispositivo Acepta Chips (o Tarjetas) (Chip(Card)-Accepting Device o CAD)*, *Lector de Tarjetas de Chip (Chip-Card Reader o CCR)* y *Unidad de Lectura-Escritura (Read-Write Unit o RWU)*.

Los lectores varían en cuanto a su complejidad y funcionalidad. Algunos pueden considerarse casi únicamente como una interfaz serial, un reloj y una fuente de voltaje[3], otros tienen componentes más sofisticados como microprocesadores y módulos de seguridad.

Cuando los lectores cuentan con un procesador es posible programarlos. Normalmente los desarrolladores del lector o terceros lo hacen en lenguajes de alto nivel como *C* o *C++*. Sin embargo se presentan dificultades en cuanto a la estandarización de los procedimientos a utilizar para transmitir los programas. Si por un lado hay una falta de estándares estrictos que regulen los mecanismos de operación de las tarjetas, para el caso de los lectores esta deficiencia es mayor.

Algunas de las soluciones a este problema son la *Europay Open Terminal Architecture (OTA)* que emplea un intérprete de *Forth*, *Java for Terminals* y las especificaciones de *EMV* las cuales abordan el concepto de código de programa cargable.

Como los lectores pueden emplearse en una gran variedad de aplicaciones es natural que existan diversos tipos. Una de las clasificaciones más comunes que se encuentra en la literatura separa a los lectores de acuerdo a su movilidad en:

- *Portátiles*
  Los lectores portábles operan mediante baterías y proporcionan la mayor movilidad. Para lograr la independencia que requieren, normalmente presentan también displays, teclados e incluso bocinas e impresoras. El uso de teclados y displays incluidos en el lector puede aumentar nuestro nivel de confianza en la seguridad del sistema porque la información confidencial que observemos y proporcionemos no tiene que pasar a través de otro sistema posiblemente inseguro como lo son los hosts o sistemas anfitriones.

- *Fijos*
  Normalmente están conectados a un sistema anfitrión o a la red. En ocasiones también cuentan con interfaces de entrada y salida como los teclados y displays pero esto no es siempre necesario ya que la misma funcionalidad de estos aditamentos la puede proporcionar el sistema anfitrión. El sistema anfitrión puede servir como intermediario entre el usuario y la tarjeta inteligente, además de permitir el uso de periféricos que incrementen la seguridad mediante información biométrica.

Otra clasificación de los lectores considera la forma en que son empleados desde el punto de vista de la comunicación con un sistema de más alto nivel y distingue dos tipos de los lectores:

- *En Línea*
  Estos lectores poseen una conexión permanente con un sistema central o de más alto nivel que participa activamente en la operación de la tarjeta. La ventaja fundamental de estos sistemas es que se facilita la sincronización de la información y se tiene un mayor control sobre la integridad.

[3]Incluso es posible crear lectores más sencillos ya que se puede utilizar un adaptador para conectar la salida del puerto paralelo de la computadora directamente a los contactos de las tarjetas inteligentes y crear todas las señales mediante software en la PC.

- *Fuera de Línea*
  Estos lectores poseen una mayor independencia que los lectores en línea ya que pueden operar sin requerir un enlace con otro sistema. En realidad estar fuera de línea se convierte en una desventaja debido a que no es fácil actualizar la información que varía con el tiempo y que los sistemas requieren para su operación[4]. Por ello normalmente se permite el intercambio de información ocasional con sistemas externos. De esta forma se combinan varias de las ventajas de ambos enfoques como lo son la mayor independencia de los sistemas fuera de línea junto con un menor costo asociado al emplear un enlace poco frecuentemente.

Debido a la popularidad de las PCs, y a la frecuencia con que se emplean en conjunción con las tarjetas inteligentes, mencionaremos algunos de los aspectos que involucran a ambas tecnologías. Primero que nada hay que dejar muy claro que las tarjetas inteligentes no requieren de una PC para funcionar, aunque es frecuente que se empleen con una. El lector puede encontrarse en muy diversas formas. Puede ser un dispositivo separado conectado a la PC mediante una interfaz serial, paralela (poco frecuente) o *USB*. Algunos lectores emplean la misma vía de comunicación que los teclados o ratones. Otras presentaciones de los lectores incluyen las tarjetas *PCMCIA*, los cuales permiten emplear las tarjetas inteligentes convenientemente en computadoras portátiles. Los disquetes–lector de tarjetas emplean la misma interfaz de los disquetes para comunicarse con la computadora y permiten utilizar el *drive* lector de disquetes común para comunicarse con la tarjeta.

Respecto a los estándares para la comunicación con el lector sobresalen *PC/SC* desarrollado principalmente por *Microsoft* junto con varias empresas líderes en el mercado de las tarjetas inteligentes y *OCF* que fue desarrollado por *IBM* y se encuentra en sistemas que emplean *Java*.

En cuanto a las medidas de seguridad empleadas en los lectores, éstas varían de acuerdo al tipo de sistema. Si el lector solamente es un intermediario entre la tarjeta y el sistema anfitrión normalmente no es necesario implantar ningún mecanismo extra de seguridad. En este caso el lector es considerado un medio inseguro por el cual fluyen los datos. El anfitrión y la tarjeta inteligente serán los que directamente se encarguen de autenticarse y validar las transacciones.

Por el contrario, si se desea que el lector tenga un papel más activo o si se desea aumentar su seguridad, es necesario añadir los mecanismos de seguridad que permitan autenticar al lector para además mantener la confidencialidad e integridad de los datos que se le confían. Para ello se emplean técnicas como el uso de claves maestras bien resguardadas dentro del lector mismo, las cuales son utilizadas en algoritmos criptográficos para verificar, por ejemplo, la autenticidad de las tarjetas. No sorpresivamente, para resguardar estas claves se emplea frecuentemente tarjetas inteligentes, las cuales funcionan como módulos de aplicación seguros (SAMs) y poseen circuitería de seguridad para que en caso de detectar cualquier intento de sabotaje borren el contenido de su memoria y destruyan las claves, evitando que un adversario las robe.

El uso de SAMs tiene la ventaja de que aumenta la flexibilidad del sistema ya que éste puede modificarse con simplemente intercambiar el módulo. Por ejemplo, si tenemos un lector con un conjunto de algoritmos criptográficos fijo y deseamos emplear uno que no está presente, podemos evitar el costoso cambio de lector simplemente con reemplazar el módulo viejo por un módulo con un criptoprocesador capaz de ejecutar el algoritmo deseado.

[4]Un ejemplo de este tipo de información son las listas de tarjetas robadas o inválidas.

El Tenedor de Tarjeta

La tarjeta inteligente provee evidencia a otros componentes, como por ejemplo al sistema anfitrión, de que la persona que presenta la tarjeta verdaderamente es la persona que dice ser. Antes de que la tarjeta permita operaciones restringidas, como por ejemplo lectura y escritura de claves o archivos confidenciales, es necesario que el usuario compruebe su identidad.

Es responsabilidad de la tarjeta implantar los mecanismos necesarios para evitar que personas no autorizadas se hagan pasar por el tenedor de tarjeta legítimo. Hay que recordar que todos los mecanismos de verificación de identidad se basan en tres posibles aspectos: algo que se sabe, algo que se tiene o algo que se es (algo intrínseco). Existen diversas técnicas que permiten autenticar al tenedor de tarjeta:

- Uso de passwords y Números de Identificación Personal (NIPs[5]).
- Uso de otro tipo de información personal conocida por el tenedor de tarjeta y la tarjeta.
- Información Biométrica. Puede dividirse en dos tipos:
  - Conductista. Se presta atención a la forma en la que hacemos las cosas y que está muy ligada a características personales intrínsecas. Algunos ejemplos son: el habla, la firma (no solamente el resultado gráfico final, sino la forma en que la trazamos) y la forma de usar el teclado.
  - Física / Biológica. Se refiere a cuestiones inherentes a nuestro cuerpo. Los ejemplos más comunes son: huellas digitales, geometría de la mano, escaneo de la retina, escaneo del iris y reconocimiento de rostros.

  Hay que considerar que en general las características biométricas no son invariantes por lo que los sistemas que se encargan de su comprobación deben contar con mecanismos de adaptación, actualización o vías de autenticación alternativas.

La tarjeta inteligente puede permitir el acceso a diversos recursos de acuerdo al nivel de autenticación que el usuario esté dispuesto a ofrecer. Por ejemplo, si la integridad de cierta información contenida en la tarjeta es muy importante, un tenedor de tarjeta puede autenticarse solamente para leer la información cuando emplea un sistema en el que no confía. Aunque la tarjeta reciba comandos solicitandole que modifique esta información, la tarjeta se negará a hacerlo hasta que el tenedor de tarjeta provea las claves necesarias para habilitar la escritura.

El Emisor de Tarjetas

Antes de que la tarjeta llegue a manos del tenedor de tarjeta, es necesario que ésta sea inicializada y personalizada. Es el emisor de tarjetas quien se encarga de ello. También realiza ciertas tareas administrativas como lo son: la renovación, el desbloqueo (en caso de que la tarjeta haya recibido un cierto número de NIPs falsos), la recuperación de estados de error y cambiar algunas claves.

En realidad hay varias entidades que se pueden encargar de las tareas mencionadas, entre ellas se encuentran la empresa que manufactura la tarjeta, el propio usuario, u otra persona, pero no profundizaremos en una clasificación más precisa.

[5]En la literatura y en los capítulos referentes a la implantación del sistema en esta tesis se emplea el equivalente en Inglés de este término: PIN.

El Sistema Anfitrión (*Host*)

El uso de las computadoras personales se ha estado extendiendo continuamente durante las últimas décadas y estas son un componente muy frecuente de los sistemas basados en tarjetas inteligentes.

Las computadoras personales en general presentan un ambiente altamente inseguro y pueden comprometer la integridad del sistema completo si no se toman las medidas necesarias. Un caso típico es el ataque de un troyano. Al tener control sobre la PC, o al menos sobre la cuenta del usuario, puede robar los *passwords* que introduce el tenedor de tarjeta, realizar operaciones sin su consentimiento o engañarlo mostrándole información falsa. Por ello, el tenedor de tarjeta debe ser juicioso para determinar a que computadoras personales puede confiar su información.

Para evitar que NIPs secretos sean interceptados indebidamente, se puede emplear lectores de tarjetas que incluyen un pequeño teclado numérico. De esta forma el NIP no necesita pasar por el sistema *host*.

Uno de los posibles usos de las tarjetas inteligentes es precisamente el incrementar la seguridad del sistema *host*. Las claves simétricas y privadas almacenadas en el disco duro, o en memoria primaria, pueden ser fácilmente accedidas por un programa que se ejecute con los privilegios del usuario. Podemos evitar la pérdida de estas claves si las mantenemos guardadas dentro de la tarjeta y nunca salen de ella.

Existen diversos protocolos que permiten la interacción de la PC con las tarjetas inteligentes y los lectores. Uno de los primeros intentos se desarrolló en Alemania y permite la conexión lógica entre la PC y el lector. La especificación fue publicada en 1984 bajo el nombre de *Multifunktionales Kartenterminal (MKT)*, más conocida como MCT. Éste es un estándar de bajo nivel, particularmente útil para manejar tarjetas de memoria.

Otro estándar muy común es *PC/SC*[6], el cual fue desarrollado por el grupo de trabajo del mismo nombre y publicado en 1997.

Las aplicaciones para tarjetas inteligentes se desarrollan principalmente del lado del *host*. El *software* que se requiere por el lado del *host* consiste en:

- *Driver del lector.*
  Normalmente son rutinas de bajo nivel que permiten transmitir datos a la tarjeta y realizar operaciones particulares que algunos lectores soportan como por ejemplo expulsar la tarjeta.
- *Software que facilita la operación las aplicaciones.*
  Librerías que encapsulan al lector y permiten un grado de abstracción más elevado.
- *Aplicaciones.*
  Programas para la administración y para el usuario final.

El software del host no sufre de las restricciones impuestas al software de las tarjetas. Esto permite emplear una gran cantidad de lenguajes para su programación, especialmente se emplean los de alto nivel como: *C*, *C++*, *Java*, *Delphi* y *Visual Basic*, entre otros.

La Red

La red es un medio de comunicación extremadamente útil. Permite que la tarjeta intercambie información con el resto del sistema que se encuentra remotamente. Una posible aplicación de las tarjetas

[6]Su nombre oficial es "Especificación de Interoperabilidad para Tarjetas de Circuito Integrado y Sistemas de Computadoras Personales" (*Interoperability Specification for ICCs and Personal Computer Systems*).

inteligentes en sistemas distribuidos consiste en incrementar el nivel de seguridad de los mismos controlando los accesos mediante procedimientos de autenticación que requieran además del conocimiento de un password la posesión de la tarjeta.

Desde el punto de vista del host y la tarjeta inteligente debe considerarse a la red como un medio inseguro y hostil. Una de las maneras más eficientes de obtener las características de seguridad deseadas es la implantación de protocolos que crean un túnel seguro de punto a punto sobre el medio inseguro, tales como SSL.

Otro problema que puede aparecer son las fallas en la comunicación. Es posible que una transacción sea interrumpida antes de que se complete o que los datos lleguen corrompidos ocasionando posiblemente la pérdida de la información y de la sincronización. Es importante que existan mecanismos de recuperación que eviten que el sistema quede en un estado inconsistente.

A.3. Secuencia de Encendido y Apagado

La secuencia de encendido se ideó porque el proceso de inserción de la tarjeta y la conexión física entre los contactos no es del todo previsible desde el punto de vista eléctrico (no todos los contactos se activan simultáneamente), y para evitar el efecto de las señales transitorias que podrían llevar a la tarjeta a un estado erróneo.

La primer señal que se activa es el voltaje Vcc. Después de un tiempo indefinido se activa la señal del reloj. Luego de un tiempo t_2 el pin I/O se pone en alto (mandamos un alto a la tarjeta). La señal de reset pasa de bajo a alto donde permanece durante toda la operación normal de la tarjeta hasta que es apagada. El tiempo que pasa entre la subida de la señal de reset y el bajo de la señal I/O es t_1. En realidad no pone el lector la señal I/O en bajo sino que lo hace la tarjeta para transmitir la *respuesta a reset* (*ATR*).

Tanto la tarjeta como el lector utilizan la misma vía para leer y escribir datos, esto podría ocasionar un corto (si uno de ellos proporcionara un voltaje alto y el otro un voltaje bajo) pero esto no ocurre por la forma en que se da voltaje a este *pin*. Físicamente entre este *pin* y Vcc por el lado del lector hay una resistencia de $20k\Omega$, y cuando se desea escribir un voltaje alto nunca se presentan 5 o 3V sino que se da una alta impedancia. El Vcc conectado a través de la resistencia proporciona el voltaje requerido. La tarjeta hace algo parecido. Para escribir pone una alta impedancia o un voltaje bajo.

El valor de t_1 es como máximo el correspondiente a 200 ciclos del reloj externo. t_2 puede tomar valores entre 400 y 40000 veces el periodo del reloj.

Después del proceso de encendido la tarjeta manda la ATR (sección 1.3.2) que contiene información importante para identificar el protocolo de transmisión a emplear, de qué tipo de tarjeta se trata, de qué fabricante, etc. Luego ocurre un protocolo opcional llamado *PTS* (sección A.4) con el cual se negocia el protocolo y la velocidad de comunicación a utilizar. A continuación el sistema *host* manda una serie de comandos a la tarjeta los cuales son ejecutados y se transmite al sistema *host* la información generada por ellos.

Finalmente se efectúa la *secuencia de apagado* que consiste en poner en bajo la señal del reloj y la de *reset*. Después de un tiempo indefinido se retira el voltaje de alimentación y la tarjeta puede ser sustraída de forma segura.

Al proceso de encendido descrito se le llama *cold reset*. Si la tarjeta ya se encuentra en operación y la señal de reset se pone en bajo, cuando dicha señal se vuelva a poner en alto se producirá un *warm reset*.

A.4. Selección del Tipo de Protocolo (*PTS*)

Mediante la *ATR*, la tarjeta especifica los parámetros de comunicación que soporta y que son los más adecuados. Es posible que el lector desee emplear parámetros diferentes, para ello existe un procedimiento llamado *Selección del Tipo de Protocolo* o *PTS*[7].

Normalmente los lectores y tarjetas se seleccionan conjuntamente y operan correctamente sin requerir del *PTS*, por lo que en muchas ocasiones no se emplea. El *PTS* no es un procedimiento obligatorio.

Veamos como funciona este procedimiento. Primero que nada hay que distinguir dos modos de trabajo de la tarjeta:

- *Modo Negociable*.
  Los valores del divisor y el factor de ajuste permanecen en sus valores por omisión durante todo el procedimiento del *PTS*.

- *Modo No Negociable*.
  Los valores del divisor y el factor de ajuste especificados en la *ATR* toman efecto inmediatamente después de ser recibidos, por lo que toda la información transmitida durante el *PTS* debe estar de acuerdo a estos parámetros.

Para determinar si la tarjeta utilizará el modo negociable o el modo no negociable debe examinarse el carácter de interfaz TA_2 que pertenece a la *ATR*.

Bit(s)	**Descripción**
bit 8	Indica si se puede cambiar entre ambos modos (bit 8 = 0) o no (bit 8 = 1).
bit 7 y bit 6	Reservados para aplicaciones futuras, por ahora valen 0.
bit 5	Dice si los parámetros de transferencia fueron definidos explícitamente (bit 5 = 1) o implícitamente (bit 5 = 0) mediante los caracteres de interfaz de la *ATR*.
bit 4–bit 1	Nibble que indica el número de protocolo a utilizar.

Cuadro A.1: Descripción del carácter TA_2.

Todo el *PTS* puede considerarse como una secuencia de máximo 6 bytes (elementos de datos) los cuales se nombran a continuación y se muestran en el mismo orden que son transmitidos:

Byte	**Nombre**
PTSS	Carácter Inicial.
PTS0	Carácter de Formato.
PTS1, PTS2, PTS3	Caracteres de Parámetros.
PCK	Carácter de Comprobación.

Cuadro A.2: Elementos de datos del protocolo *PTS*.

La tarjeta recibe la cadena de bytes del *PTS* y si está de acuerdo manda de regreso esta misma información al lector. En caso de que no esté de acuerdo no regresa ningún valor, ocasionando que el lector haga un *warm reset* y vuelva a intentar alguna combinación de parámetros distinta. Esto debido a que no

[7] También llamado *Selección de Parámetros del Protocolo* o *PPS* (*Protocol Parameter Selection*).

es posible realizar más de una vez el *PTS* por cada vez que se haga un *reset*.

Veamos una descripción de cada uno de los bytes que constituyen el *PTS*:

- *PTSS*. Indicador de que el lector desea realizar el *PTS*, siempre vale 0xFF.
- *PTS0*. Especifica si los bytes *PTS1*, *PTS2* y *PTS3* estarán presentes. También indica el protocolo a utilizar.

  Veamos una descripción de los campos que contiene *PTS0* en la siguiente tabla:

Bit	Función
bit 8	Reservado para aplicaciones futuras. Por omisión vale 0.
bit 7	Avisa que PTS3 será transmitido (bit 7 = 1) o no (bit 7 = 0)
bit 6	Avisa que PTS2 será transmitido (bit 6 = 1) o no (bit 6 = 0)
bit 5	Avisa que PTS1 será transmitido (bit 5 = 1) o no (bit 5 = 0)
bit 4–bit 1	Nibble que contiene el número del protocolo a utilizar.

Cuadro A.3: Campos del byte *PTS0*.

- *PTS1* (opcional). Contiene la misma información que el byte TA_1 y está codificado exactamente de la misma manera, esta información se encuentra en la página 15.
- *PTS2* (opcional). Permite modificar el tiempo de guardia de acuerdo a la siguiente tabla:

Bit	Función
bit 8–bit 3	Reservados para aplicaciones futuras.
bit 2 y bit 1	Solamente se emplean 3 combinaciones: '00': No se necesita tiempo de guardia extra. '01': Tiempo de guardia = 255 etu. '10': Tiempo de guardia extra de = 12 etu.

Cuadro A.4: Significado del byte *PTS2*.

- *PTS3*. Reservado para aplicaciones futuras.
- *PCK*. Byte que permite detectar errores en la transmisión del *PTS*. Es el XOR de todos los bytes que componen el *PTS* hasta justo antes del *PCK*. Este byte siempre debe estar presente.

A.5. Estándares Relacionados con la Tecnología de las TIs

En general, un estándar es un documento, modelo o conjunto de reglas establecidas por una o varias autoridades competentes que son aceptadas como criterios para calificar algo. También se incluyen características y definiciones que nos aseguren que lo que se está evaluando sirva para el propósito que tiene establecido.

Una especificación es un documento que indica los requerimientos que algo debe cumplir. Muchas veces una especificación es una interpretación particular bien limitada de un estándar.

Debido a las características de los estándares, es frecuente que dos sistemas que cumplen el estándar no sean interoperables. Esto se debe a que hay muchos aspectos que el estándar cubre de manera burda y queda espacio para que las personas que desarrollan los sistemas lo interpreten o aprovechen a su conveniencia.

Existe una cantidad inmensa de estándares y especificaciones que pueden aplicarse directa o indirectamente a las tarjetas inteligentes. Sin ellos no hubiera sido posible el crecimiento del mercado tan grande que han experimentado.

Una de las organizaciones que más ha aportado al desarrollo de las tarjetas inteligentes es la *ISO*[8]. En particular gracias al estándar 7816 publicado junto con *IEC*[9].

Además de la ISO existen diversas organizaciones que en ocasiones trabajan conjuntamente para evitar duplicidad de estándares sobre un mismo tema y aumentar la cobertura del estándar. Entre ellas se encuentra la IEC (Comisión Electrotécnica Internacional) que se ha ocupado de aspectos relacionados con la electrónica. El CEN[10] (Comité Europeo de Normalización) ha trabajado junto con la ISO para desarrollar estándares reconocidos internacionalmente.

Cabe señalar que los estándares y especificaciones no son estáticos. Sufren revisiones para adecuarse mejor a las prácticas de la industria. A continuación daremos una lista de los estándares y especificaciones más importantes, con una descripción muy breve de ellos, solamente se presenta una pequeña parte de las decenas existentes:

- *ANSI X9.19* Códigos de autenticación de mensajes.
- *ANSI X9.31-1* Criptografía de clave pública para la industria de servicios financieros. En su parte uno trata sobre *RSA*.
- *EMV* Especificación de tarjetas de circuito integrado para sistemas de pago. Es un conjunto de estándares formados por tres partes:
  - La primera trata sobre las características mecánicas y eléctricas de las tarjetas y lectores, protocolos de comunicación, comandos, códigos de retorno, etc.
  - La segunda trata sobre el proceso de una transacción y las funciones de seguridad.
  - La tercera parte especifica algunos requerimientos obligatorios y otros opcionales para los lectores.

  Más información sobre *EMV* se presenta en la sección sobre aplicaciones de las tarjetas inteligentes (sección 1.4).
- *IEEE 1363:1998* Trata sobre *RSA*, *Diffie–Hellman* y criptografía de clave pública. Abarca diversos aspectos como la generación de claves, firmas digitales e intercambio de claves.
- *EN 726* Estándar que define los requerimientos para lectores y tarjetas empleados en telecomunicaciones. Cubre una gran variedad de aspectos desde el nivel más bajo hasta cuestiones más específicas para las telecomunicaciones, como por ejemplo aplicados a la telefonía. Hay cierto empalme con el estándar *ISO 7816*, y sirve como base para la especificación de *GSM*.

[8] Organización Internacional para la Estandarización ("*International Organisation for Standardization*"). Como nota curiosa, ISO no es un acrónimo sino que se deriva de una raíz Griega y es el nombre de una organización por lo que en todos los idiomas se escribe igual.

[9] *International Electrotechnical Commission*.

[10] *Comité Européen de Normalisation*.

- *EN 1546* Es el estándar más importante y de mayor aceptación mundial en cuanto a monederos electrónicos.
- *FIPS 46* El Estándar de Cifrado de Datos o *DES*. Para el propósito de esta tesis, este algoritmo es útil solamente como referencia histórica y punto de comparación.
- *PC/SC* Especificación para la Interoperatividad entre Tarjetas de Circuito Integrado y Sistemas de Computadora Personal. Esta especificación permite la comunicación entre lectores y computadoras personales. Una de las consideraciones es la independencia de plataforma.
- *GSM*[11] Es un conjunto de estándares que trata sobre protocolos de comunicación, los *SIM*[12] (*Módulo de Identidad del Subscriptor*) y otros temas.
- *ISO/IEC 7816* Tarjetas de circuito integrado con contactos. Es en nuestra opinión el estándar relativo a las tarjetas inteligentes más importante de todos. Muchos estándares se apoyan en éste. Aquí se presenta una descripción muy breve de cada una de sus partes:
  - *Parte 1:* Características físicas. Especifica las dimensiones, materiales, resistencia de las tarjetas y en que ambientes van a operar, además de las pruebas mecánicas que deben superar.
  - *Parte 2:* Dimensiones y localización de los contactos.
  - *Parte 3:* Señales electrónicas y protocolos. Define las características electrónicas necesarias para la transmisión de la información. Especifica el comportamiento de la tarjeta cuando se le da la señal de *reset*, y los protocolo de comunicación mas frecuentes (T=0 y T=1). También trata sobre cuestiones como negociar el voltaje de operación y la velocidad del reloj con el lector.
  - *Parte 4:* Comandos inter–industria para intercambio. Provee las especificaciones necesarias para crear aplicaciones que empleen comandos estándares. Especifica un sistema de archivos jerárquico que soporta diversos tipos de archivos y mecanismos para controlar el acceso a los mismos. Se describen los comandos que pueden aceptar las tarjetas (no todos son obligatorios) y la forma en que deben responder en caso de error.
  - *Parte 5:* Sistema de numeración y procedimiento de registro para identificadores de aplicación (AID).
  - *Parte 6:* Elementos de datos inter-industria.
  - *Parte 7:* Comandos interindustria para el lenguaje SCQL (*Structured Card Query Language*).
  - *Parte 8:* Comandos interindustria relacionados con la seguridad.
  - *Parte 9:* Comandos interindustria mejorados.
  - *Parte 10:* Respuesta a reset (*ATR*) para tarjetas síncronas (tarjetas de memoria).
  - *Parte 11:* Estructura de la tarjeta y funciones mejoradas para uso de aplicaciones múltiples.
- *ISO/IEC 9798* Tecnología de la Información –Técnicas de Seguridad– Autenticación de Entidades. Describe diversas técnicas de autenticación de entidades. Es un conjunto de estándares que ocupa desde criptografía simétrica hasta asimétrica y técnicas de conocimiento cero.

[11] *Global System for Mobile communications*.
[12] *Subscriber Identity Module*.

A.6. Futuro de las Tarjetas Inteligentes

Al igual que en la mayoría de las áreas de la tecnología, es difícil hacer un pronóstico preciso del futuro que les depara a las tarjetas inteligentes. Sin embargo es muy probable sigan las mismas tendencias que han seguido hasta ahora.

Existen cuatro factores principales que marcarán el camino de la tecnología de las tarjetas inteligentes [18]:

- *Aplicaciones*
  Una de las aplicaciones en las que posiblemente las tarjetas iteligentes tendrán un mayor desarrollo es en el comercio electrónico. Se espera que la mayoría de los sistemas que acepten tarjetas serán operados directamente por el usuario, de forma que el vendedor y el consumidor no tendrán que encontrarse en el mismo lugar.

  Los sistemas de prepago, como los monederos electrónicos, permitirán que sectores grandes que antes no eran muy tomados en cuenta, como lo son los menores de edad, puedan realizar compras electrónicas. Otra ventaja que incrementará la popularidad de los sistemas de prepago es que los costos asociados a las transacciones es reducido, lo que ocasionará un mayor uso de este tipo de tarjetas para operaciones cotidianas con montos pequeños, como lo son los pagos de parquímetros, estacionamientos, *tickets*, etc.

  También tendrán un papel más importante las tarjetas inteligentes como mecanismos de control de acceso a sistemas de cómputo, LANs (redes de área local) y la Internet. Debido a que *Microsoft* se ha enfocado a este mercado y a la gran cantidad de recursos que puede invertir, se espera que la cantidad de tarjetas empleadas con este fin aumente considerablemente.

- *Tecnología*
  A causa de la constante demanda de mayor poder computacional en las tarjetas, especialmente de las que soportan múltiples aplicaciones, se cree que habrá mejoras en la velocidad del reloj, el consumo de potencia y la cantidad de memoria. Hasta ahora, la mayoría de las tarjetas solamente soportan la comunicación *half–duplex*, pero pronto se popularizarán las tarjetas *full–duplex*, con lo que las aplicaciones podrán ser más versátiles al eliminarse el modelo maestro-esclavo.

  La velocidad de comunicación más frecuente hasta ahora es la de 9600 $baud/seg$, con la tecnología actual no es necesario que la comunicación sea tan lenta, sobretodo debido al uso de frecuencias de reloj más altas y de *hardware* de comunicación serial.

  De acuerdo a [24], para el 2005 la tecnología empleada para el circuito integrado será de 0.1 μm con un voltaje de operación de 1V (actualmente se utiliza tecnología de aproximadamente 0.35 μm con voltajes de 5V o 3V). La tecnología de 0.1 μm permitirá una mayor cantidad de transistores en la misma unidad de área, por lo que más y mejores componentes podrán incluirse.

  Por ahora los microcontroladores más utilizados son de 8 bits pero en poco tiempo serán sustituidos por microcontroladores de 32 bits.

  Con la tecnología actual es posible tener velocidades del reloj interno de al rededor de 50 MHz, especialmente con la tecnología RISC.

  La memoria *FRAM* en algunos años sustituirá a la memoria *EEPROM*, aunque hay que reconocer que esta tecnología todavía sufre de algunos problemas: Si bien en la *EEPROM* el número de escrituras está limitado y el de lecturas es virtualmente ilimitado, en las *FRAM* también hay un límite para número de ciclos de lectura o escritura combinados (al rededor de 10^{10} ciclos). Debido

a las propiedades de los materiales ferromagnéticos utilizados, aparecen limitaciones nuevas en cuanto al rango de temperatura en que pueden operar las tarjetas. La ventaja principal es que la velocidad de escritura de este tipo de tarjetas es mucho mayor y es igual a la velocidad de lectura, además de que permite mayores densidades con lo que se tendrán memorias de por ejemplo de 512 kBytes.

Los sistemas operativos evolucionarán convirtiendo a la tarjetas inteligentes en plataformas muy flexibles que soportarán aplicaciones múltiples, las cuales podrán ser cargadas después de que la tarjeta se encuentre en las manos del usuario [20]. Para ello se utilizarán sistemas operativos o APIs como *Multos* y *Java Card*. Éstos tendrán métodos para aislar las aplicaciones (como el *Firewall* de las *Java Cards*) que para ser más robustos posiblemente emplearán mecanismos de *hardware* como una Unidad de Manejo de Memoria (MMU[13]).

Los sistemas operativos abiertos permitirán que distintas tarjetas fabricadas por diversas empresas y con microcontroladores diversos puedan ser usadas con las mismas aplicaciones (a esto se le llama independencia del circuito integrado). Esto lo lograrán gracias al uso de máquinas virtuales.

- *Infraestructura*
  Se pronostica que la infraestructura requerida para operar las tarjetas inteligentes se incrementará significativamente, especialmente en los Estados Unidos que hasta ahora han estado atrás en este respecto. Será más común encontrar lectores de tarjetas inteligentes, posiblemente los encontremos incluidos en diversos aparatos, desde televisores hasta refrigeradores. Surgirán diversos dispositivos en los que la tarjeta inteligente podrá ser empleada. Un ejemplo es el ya presente "Visa viewer" con el cual se puede conocer el balance de la tarjeta y las últimas operaciones realizadas.

  Para facilitar el comercio electrónico, encontraremos frecuentemente lectores de tarjetas en teléfonos y computadoras.

- *Seguridad*
  Existen diversos ataques a los que está expuesta una tarjeta inteligente. Por una parte tenemos los ataques a los algoritmos criptográficos empleados, pero además tenemos problemas asociados con las implantaciones específicas de los mismos y de otros componentes del sistema. Los ataques más importantes a las implantaciones de las tarjetas son los de: medición de tiempos, inserción defectuosa, análisis de consumo de potencia, análisis de emisiones electromagnéticas y lectura directa de la memoria en el semiconductor. Además de otros más exóticos como alteración temporal del voltaje de operación o de la frecuencia del reloj externo.

  Para prevenir este tipo de ataques debe mejorarse la tecnología de los circuitos integrados de las tarjetas inteligentes para que detecten cualquier indicio de que se está tratando de realizar un ataque y puedan inutilizar la tarjeta o dejar ininteligible la información que contiene.

  En cuanto a los algoritmos de criptográficos, es necesario utilizar tamaños de clave adecuados y dar preferencia a los algoritmos más fuertes que se pueda pero manteniendo la facilidad de uso del sistema. La presencia de un coprocesador criptográfico incrementa el costo total del circuito integrado, es posible prescindir de él si utilizamos un procesador eficiente con instrucciones adaptadas a la criptografía. En cuanto a los algoritmos de clave asimétrica, *RSA* presenta el problema de que requiere demasiados recursos computacionales por lo que generar una clave puede tomar hasta 10 segundos y firmar hacer una firma 200 ms (con claves de 1024 bits), lo cual puede ser demasiado lento para algunas aplicaciones. Por otro lado la criptografía de curvas elípticas es más eficiente

[13] *Memory Management Unit*.

(en velocidad y tamaño) y fácil de implantar tanto en *hardware* como en *software*. Además ofrece el mismo nivel de seguridad que *RSA* pero con claves más pequeñas.

Actualmente, la API *Java Card 2.2* [39] especifica diversos algoritmos criptográficos que están disponibles para las aplicaciones. Entre ellos se encuentra el *AES* con sus tres tamaños de claves: 128, 192 y 256 bits, además de que se soporta el modo de operación *CBC*. Esta API también soporta otros algoritmos como *RSA*, basados en curvas elípticas y en su nueva versión algunas características exóticas como invocación remota de métodos (*RMI*) y elimina varias de las restricciones al subconjunto del lenguaje *Java* que estaban presentes en la versión 2.1.

Bibliografía

[1] Smart Card Alliance, *Current Press Releases*, http://www.smartcardalliance.org/about_alliance/press_10072002.cfm, Octubre 2002.

[2] ATMEL, *8-bit AVR Microcontroller with 8k Bytes In-System Programmable Flash. AT90S8515*, http://www.atmel.com/atmel/acrobat/doc0841.pdf.

[3] ATMEL, *AVR Instruction Set*, http://www.atmel.com/dyn/resources/prod_documents/DOC0856.PDF.

[4] ATMEL, *IAR Embedded Workbench for the ATMEL AVR AT90S*, http://www.equinox-tech.com/Downloads/PDF/Leaflets/LF-ewa90.pdf.

[5] AVR, *AVR ICE 200 Users Guide*, http://www.atmel.com/dyn/resources/prod_documents/doc1413.pdf.

[6] Friedrich Bauer, *Decrypted Secrets: Methods and Maxims of Cryptology*, segunda ed., Springer–Verlag, 2000.

[7] Cillian O'Driscoll B.E., *Hardware Implementation Aspects of the Rijndael Block Cipher*, Master's thesis, National University of Ireland, Octubre 2001.

[8] Matthias Brüstle, *SOSSE*, http://www.mbsks.franken.de/sosse/.

[9] Johannes A. Buchmann, *Introduction to Cryptography*, Springer–Verlag, 2000.

[10] Tony Bybell and the Amulet group, *Gtkwave Electronic Waveform Viewer*, http://www.cs.man.ac.uk/amulet/tools/gtkwave/index.html.

[11] Costas Christoyannis, *What is Cryprography*, http://www.hack.gr/users/dij/crypto/overview/whatis.html.

[12] Joan Daemen and Vincent Rijmen, *AES Proposal: Rijndael*, http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip.

[13] ———, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer–Verlag, 2001.

[14] Cipher A. Deavours, David Kahn, Louis Kruh, Greg Mellen, and Brian Winkel, *Cryptology. Yesterday, Today, and Tomorrow*, Artech House, 1987.

[15] Dorothy Denning, *Cryptography and Data Security*, Addison–Wesley, 1983.

[16] Henry Dreifus and J. Thomas Monk, *Smart Cards. A guide to building and managing smart card applications*, John Wiley and Sons Inc., 1997.

[17] David S. Dummit and Richard M. Foote, *Abstract Algebra*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[18] David Everett, *Smart Card Tutorial – Part 1 Smart Cards and the Future*, Documento disponible en Internet `http://www.smartcard.co.uk/resources/tutorials/sct-scatf.pdf`, Diciembre 1998.

[19] Joanne Fuller and William Millan, *On Linear Redundancy in the AES S-Box*, `http://eprint.iacr.org/2002/111.ps`, 2002.

[20] Gemplus, *Smart Cards Introduction, part 3: Smart card operating systems*, disponible en: `http://www.cs.uku.fi/kurssit/ads/SC_OperatingSystems.pdf`.

[21] Jovan Dj. Golc, *Multiplicative Masking and Power Analysis of AES*, `http://eprint.iacr.org/2002/091.ps`, 2001.

[22] Scott Guthery and Tim Jurgensen, *Smart Card Developer's Kit*, primera ed., Macmillan Computer Publishing, Febrero 1998.

[23] Gaël Hachez, François Kouene, and Jean-Jacques Quisquater, *cAESar results: Implementation of Four AES Candidates on Two Smart Cards*, `http://www.dice.ucl.ac.be/crypto/publications/1999/CAESAR.pdf`, Septiembre 2002.

[24] Mike Hendry, *Smart Card Security and Applications*, segunda ed., Artech House, 2001.

[25] Hitachi, *News Releases from Headquarters*, `http://global.hitachi.com/New/cnews/E/2002/0412/`, Abril 2002.

[26] Geoffrey Keating, *Performance Analysis of AES Candidates on the 6805 CPU Core*, `http://csrc.nist.gov/CryptoToolkit/aes/round1/conf2/papers/keating.pdf`, Abril 1999.

[27] Sungha Kim and Ingrid Verbauwhede, *AES Implementation on 8–bit Microcontroller*, AES_on_AVR.doc disponible en `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/atmal.zip`, 2003.

[28] Helger Lipmaa, Phillip Rogaway, and David Wagner, *Comments to NIST concerning AES Modes of Operation: CTR-Mode Encryption*, disponible en `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ctr/ctr-spec.pdf%`.

[29] Michael Luby, *Pseudorandomness and Cryptographic Applications*, Princeton Computer Science Notes, Princeton Academic Press, 1996.

[30] Alfred Menezes and Paul Oorschot, *Handbook of Applied Cryptography*, CRC Press, 1997.

[31] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan, *Investigations of Power Analysis Attacks on Smartcards*, Proceedings of USENIX Workshop on Smartcard Technology, Mayo 1999, `http://citeseer.nj.nec.com/messerges99investigations.html`, pp. 151–161.

[32] Movement for the Use of Smart Cards in a Linux Environment MUSCLE, *Sitio electrónico oficial*, http://www.linuxnet.com.

[33] Wired News, *Pirates Cash In on Weak Chips*, disponible en http://www.wired.com/news/technology/0,1282,12459,00.html, Mayo 1998.

[34] NIST, *Modes of Operation*, disponible en: http://csrc.nist.gov/CryptoToolkit/modes/.

[35] Wolfgang Rankl and Wolfgang Effing, *Smart Card Handbook*, tercera ed., John Wiley and Sons, Inc., 2000, Nombre original: *Handbuch der Chipkarten*.

[36] Theodore A. Roth, *Simulavr: an AVR simulator*, http://savannah.nongnu.org/projects/simulavr/.

[37] Bruce Schneier, *Applied Cryptography*, segunda ed., John Wiley and Sons, Inc, 1996.

[38] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall, *Performance Comparison of the AES Submissions*, http://www.counterpane.com/aes-performance.pdf, Febrero 1999.

[39] Inc. Sun Microsystems, *Java Card 2.2 Platform Specification*, http://java.sun.com/products/javacard/specs.html.

[40] Kim Sungha, *Página electrónica personal*, http://www.ee.ucla.edu/~yevgeny/index.html.

[41] First World War.com, *Primary Documents: Zimmerman Telegram, 19 January 1917*, http://www.firstworldwar.com/source/zimmerman.htm, Noviembre 2001.

[42] Doug Whiting, Russ Housley, and Niels Ferguson, *Counter with CBC–MAC (CCM) AES Mode or Operation*, disponible en http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ccm/ccm.pdf.

[43] PC/SC Workgroup, *Sitio electrónico oficial*, http://www.pcscworkgroup.com.