Hacia la especificación y verificación formal de algoritmos criptográficos: Mini-AES certificado

Informe de Proyecto de Grado
Presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Montevideo, 2014

Autores

Mauricio Martínez
Enrique Rodríguez

Tutores

Gustavo Betarte
Juan Diego Campo
Carlos Luna

Facultad de Ingeniería, Universidad de la República

Resumen

El algoritmo de encriptación de información AES (Advanced Encryption Standard) es un estándar internacional ampliamente utilizado tanto por gobiernos como comercialmente; por lo tanto resulta de importancia crítica garantizar el correcto comportamiento de sus distintas implementaciones. Existe una versión simplificada del algoritmo, llamada Mini-AES, que fue creada con fines académicos pero sin perder la esencia de su funcionamiento y sus propiedades criptográficas. En este trabajo se define un lenguaje de programación imperativa, se implementa Mini-AES sobre el mismo y se demuestra formalmente que su comportamiento es el deseado, esto es, que los procedimientos de encriptar y desencriptar son inversos. El lenguaje de programación en el cual se define el algoritmo Mini-AES se formaliza en el asistente de pruebas Coq, que es utilizado también para desarrollar pruebas sobre el comportamiento de dicho algoritmo, utilizando lógica de Hoare. Se presenta entonces una especificación formal del algoritmo Mini-AES junto con su correspondiente implementación funcionalmente correcta.

Índice

1. Introducción

Dada la masificación actual de sistemas informáticos de toda categoría, ya sean estos empresariales, gubernamentales, relacionados a la salud, redes sociales, almacenamiento en la nube, o de esparcimiento personal, y la importancia de los mismos en cuanto a la información que contienen, resulta crítico que su funcionamiento sea seguro y confiable. La gran mayoría de estos sistemas manipulan de alguna forma información que puede llegar a ser utilizada con fines que resulten perjudiciales a quienes los emplean, o simplemente con un fin distinto al original. Para poder garantizar su uso confiable, dichos sistemas deben muchas veces proveer garantías de privacidad de la información o de autenticidad de origen [28]. En computación ello implica el manejo de primitivas criptográficas. La criptografía es un área de estudio amplia que provee herramientas sobre las cuales se puede basar la computación para garantizar las necesidades mencionadas [22]. Pero existe una distancia entre la teoría en criptografía y su aplicación práctica en la computación, ya sea debido a limitaciones físicas que pueden ser de espacio o tiempo, o simplemente debido a errores humanos al aplicar la teoría.

La motivación de este trabajo es acortar la distancia entre la teoría y su aplicación práctica para poder contar con sistemas informáticos que sean seguros. Para ello es necesario validar los lenguajes de programación y los algoritmos escritos en ellos, utilizando herramientas teóricas como modelos y demostraciones formales.

1.1. Objetivos del trabajo

Los objetivos de este trabajo son diseñar un lenguaje de tipo imperativo e implementar sobre éste los algoritmos AES (Advanced Encryption Standard) [24] y Mini-AES [30] para que sirvan como implementaciones de referencia. Además para Mini-AES se pretende probar formalmente que la implementación de referencia es correcta funcionalmente. Entendemos por corrección funcional al comportamiento del algoritmo desde el punto de vista de la relación entre su entrada y su salida. Se espera que los procedimientos de encriptar y desencriptar sean inversos, es decir que al encriptar una entrada y desencriptar el resultado se obtenga una salida idéntica a la entrada original.

1.2. Trabajos relacionados

A la fecha de la publicación de este trabajo no encontramos otros trabajos que se encuentren vinculados a esta temática y que sigan el mismo enfoque. Por lo tanto consideramos al presente un aporte en el área, a partir del cual se puede continuar en la línea de investigación. A continuación mencionamos algunos proyectos cuyos objetivos se encuentran relacionados de alguna forma con el presente.

CompCert

CompCert [18] es un trabajo muy completo en el que se verifica la corrección de compiladores de algunos lenguajes de programación. Su principal resultado es *CompCert C*, un compilador de ISO C90 / ANSI C que genera código para las arquitecturas PowerPC, ARM y x86. La importancia de *CompCert* radica en brindar un compilador que se encuentra verificado mediante pruebas matemáticas que garantizan la correspondencia entre la semántica del código fuente y el código generado. El compilador completo se encuentra especificado, programado y verificado en Coq, y es liderado por Xavier Leroy, quien comenzó el proyecto en 2005. Actualmente se encuentra en desarrollo y se cuenta con la verificación formal del 90 % de los algoritmos del compilador.

El enfoque de este proyecto es obtener un compilador certificado. Al utilizar un compilador certificado se garantiza que el código generado es una traducción sin errores desde el lenguaje original hacia el objetivo, pero no se puede probar que el algoritmo en el lenguaje original se encuentre correctamente implementado.

Ynot

Ynot [9] es una librería desarrollada en Coq [21] que permite escribir programas imperativos y verificarlos. Esta librería implementa una extensión de la lógica de Hoare, denominada lógica de separación y cuenta con tácticas automatizadas que permiten probar fórmulas de este estilo. El proyecto se encuentra soportado por la National Science Foundation y Microsoft Research.

CertiCrypt

CertiCrypt [15] es una serie de herramientas para la construcción y verificación de algoritmos criptográficos. Soporta patrones comunes para razonar en criptografía, y ha sido utilizada para probar la seguridad de algunas construcciones, incluyendo esquemas de firma digital y funciones de hash. Su principal resultado es un framework verificado formalmente sobre Coq. Se basa en un lenguaje de programación extensible, probabilista e imperativo. Este proyecto fue fundado por el INRIA-Microsoft Research Joint Centre y el proyecto del Agence Nationale Recherche *SCALP* entre otros.

1.3. Estructura del documento

En el resto de esta sección se muestra un breve marco conceptual de criptografía en general, se introducen los algoritmos AES y Mini-AES. También se explican algunos conceptos del asistente de pruebas Coq [21], así como el modelado de lenguajes sobre el mismo.

En la sección 2 se presenta el lenguaje desarrollado para el trabajo, donde se detalla su sintaxis y semántica, además del módulo de matrices que fue necesario para poder completar la implementación. La sección 3 explica el funcionamiento detallado del algoritmo AES y Mini-AES, para el último se muestra su implementación en el lenguaje explicado anteriormente. Luego en la sección 4 se introducen los conceptos básicos de la lógica de Hoare así como su utilización en nuestro caso de estudio. En la sección 5 se explica la verificación del algoritmo. Finalmente en la sección 6 se presentan las conclusiones y el trabajo a futuro. A continuación de las referencias bibliográficas se incluyen los anexos A, B y C que contienen la especificación completa del lenguaje, la implementación completa de Mini-AES y la prueba completa de corrección, respectivamente.

1.4. Marco conceptual

1.4.1. Criptografía simétrica

Se define texto *plano* como un mensaje o información en su forma original y texto *cifrado* o *encriptado* como la codificación del texto plano en una cierta forma que aparente ser un flujo de información aleatoria o sin sentido, pero que podrá ser procesada solamente por entidades autorizadas para recuperar el texto plano. También se define *encriptación* o *codificación* como el proceso de transformar texto plano en texto cifrado y *desencriptación* o *decodificación* como el proceso de recuperar el texto plano a partir del cifrado. Los esquemas utilizados para encriptar, que se denominan *sistemas criptográficos*, constituyen un área de estudio llamada *criptografía*

y las técnicas para desencriptar un texto cifrado sin conocimiento de los detalles de encriptación forman el área de *criptoanálisis*. Las áreas de criptografía y criptoanálisis en conjunto son llamadas *criptología*.

Durante el proceso de encriptación y desencriptación, los algoritmos criptográficos utilizan un parámetro de entrada especial denominado *clave* el cual determina la salida específica del algoritmo, haciendo que el mismo texto al ser encriptado con dos claves distintas produzca salidas distintas. Se puede ver entonces al proceso de encriptación como una función que recibe de entrada un texto plano y una clave y produce como salida un texto cifrado. En el sentido opuesto, la desencriptación es una función que realiza el procedimiento inverso y, dado un texto cifrado junto con una clave, devuelve el texto plano original. Resumiendo, dados un texto plano X y las claves K_e y K_d, las funciones de encriptación y desencriptación, denominadas e y d respectivamente, deben cumplir:

$$e(X, K_e) = Y \ \rightarrow \ d(Y, K_d) = X \tag{1}$$

Actualmente existen dos modelos mediante los cuales se categorizan los algoritmos de encriptación: por un lado los algoritmos *simétricos* o *de clave privada* y por otro los *asimétricos* o *de clave pública*. La principal diferencia entre ambos modelos radica en que para los algoritmos simétricos se cumple $K_e = K_d$, es decir que se utiliza la misma clave tanto para el procedimiento de encriptar como el de desencriptar, mientras que los algoritmos de clave pública utilizan dos claves distintas. La mayor utilidad práctica de contar con dos claves diferentes es poder publicar la clave mediante la cual se encripta —clave pública, de forma que cualquiera pueda encriptar y enviar texto codificado pero solamente el receptor de dicho texto, haciendo uso de la otra clave que solamente él conoce —clave privada, pueda desencriptar e interpretar el mensaje original. Si bien los algoritmos de clave pública tienen más utilidades que las aquí mencionadas, nuestro objetivo es trabajar con AES que es simétrico y por lo tanto no profundizaremos en dicho tema.

Volviendo a la ecuación 1, se define $X = [X_1, X_2, ..., X_N]$ donde X_i es una letra de un alfabeto finito, que en nuestro contexto es el alfabeto binario $\{0, 1\}$. De la misma forma, $K = [K_1, K_2, ..., K_L]$ y $Y = [Y_1, Y_2, ..., Y_M]$, notar que a la clave al ser única es denominada sin subíndice, simplemente K. Con el mensaje X y la clave K se puede utilizar la función e para generar el texto cifrado $Y = e(X, K)$ y luego quien posea la clave —que se espera sea el destinatario del mensaje, puede utilizar la función d para acceder a $X = d(Y, K)$. Ahora bien, un atacante observando Y puede intentar acceder a X o K o ambos. Si está interesado en recuperar solamente el actual mensaje entonces intentará recuperar X generando un texto plano estimado $\hat{X}$. Sin embargo, por lo general el atacante está interesado en leer también futuros mensajes y para ello debe recuperar K generando una clave estimada $\hat{K}$.

Los algoritmos criptográficos más utilizados basan su seguridad en la clave y no en el algoritmo en sí mismo. Esto quiere decir que conocer el algoritmo además del texto cifrado no debe ser de ninguna ayuda para descifrarlo. Para generar $\hat{K}$ existen dos enfoques: **criptoanálisis** y **fuerza bruta**. El criptoanálisis busca aproximarse a la clave o a un texto plano basándose en las características del comportamiento del algoritmo, analizando posibles debilidades en su funcionamiento que puedan ser explotadas por un atacante. Por otro lado la fuerza bruta consiste en intentar con cada posible combinación de clave hasta encontrar un texto plano adecuado, notar que en promedio se puede dar con la clave al probar con la mitad de todas las combinaciones posibles.

1.4.2. Propiedades de un algoritmo seguro

Para que un algoritmo simétrico sea considerado seguro, un atacante no debe ser capaz de decodificar el texto cifrado o descubrir la clave, aún teniendo una cantidad de textos planos con

sus correspondientes textos cifrados. En cuanto al manejo de la clave, tanto quien encripta como quien desencripta el texto deben haber obtenido la clave de forma segura y así deben mantenerla. Si un atacante descubre la clave y conoce el algoritmo, toda comunicación podrá ser leída por el mismo.

Las razones por las cuales AES realiza las operaciones que estudiaremos, se basan fuertemente en conceptos de criptografía. Presentamos aquí brevemente las nociones mínimas necesarias para comprender dichas operaciones ya que profundizar en dichos temas escapa al alcance de este trabajo pero se puede estudiar en mayor profundidad estos temas en [28].

Confusión Esta propiedad implica que no debe existir una relación sencilla entre la clave y el texto cifrado, por lo tanto cada caracter del texto cifrado debe depender de la mayoría de los bits de la clave. Esto hace que sea difícil descubrir la clave, aún teniendo acceso a un conjunto grande de parejas de texto plano junto con su versión cifrada, todas utilizando la misma clave —este tipo de ataque se categoriza como *known-plaintext*. Por ello es importante que en la práctica cada bit del texto cifrado dependa de todos los bits de la clave y en diferentes maneras de cada uno.

Difusión Por otro lado, la relación entre el texto plano y el texto cifrado debe ser también compleja. Más precisamente, dado un texto plano cualquiera de entrada y su correspondiente texto cifrado de salida, al cambiar un bit en la entrada la probabilidad de cambiar para cada bit de la salida es 0.5, o sea que al menos la mitad de los bits deben cambiar al tomar el promedio de todos. Esta propiedad es importante para evitar los ataques categorizados como *chosen-plaintext* en los cuales el atacante tiene acceso a elegir textos planos arbitrarios para ser encriptados y obtener su cifrado.

No linealidad Siendo vista como una función, la encriptación de un bloque de texto plano no debe poder distinguirse de una permutación aleatoria de los bits. En caso de que esto no ocurriera, existe una técnica denominada *criptoanálisis lineal* que es capaz de encontrar transformaciones *afines* y mediante ciertos mecanismos descubrir bits de la clave. Junto con el *criptoanálisis diferencial* estas son las técnicas más utilizadas contra los algoritmos de cifrado por bloques.

Las dos primeras propiedades mencionadas —confusión y difusión— se logran por lo general utilizando lo que se llama una *red de sustitución-permutación*, y AES es un ejemplo de ello. Básicamente, este tipo de algoritmos consiste en una serie de operaciones matemáticas donde cada una de ellas se aplica a los bits de entrada y su salida es la entrada para la siguiente operación. Dichas operaciones son, como lo dice su nombre, sustituciones y permutaciones de los bits de entrada, veremos en AES cómo estas operaciones se llevan a la práctica.

1.4.3. El estándar AES

El denominado AES es una especificación para la codificación y decodificación de datos digitales. Su historia comienza en el año 1997 cuando el NIST (National Institute of Standards and Technology) decide realizar una convocatoria pública para recibir propuestas sobre un algoritmo de encriptación simétrico que fuera un estándar no clasificado y divulgado públicamente. Este algoritmo debía ser capaz de proteger la información del gobierno de los Estados Unidos para el siguiente siglo, incluyendo información de sensibilidad nivel *top secret*.

Su predecesor, el Data Encryption Standard (abreviado DES), fue seleccionado luego de una convocatoria de características similares realizada en 1973. En 1974, la multinacional IBM propuso un algoritmo diseñado por Horst Feistel, que sería seleccionado como estándar en 1976

y publicado oficialmente como Federal Information Processing Standard (FIPS) en 1977 por el NIST, que en ese entonces se llamaba NBS (National Bureau of Standards). Al ser elegido como un estándar por el gobierno, DES ganó adopción internacionalmente en muchas otras áreas y también fue estudiado y criticado en ámbitos académicos. A pesar de haber generado ciertas controversias debido a ciertos elementos de diseño que eran clasificados o a su reducido tamaño de clave de 56 bits, este algoritmo se consolidó como estándar internacional de encriptación simétrica durante los siguientes veinte años; su especificación aún se conserva en los archivos del NIST [23].

Para fines de la década de los 90 el tamaño de clave de 56 bits ya resultaba vulnerable a ataques de fuerza bruta, además de que DES fue pensado para correr en hardware y resulta poco eficiente a la hora de ser implementado en software. Por ello fue realizada la convocatoria por el NIST en 1997 y el proceso de selección se realizó en tres etapas que se pueden relacionar con las tres conferencias denominadas AES1, AES2 y AES3. Desde la convocatoria realizada en setiembre de 1977 hasta la primera conferencia, fueron presentados en total quince algoritmos: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, Rijndael, SAFER+, Serpent y Twofish. En las primeras dos conferencias, realizadas en agosto de 1998 y marzo de 1999 respectivamente, se discutieron las ventajas y desventajas de cada candidato en cuanto a seguridad y performance, se descartaron algunos de ellos y se llegó a los cinco denominados *finalistas*: Rijndael, Serpent, Twofish, RC6 y MARS. En la conferencia AES2 se realizó una votación en la cual Rijndael obtuvo 86 votos positivos y 10 negativos, seguido por Serpent el cual obtuvo 59 positivos y 7 negativos. En la última conferencia, AES3 realizada en abril del año 2000, un representante por el equipo de cada candidato realizó una presentación sobre las razones por las cuales consideraba mejor su algoritmo. Luego de un intenso debate, en octubre del mismo año el NIST anunció que había seleccionado a Rijndael y luego de realizar un borrador solicitando comentarios, en noviembre de 2001 se aprueba Rijndael como el FIPS 197 [24].

Rijndael fue diseñado por los belgas Joan Daemen y Vincent Rijmen —su nombre es una conjunción de los apellidos— y su propuesta puede encontrarse en [13]. Debido a su riguroso proceso de selección y al ser el sucesor de un estándar que llegó a durar veinte años en la industria, se espera que AES pueda ser utilizado quizás por otros tantos años más.

El algoritmo se categoriza como algoritmo simétrico y por bloques; simétrico por utilizar una misma clave tanto para el proceso de encriptar como para el de desencriptar y por bloques debido a que opera sobre grupos de bits de largo fijo, aplicando una transformación invariante sobre cada uno de dichos bloques. Las claves de AES pueden tener un tamaño de 128, 192 o 256 bits y el tamaño del bloque es 128 bits. La estructura del algoritmo es una red de sustitución-permutación ya que consiste en realizar un número de *rondas* determinado, aplicando una serie de operaciones matemáticas a cada bloque del texto en cada una de las mismas.

Hasta el día de hoy no existen ataques computacionalmente posibles contra AES y su complejidad es de $2^{126.1}$, $2^{189.7}$ y $2^{254.4}$ para claves de 128, 192 y 256 bits respectivamente, utilizando el denominado método biclique, que solamente es mejor que aplicar fuerza bruta en un factor de cuatro.

Otras de sus cualidades son su performance en cómputo y memoria, así como su simplicidad de implementar tanto en software como en hardware. Hoy existen procesadores relativamente económicos, de uso personal, que implementan una extensión de la arquitectura x86 llamada AES-NI (AES New Instructions) con instrucciones para ejecutar una ronda completa de encriptación o expandir la clave por ejemplo. Utilizando estas primitivas, existen librerías que pueden llegar a procesar texto a una velocidad de 700 MB/s por thread.

Debido a las cualidades mencionadas es uno de los algoritmos más populares, siendo utilizado por herramientas de compresión, servidores y navegadores web, protocolos de acceso remoto,

sistemas operativos y protocolos de comunicación en redes de área local, entre otros ejemplos. Es por esto que AES es considerado como un algoritmo de seguridad crítico.

1.4.4. Mini-AES

En el año 2002, casi inmediatamente luego de que fuera publicado oficialmente AES, el criptólogo de la *Multimedia University* e investigador del instituto *Swinburne Sarawak Institute of Technology*, instituciones de Malasia Raphael Chung-wei Phan , publicó un artículo denominado *Mini Advanced Encryption Standard (Mini-AES): A Testbed for Cryptanalysis Students* [30]. Allí se presenta una versión reducida del AES, simplificando los parámetros que se manejan en el mismo pero, como es aclarado en el mismo artículo, sin perder la estructura original. Basándose en Mini-AES es posible estudiar las propiedades de AES, de manera tal de concentrarse en las partes más relevantes para este trabajo.

1.4.5. Lenguajes para criptografía

Actualmente hay disponibles varias implementaciones de AES, se pueden encontrar librerías para los lenguajes más populares como lo son Java, C/C++, C#, Objective-c, PHP, Python, Ruby, Perl, Javascript, Haskell u OCaml. A pesar de poder encontrarse en cualquier tipo de lenguaje, las más utilizadas son las implementaciones en C o en su defecto en lenguajes basados en este como lo son C++, Java y C#. Por ejemplo, en C se pueden destacar las librerías PolarSSL [8], Libgcrypt [5] y OpenSSL [6] de las cuales las últimas dos se encuentran validadas por el NIST como correctamente implementadas —ver [25]. En cuanto a C++ se cuenta con Crypto++ [3] y Botan[1] de particular relevancia y la primera se encuentra también validada por el NIST. Por parte de la plataforma Java se puede nombrar la Java Cryptography Extension (JCE) que es un framework el cual debe ser proveído por la versión utilizada pero por ejemplo Oracle provee su propia implementación ver [7] y también Bouncy Castle [2] que es muy utilizada en la industria.

Estos lenguajes comparten como característica principal ser imperativos y estáticamente tipados. Poseen tipos de datos básicos para representar booleanos y enteros y realizar operaciones sobre estos —tanto operaciones aritméticas como bit a bit— y tipos más complejos como arreglos y a partir de ellos matrices. Además brindan estructuras de control como *if*, *for* y *while*.

1.4.6. El asistente de pruebas Coq

Se eligió *Coq* [21] como herramienta principal para el trabajo. Este es un asistente de pruebas, que sirve para expresar definiciones matemáticas así como escribir pruebas sobre las mismas y verificar dichas pruebas automáticamente. Además de esto permite escribir programas y ejecutarlos, especificar propiedades sobre los mismos y hasta extraer código ejecutable de la prueba realizada sobre la especificación.

Coq implementa un lenguaje llamado *Gallina*, basado en la teoría conocida como *Cálculo inductivo de contrucciones* (CiC), la cual a su vez es una extensión del *Cálculo de construcciones* [12]. Esta última fue creada por Thierry Coquand, quien también comenzó el desarrollo de Coq en el año 1984. En el CiC, los programas, sus propiedades y sus pruebas, son formalizados bajo el mismo lenguaje y por lo tanto todos los juicios lógicos terminan reduciéndose a juicios de tipado.

1.4.7. Modelado de lenguajes en Coq

Coq provee un lenguaje formal mediante el cual se puede modelar la sintaxis y semántica del lenguaje objetivo. Utilizando definiciones inductivas en Coq se puede modelar la sintaxis de un

lenguaje de forma clara y concisa. Por ejemplo la siguiente definición es válida en Coq y muestra una posible definición de expresión aritmética con operadores de suma, resta y multiplicación:

```
Aexp :=
  arith_num   : nat  → Aexp
  arith_plus  : Aexp → Aexp → Aexp
  arith_minus : Aexp → Aexp → Aexp
  arith_mult  : Aexp → Aexp → Aexp
```

Fragmento 1: Ejemplo de definición de expresión sintáctica.

Luego es posible escribir una expresión de la siguiente forma, que representa la noción más común de $3 * (2 + 4)$:

```
arith_mult (arith_num 3) (arith_plus (arith_num 2) (arith_num 4))
```

1.5. Notación utilizada

Debido a que el trabajo está fuertemente basado en Coq, muchas de las notaciones son tomadas directamente del lenguaje Gallina.

Expresiones lógicas

Para las expresiones lógicas son utilizados los operadores estándar: $\wedge$ (conjunción o *and*), $\vee$ (disjunción u *or*), $\neg$ (negación o *not*), $\rightarrow$ (implicación), $\forall$ (cuantificación universal) y $\exists$ (cuantificación existencial). Estas expresiones son escritas en cursiva y centradas como se muestra en el siguiente ejemplo:

$$\forall\ (i\ j:\ \mathbb{N}),\ \exists\ (m:\ \mathbb{N}),\ m > i + j\ \rightarrow\ m > i\ \wedge\ m > j$$

Código fuente

Por otro lado, las citas a código fuente como nombres de variables o tipos son escritas en letra estilo Courier, por ejemplo `int i ::= 0;` cuando se encuentra dentro de un párrafo. Al insertar fragmentos de código se utiliza un recuadro como el siguiente:

```
byte a ::= matrix[0][0];
matrix[0][1] ::= a;
```

Fragmento 2: Ejemplo de fragmento de código.

Tipos inductivos

Se ven a lo largo de este documento varios ejemplos de definiciones mediante tipos inductivos, esto no es casual, ya que este trabajo se basa fuertemente en el CiC. La sintaxis utilizada es simplemente: el nombre del tipo seguido de `:=` y la lista de constructores, uno por línea, como en el siguiente ejemplo:

```
Natural :=
  zero : Natural
  suc  : Natural → Natural
```

Fragmento 3: Ejemplo de definición de tipo inductivo.

Tipos y funciones

Para definir el tipo de un conjunto de símbolos se colocan sus nombres separados por espacios y dos puntos para introducir el tipo, por ejemplo `i j: Integer`. En el caso de requerir más de un tipo, se separan utilizando paréntesis como `(m n: Natural)(i j: Integer)`. Las funciones son definidas mediante su nombre, seguido de todos sus parámetros con sus correspondientes tipos, dos puntos para introducir (opcionalmente) el tipo de retorno y luego `:=` para separar de la definición. El siguiente es un ejemplo de función:

```
sum (m n: Natural): Byte := m + n
```

Fragmento 4: Ejemplo de definición de función.

Al aplicar una función, se utilizan espacios y asociación a la izquierda como se hace en Gallina o Haskell. Por ejemplo la función `sum` recién definida se aplica de la siguiente forma: `sum (sum 1 1) 5`.

1.6. Código fuente

Todo el código fuente generado como parte de este trabajo se encuentra disponible en [4]. Allí se puede encontrar un archivo de texto llamado *README* que contiene una breve explicación sobre cómo compilar los módulos y ejecutar las pruebas.

2. Lenguaje

Las implementaciones mencionadas en la sección 1.4.5 contienen código que puede resultar sumamente complejo de utilizar para realizar demostraciones formales como las que se plantean en este trabajo. Por otro lado, elegimos el algoritmo Mini-AES como objetivo para nuestras pruebas, ya que justamente simplifica la comprensión de AES. Aprovechamos entonces la oportunidad de tener que implementar Mini-AES para hacerlo sobre un lenguaje de programación definido en Coq. Esto nos brinda una mayor flexibilidad al realizar las pruebas y nos permite además tener una versión del algoritmo que puede servir como referencia para implementaciones en otros lenguajes.

Dado que precisamos un lenguaje para criptografía y en particular poder implementar los algoritmos AES y Mini-AES —presentados en [14] y [30]— es necesario analizar las principales características con las que cuentan los lenguajes utilizados para dicho propósito. Como ya fue visto en la sección 1.4.5, estos siguen el paradigma imperativo y por lo tanto debemos modelar primitivas fundamentales como lo son la asignación de valores a variables en memoria y la ejecución de secuencias de instrucciones.

Luego de estudiar el funcionamiento de los algoritmos —detallados en la sección 3— contamos con una visión más precisa sobre las funcionalidades que debe proveer nuestro lenguaje. Dichas funcionalidades abarcan básicamente manipulación de bytes: asignación en memoria, suma exclusiva, búsqueda en una posición de una matriz, asignación a una posición de matriz y estructuras de control y repetición.

Pero también resulta interesante para el trabajo que el lenguaje sea completo, para que se pueda continuar trabajando sobre el mismo. Por eso lo desarrollamos basándonos en [26] y extendiéndolo para que cuente con las siguientes características: tipos de datos entero, byte, booleano y matrices; así como manejo de variables, sentencias condicionales y loops. Contamos entonces con un modelo de lenguaje imperativo sobre el cual se puede implementar no solamente los algoritmos AES y Mini-AES, sino que teóricamente cualquier algoritmo —debido a que es Turing-completo [10]— y que en la prácitca resulta especialmente útil para algoritmos criptográficos.

Para que la sintaxis del lenguaje pueda ser reutilizada, decidimos implementarla de forma modular, definiendo en un módulo la sintaxis y manteniendo tanto a los tipos de datos como a sus operaciones de manera abstracta. Así, uno puede utilizar diferentes implementaciones de enteros, bytes, suma o resta por nombrar algunos ejemplos. Esto se puede lograr en Coq definiendo un módulo como `Module Type`, dentro del cual declaramos los tipos de datos y sus operaciones como parámetros. Luego hacemos a la sintaxis y semántica del lenguaje dependientes del módulo de tipos y éstas pueden utilizar los tipos de datos sin tener que conocer los detalles de su implementación.

A lo largo de la definición del lenguaje se puede ver que se realizan referencias a estos tipos de datos y sus operaciones. Los nombres de los tipos son introducidos en la sección 2.1.2 y los nombres de las operaciones se distinguen sencillamente, siendo el nombre del tipo y operación separados por un guión bajo como por ejemplo `int_plus`. En caso de modificar la implementación de alguno de ellos, no es necesario en absoluto modificar la definición del lenguaje.

En este trabajo utilizamos las implementaciones de la librería estándar de Coq donde nos es posible, la única excepción son las matrices, que son explicadas más adelante.

A continuación presentamos la sintaxis de nuestro lenguaje y luego la semántica.

2.1. Sintaxis

2.1.1. Estructura de un programa

Veamos primero cómo se compone a grandes rasgos un programa en este modelo. Básicamente, un programa consiste de un conjunto de variables globales y definiciones de procedimientos que pueden tomar parámetros. Las variables no contienen información de tipo, por lo cual al ser declaradas no se les asigna ningún tipo pero debe ser especificado a la hora de leer o escribir un valor. Los procedimientos se definen mediante su nombre, los parámetros entre paréntesis y a continuación el cuerpo del mismo.

Todas estas definiciones —identificadores y procedimientos— se introducen luego de su nombre mediante el símbolo `:=` y se separan por medio de un punto. Esto no es arbitrario ya que se trata de definiciones válidas en Coq, en el código fuente se puede observar que las mismas deben ser precedidas por la palabra clave `Definition`, que omitimos en este documento en favor de una mayor claridad.

```
two := id 1.

double(num: Id) :=
  int two ::= 2;
  int num ::= num * two;
.

plus_three(num: Id) :=
  int num ::= num + 3;
.

double_plus_three(num: Id) :=
  double num;
  plus_three num;
.
```

Fragmento 5: Estructura de un programa.

En el fragmento 5 se puede notar que siempre utilizamos parámetros para modificarlos, esto se debe a que no fue implementado el tipo de procedimientos con *retorno* de un nuevo valor, ya que esto agregaba mucha complejidad extra y escapaba del alcance de este proyecto. Por lo tanto se debe modificar una variable para realizar un efecto externamente visible.

También se puede observar que utilizamos símbolos característicos para las operaciones, como por ejemplo `*` para la multiplicación y `+` para la suma. Esto es posible realizarlo en Coq utilizando `Notation`s y fue utilizado en este trabajo para permitir escribir código más conciso; en cada subsección del anexo A.1 mostramos una tabla con los símbolos correspondientes a las operaciones.

2.1.2. Tipos de datos

Es imprescindible contar con números enteros para ser utilizados en contadores y loops entre otras operaciones fundamentales, así como valores binarios para utilizar por ejemplo en sentencias condicionales, es por ello que el lenguaje cuenta con los tipos `Integer` y `Bool`. Este lenguaje cuenta también con los tipos `Byte` y `Matrix`, que tienen claramente el objetivo de ser utilizados para AES y Mini-AES y serán de hecho los más utilizados en este trabajo. El tipo `Matrix` nos permite representar matrices de bytes aunque puede ser fácilmente extendido a matrices de otros tipos.

2.1.3. Identificadores

La forma más sencilla de representar las variables es viendo a la memoria como un arreglo de posiciones, donde un valor puede ser asignado en cualquiera de ellas. Podemos asignarle entonces a una variable un identificador, que no es más que un número natural indicando la posición ocupada en memoria; como consecuencia, si dos variables se encuentran identificadas por el mismo identificador entonces al escribir cualquiera de ellas estaremos escribiendo la misma posición en memoria y sobrescribiendo mutuamente los valores.

Para los identificadores creamos el tipo `Id`, este se trata simplemente de un tipo inductivo con un único constructor, que toma un número natural como parámetro:

```
Id :=
  id : nat → Id
```

Fragmento 6: El tipo de los identificadores

Por lo tanto `id 1` es una expresión de tipo `Id`, y al asignarle un nombre mediante una definición —como `two` en la primera línea del fragmento 5— decimos que se trata de una variable.

2.1.4. Expresiones

Las expresiones son las que van a permitir al programador representar valores correspondientes a los tipos de datos disponibles. Por ejemplo, un valor correspondiente a los números enteros, que en nuestro lenguaje es el tipo de datos `Integer`, puede ser un número literal como `12` o también puede ser una suma entre números enteros como `12 + 1`. Pero también se pueden representar valores de otros tipos, como por ejemplo `3 < 4`, el cual es de tipo `Bool` y es el resultado de la comparación entre ambos números enteros; o un valor de tipo `Byte` que es sea el resultado de acceder a una posición en una matriz de bytes, como por ejemplo `matr[0][0]`.

Para mostrar la forma en que definimos las expresiones del lenguaje, veamos el ejemplo de las expresiones de números enteros, a las cuales llamamos *expresiones aritméticas*. Las expresiones aritméticas nos permiten representar números enteros y sobre ellos aplicar las operaciones de suma, resta, multiplicación, división y modulo de números enteros; ya sean literales o variables que contengan números enteros.

La forma de definirlas es inductivamente, partiendo de números literales y variables como casos base y luego construyendo expresiones más complejas a partir de ellos. Un ejemplo de expresión compleja es la multiplicación, ya que para construir una expresión de multiplicación es necesario primero contar con dos expresiones aritméticas, las cuales a su vez pueden ser resultados de otras operaciones —incluyendo la misma multiplicación— y de esta forma continuar en la recursión hasta llegar a los casos base.

Veamos entonces una definición más formal de algunas de las expresiones aritméticas en forma inductiva.

```
ArithExp :=
  arith_exp_num  : Integer  → ArithExp
  arith_exp_id   : Id       → ArithExp
  arith_exp_mult : ArithExp → ArithExp → ArithExp
```

Fragmento 7: Definición de algunas expresiones aritméticas.

Aquí se puede ver como los constructores base `arith_exp_num` y `arith_exp_id` no requieren de otra expresión aritmética y nos devuelven una expresión válida. A partir de estos constructores y contando con variables y/o enteros literales, podemos construir expresiones aritméticas como

por ejemplo `arith_exp_mult (arith_exp_num 3) (arith_exp_id x)`. Utilizando la notación del cuadro 4 del anexo A.1 se puede escribir la misma expresión más naturalmente como `3 * x`.

Las expresiones de tipo `Bool` tienen una definición muy similar a la de las expresiones aritméticas ya que la lógica de la definición es la misma. Al igual que para los números enteros, para los booleanos los casos base son los literales y las variables, pero en los casos inductivos podemos encontrar más variedad, aunque no más complejidad. La diferencia es que dentro de las expresiones booleanas hay algunas que se construyen a partir de expresiones de otros tipos, en particular expresiones aritméticas. Ejemplos de ello son la comparación entre enteros o bytes, que se construyen a partir de un par de expresiones aritméticas o un par de expresiones de byte respectivamente. En el siguiente fragmento se pueden ver algunos ejemplos.

```
BoolExp :=
  bool_exp_lit : Bool     → BoolExp
  bool_exp_id  : Id       → BoolExp
  bool_exp_lt  : ArithExp → ArithExp → BoolExp
  bool_exp_gtb : ByteExp  → ByteExp → BoolExp
  bool_exp_not : BoolExp  → BoolExp
```

Fragmento 8: El tipo de las expresiones booleanas.

Un ejemplo de expresión booleana es el siguiente: `bool_exp_not (bool_exp_lt (arith_exp_num 3) (arith_exp_id x))` . Que al ser reescrito utilizando la notación del cuadro 5 del anexo A.1 queda `!(3 < 4)`.

Las expresiones de bytes y de matrices son análogas y se definen de la misma forma que lo hicimos con las expresiones aquí mostradas. Un ejemplo de expresión de byte es `byte_exp_shiftr (byte_exp_matrix matr (arith_exp_num 0) (arith_exp_num 1)) (arith_exp_num 1)`. O más simplemente utilizando la notación del cuadro 6 del anexo A.1: `matr[0][1] shr 1`. Y finalmente dos ejemplos de expresión de matriz: `matrix_exp_id mid` y `matrix_exp_init (arith_exp_num 2) (arith_exp_id two)`.

En el anexo A.1 se puede ver la definición formal completa del lenguaje, allí la notación utilizada es muy similar a la de Coq ya que es casi directamente lo que se encuentra en el código fuente.

2.1.5. Instrucciones

Las expresiones por si mismas representan un valor, ya sea expresado por ejemplo como una operación matemática o como el valor almacenado en memoria de una variable, pero para escribir programas que realmente puedan realizar tareas útiles es necesario contar con instrucciones, las cuales nos permitirán alterar el estado de la memoria y así generar efectos visibles externamente.

Estas instrucciones determinan acciones a realizar, que al ser ejecutadas podrán transformar el estado de la memoria de distintas formas. Definimos entonces un conjunto básico de instrucciones el cual nos permitirá operar de manera eficaz teniendo como objetivo los algoritmos AES y Mini-AES.

A continuación veremos una a una las instrucciones junto con su notación.

Skip Esta es la más sencilla de las instrucciones ya que al ser ejecutada no realiza ninguna acción. Puede resultar inútil desde el punto de vista de un programador, pero la utilizamos para algunas estructuras internas al lenguaje. No tiene notación asignada y en caso de utilizarla simplemente se puede escribir `instr_skip`.

Asignación Podríamos decir que la asignación es una de las instrucciones más importantes ya que al ser ejecutada esta modifica una posición en la memoria, de hecho junto con la modificación de posiciones de matrices, estas son las únicas instrucciones que modifican la memoria.

Más adelante veremos que al evaluar dicha instrucción se hará efectivo el cambio en la memoria pero en cuanto a sintaxis la asignación consta sencillamente de un tipo, una variable y una expresión del tipo dado.

Por ejemplo la asignación booleana se escribe de la siguiente forma:

```
bool X ::= bexp;
```

Donde `X` es un identificador y `bexp` una expresión booleana.

La notación utilizada fue definida en Coq para que quien escriba un programa en este lenguaje pueda hacerlo de forma más clara, pero la forma de definir formalmente las instrucciones de asignación es de manera inductiva, al igual que las expresiones. En el anexo A.1.2 se encuentra la definición formal de todas las instrucciones.

La asignación de enteros, bytes y matrices es análoga pero cambiando `bool` por `int`, `byte` o `matrix` respectivamente.

Modificación de una matriz Esta instrucción será de suma utilidad para nuestro algoritmo, la misma permite asignar un byte a una posición de un matriz, utilizando expresiones aritméticas para indicar la posición. La notación es la siguiente:

```
matr[i][j] ::= bexp;
```

Donde `matr` es un identificador que contiene un valor de tipo `Matrix Byte`, `i` y `j` son expresiones aritméticas y `bexp` es una expresión de tipo `Byte`.

Instrucción condicional IFB Siendo la primera de las instrucciones de control, esta instrucción permite ejecutar un bloque de código en función de una condición booleana, sin ejecutar nada cuando la condición no se cumple. Este es un ejemplo en el que utilizamos la instrucción `instr_skip`; para tener una sola implementación para el condicional con y sin alternativa `ELSE`. Entonces la notación para la instrucción es la siguiente:

```
IFB bexp THEN
  instr;
END
```

Donde `bexp` es una expresión booleana e `instr` es una instrucción.

Instrucción condicional IFB THEN ELSE Esta es como la anterior pero se agrega la posibilidad de ejecutar un bloque de código si la condición no se cumple. La notación es la siguiente:

```
IFB bexp THEN
  instr1;
ELSE
  instr2;
END
```

Donde `bexp` es una expresión booleana e `instr1` e `instr2` son instrucciones.

Instrucción WHILE Esta instrucción nos permite repetir una instrucción mientras se cumpla una condición booleana. El `WHILE` procederá a verificar el valor que contiene la condición booleana y ejecutará la instrucción en caso de ser verdadera, luego volverá a verificar la condición y ejecutar condicionalmente, así sucesivamente hasta que la condición contenga un valor falso. La notación para esta instrucción es la siguiente:

```
WHILE bexp DO
  instr;
END
```

Donde `bexp` es una expresión booleana e `instr` es una instrucción o una secuencia de instrucciones.

Instrucción FOR Esta instrucción es un caso especial de `WHILE` donde la condición siempre establece que el identificador pasado como parámetro contenga un valor menor o igual al límite establecido. Si bien para toda instrucción `FOR` existe una de tipo `WHILE` equivalente, resulta útil contar con una iteración más limitada evitando posibles errores de programación. La notación es la siguiente:

```
FOR id TO aexp DO
  instr;
END
```

Donde `id` es un identificador que contiene un valor de tipo `Integer`, `aexp` es una expresión aritmética e `instr` es una instrucción o una secuencia de instrucciones.

Secuencia de instrucciones Como no tiene sentido en un lenguaje imperativo escribir programas de una sola instrucción, es necesario incluir una forma de ejecutar una instrucción luego de la otra. Para ello existe la secuencia de instrucciones, que toma como parámetros de forma inductiva dos instrucciones y devuelve una nueva instrucción. La notación para la secuencia es simplemente separar las instrucciones mediante punto y coma, solo que para unificar la notación utilizamos el punto y coma al final de todas las instrucciones aunque sea una sola o aunque sea la última de la secuencia. Este es otro caso en el que utilizamos la instrucción `skip` ya que cuando hay un punto y coma pero no sigue otra instrucción el lenguaje implícitamente agrega la instrucción `skip`. La notación es entoces:

```
instr1;
instr2;
```

Donde `instr1` e `instr2` son instrucciones.

2.2. Semántica

Para poder dar semántica a nuestro lenguaje tenemos que definir cómo evaluamos las construcciones sintácticas. Al hablar de evaluación queremos decir la forma de ejecutar las instrucciones vistas en la parte anterior y, por consecuencia, también evaluar el resultado de las expresiones. Por ejemplo, supongamos que un programa contiene la siguiente instrucción:

```
int x ::= 3 + y;
```

En un caso así será necesario utilizar la definición de cuatro tipos de evaluación para poder ejecutar la instrucción:

1. Evaluar el literal `3`
2. Evaluar la variable `y`
3. Evaluar el resultado de la suma
4. Evaluar la instrucción de asignación

Para poder realizar estas definiciones utilizaremos los conceptos de *estados* de memoria y *transiciones* entre dichos estados mediante la evaluación de las instrucciones. Pero primero vamos a definir *valores*, que serán el contenido de las posiciones en la memoria.

2.2.1. Valores

Los valores son los que ocupan las posiciones en memoria y, además de guardar el contenido en si mismo, también los marcamos con el tipo correspondiente. Definimos formalmente el tipo `Val` como:

```
Val :=
  bool_val        : Bool    → Val
  int_val         : Integer → Val
  byte_val        : Byte    → Val
  byte_matrix_val : ∀ (m n: ℕ), Matrix Byte m n → Val
```

Fragmento 9: El tipo de los valores.

De esta forma cuando leamos un valor de la memoria obtendremos por ejemplo `int_val 9` indicando el tipo entero y el número correspondiente.

2.2.2. Estado

Un estado representa los valores de todas las variables en un momento dado. Siguiendo la definición de la memoria como la vimos en la sección 2.1.3, si la memoria es un arreglo de posiciones, entonces un estado es como una imagen de dicho arreglo en un determinado momento. Utilizando el estado podemos *leer* la memoria, es decir que dado un identificador, un estado nos devuelve el valor correspondiente para esa variable en un determinado momento de la ejecución. Por lo tanto definimos a cada estado como una función que toma un identificador y devuelve su valor, formalmente:

```
State := Id → option Val
```

Fragmento 10: El tipo del estado.

La función de estado no es total ya que no existe un valor correspondiente a una variable no definida, por eso el tipo de retorno es `option`, que puede ser `None` cuando la variable no está definida o `Some val` donde `val` es el valor correspondiente.

Definimos un estado vacío, en el cual no hay variables definidas, de la siguiente forma:

```
empty_state: State := _ ⇒ None
```

Fragmento 11: Definición del estado vacío.

Este estado nos será de utilidad para poder realizar una definición recursiva de los estados.

Por último tenemos que definir una función que nos permita modificar el valor de una variable, definimos la siguiente función:

```
update(st: State)(id: Id)(val: Val) : State :=
  (id': Id) ⇒ if (equal_id id id') then Some val else st id'
```

Fragmento 12: Definición de una función para actualizar el estado.

La función `update` toma como parámetro un estado y retorna uno nuevo, para el cual se encuentra definido el identificador junto con su valor que también fueron pasados como parámetros. El nuevo estado entonces, toma como parámetro un identificador y se fija si éste es el mismo que fue escrito utilizando `update`, en cuyo caso simplemente debe retornar el valor correspondiente; en caso contrario delega la respuesta al estado original utilizado para la definición —es aquí donde será de utilidad `empty_state` para poder finalizar las llamadas recursivas.

La función `equal_id` simplemente compara identificadores y la mantenemos en una definición abstracta para evitar depender de su implementación.

Las palabras clave `if`, `then` y `else` fueron resaltadas especialmente para evitar confusión ya que esta no es la instrucción de nuestro lenguaje, sino que es el condicional de Coq.

Veamos un ejemplo de un estado que puede llegar a ser construido usando estas primitivas. El siguiente estado contiene dos variables definidas mediante los identificadores `id 1` e `id 2` y sus valores correspondientes son `int_val 2` y `bool_val TRUE` respectivamente.

```
update (update empty_state (id 2) (bool_val TRUE)) (id 1) (int_val 2)
```

Llamémosle `st` a dicho estado, si queremos leer el valor que se encuentra identificado por `id 2` debemos aplicar la función de estado, esto es, `st (id 2)`. Para ver su funcionamiento reducimos la expresión con la definición mostrada en 12:

```
st (id 2)
=
if (equal_id (id 1) (id 2)) then Some (int_val 2)
else (update empty_state (id 2) (bool_val TRUE)) (id 2)
=
if (false) then Some (int_val 2)
else (update empty_state (id 2) (bool_val TRUE)) (id 2)
=
(update empty_state (id 2) (bool_val TRUE)) (id 2)
=
if (equal_id (id 2) (id 2)) then Some (bool_val TRUE)
else emtpy_state (id 2)
=
if (true) then Some (bool_val TRUE)
else emtpy_state (id 2)
=
Some (bool_val TRUE)
```

Fragmento 13: Ejemplo de aplicación de la función de estado.

Notar que si hubiéramos querido obtener el valor de una variable que nunca fue definida, la reducción hubiera continuado hasta llegar a `empty_state`, que no importa el parámetro que tome siempre retorna `None`.

2.2.3. Evaluación

La evaluación de las instrucciones es la que determina realmente la ejecución de un programa, es aquí donde definimos cómo cada instrucción del lenguaje —dependiendo de la evaluación de sus expresiones internas— lleva de un estado a otro.

Veremos primero la evaluación de las expresiones aritméticas, booleanas, bytes y matrices antes de pasar a las instrucciones. Nuestra forma de definir la evaluación de dichas expresiones es mediante una relación entre la expresión, el estado y su valor. Diremos que una expresión evalúa a un cierto valor en un cierto estado si y solamente si la terna (expresión, estado, valor) se encuentra en la relación. Esto nos permite definir nuestra semántica mediante relaciones inductivas. Por ejemplo, siendo $\mathbb{S}$ el conjunto de todos los posibles estados de memoria, las expresiones aritméticas literales se evalúan de la siguiente forma:

$$\forall (z \in \mathbb{Z}, st \in \mathbb{S}),\ st(id)\ =\ Some\ (int\_val\ z)\ \rightarrow (arith\_exp\_id\ z,\ st,\ z) \in ArithEval$$

En las subsecciones a continuación asumiremos la existencia de un estado `st` y utilizaremos la notación `exp ⇓ val` para decir que la expresión `exp` evalúa al valor `val` en el estado `st`.

2.2.4. Evaluación de expresiones

Las siguientes tres, son las definiciones formales de evaluación de literal, identificador y multiplicación aritmética.

```
ArithEval :=
  arith_eval_num   : ∀ (n: Integer),
      arith_exp_num n ⇓ n

  arith_eval_id    : ∀ (i: Id)(z: Integer),
      st i = Some (int_val z) → arith_exp_id i ⇓ z

  arith_eval_mult  : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 * e2 ⇓ int_mult n1 n2
```

Fragmento 14: Relación que define la evaluación de expresiones aritméticas.

Notar que la evaluación de identificadores es la traducción del ejemplo visto en la sección anterior. Desde el punto de vista semántico, la evaluación de un identificador se realiza leyendo dicho valor en la memoria.

Como puede observarse, la evaluación de un literal aritmético vale directamente el valor del número entero y la evaluación de un identificador corresponde al valor que se encuentra en esa posición de memoria —representada por el estado `st`— siempre y cuando dicha variable se encuentre definida en `st`. La multiplicación primero evalúa el resultado de sus dos operandos y luego evalúa la operación de multiplicación en sí misma, aquí es donde utilizamos las operaciones abstractas mencionadas al comienzo de la sección, en este caso `int_mult` cuyos detalles de implementación pueden ser sustituidos sin tener que cambiar nuestra definición de evaluación. Como ya fue dicho, en nuestro trabajo utilizamos las primitivas de Coq, entre las cuales se encuentran los números enteros y su multiplicación.

El resto de las expresiones se comporta de una forma muy similar y se puede leer su definición completa en el anexo A.2, sin embargo mostraremos algunos otros ejemplos que resultan interesantes y pueden ayudar a una mejor comprensión de la semántica. Veamos algunas definiciones de evaluación de expresiones de bytes:

```
ByteEval :=
  byte_eval_lit : ∀ (b:Byte),
      byte_exp_num b ⇓ b

  byte_eval_id : ∀ (i:Id)(b:Byte),
      st i = Some (byte_val b) → byte_exp_id i ⇓ b

  byte_eval_shiftl : ∀ (e: ByteExp)(x: Integer)(n: ArithExp)(b: Byte),
      e ⇓ b → n ⇓ x → e shl n ⇓ byte_shl b x

  byte_eval_matrix :
      ∀ (e1 e2: ArithExp) (m n i j: Integer)
        (mexp: MatrixExp) (matr: Matrix Byte m n),
     mexp ⇓ matr → e1 ⇓ i → e2 ⇓ j → mexp[e1][e2] ⇓ matrix_get i j matr
```

Fragmento 15: Relación que define la evaluación de expresiones de bytes.

En el caso de las expresiones de bytes se aplica el mismo razonamiento para los literales y los identificadores. Por otro lado se puede notar que la operación de shift debe tomar como parámetros un byte y un entero, cada expresión evalúa al correspondiente valor de su tipo y luego la expresión completa evalúa al resultado de la operación abstracta `byte_shl`. El último constructor corresponde al acceso a una posición de una matriz, éste consta de una expresión de matriz junto con dos expresiones aritméticas que representan el índice al cual se está accediendo. Por la forma en que definimos nuestro módulo de matrices, que será explicado en la sección 2.3, en el caso de que la posición no se encuentre definida en la matriz, como por ejemplo un intento de acceso fuera de los límites de su tamaño, la operación `matrix_get` retorna `None`.

2.2.5. Evaluación de instrucciones

En esta sección vamos a definir qué queremos decir con evaluar una instrucción. Ejecutar una instrucción nos lleva de un estado de la memoria a otro, por lo que definimos la evaluación de una instrucción como una relación entre un estado inicial, una instrucción y el estado luego de ejecutar la misma. Podemos plantear un ejemplo similar al de la evaluación de expresiones en cuanto a relación inductiva; la siguiente es la definición de evaluación de asignación de enteros:

$$\forall(zexp \in ArithExp,\ z \in \mathbb{Z}),\ zexp \Downarrow z\ \rightarrow (int\ x\ ::=\ zexp,\ st,\ st[x \leftarrow z]) \in InstrEval$$

Donde se puede notar que al pertenecer la terna (instrucción, estado inicial, estado final) a la evaluación de instrucciones, entonces dicha instrucción al ser ejecutada realiza la transición del estado inicial al final. La notación `st[x ← z]` es equivalente a `update st x z`, función vista en la sección 2.2.2 y quiere decir que es el mismo estado `st` donde escribimos el valor `z` en la variable `x`.

A continuación mostramos algunos ejemplos de la definición formal de la evaluación de instrucciones como tipo inductivo. En los lugares en que utilizamos una evaluación de expresión, por ejemplo `bexp ⇓ TRUE`, estamos referenciando implícitamente al único estado que es nombrado dentro del constructor y que en todos los casos se llama `st`. También utilizamos la notación `instr / st ⇓ st'` donde `instr` es la instrucción y `st` y `st'` son el estado inicial y final respectivamente.

Skip Esta instrucción no tiene ningún efecto sobre el estado, por lo tanto nos lleva de un estado `st` al mismo estado `st`.

```
skip_eval :
  instr_skip / st ⇓ st
```

Asignación Esta instrucción toma como parámetros un identificador y una expresión y nos lleva de un estado `st` a un estado `st'`, donde `st'` es el resultado de en el estado `st` asignarle al identificador el valor de la expresión.

```
assign_int_eval : ∀ (zid: Id) (zexp: ArithExp) (z: Integer),
  zexp ⇓ z → int zid ::= zexp / st ⇓ st[zid ← z]
```

El ejemplo corresponde a la asignación de enteros pero es análoga para booleanos, bytes y matrices. La definición completa se puede ver en el anexo A.2

Instrucción de secuencia Esta instrucción se evalúa a partir de la evaluación de dos instrucciones. Toma como parámetro una instrucción `i1` que nos lleva de un estado `st` a uno `st'`, y una instrucción `i2` que nos lleva de un estado `st'` a uno `st''`. Por lo tanto esta instrucción nos lleva de un estado `st` a uno `st''`.

```
seq_eval : ∀ (i1 i2: Instr) (st' st'': State),
  i1 / st ⇓ st' → i2 / st' ⇓ st'' → i1 ; i2 / st ⇓ st''
```

Instrucción IF Esta instrucción recibe como parámetro una expresión booleana y dos instrucciones `i1` e `i2`. Realizamos la evaluación dependiendo si el resultado de evaluar la expresión booleana es verdadero o falso, esto se puede ver en los dos constructores `if_eval_true` y `if_eval_false`. En el primer caso, el resultado es verdadero y esta instrucción nos lleva del estado `st` al `st'` ejecutando la instrucción `i1`, en el segundo caso el resultado es falso por lo que la instrucción nos lleva del estado `st` a otro estado `st'` ejecutando la instrucción `i2`.

```
if_eval_true : ∀ (bexp: BoolExp) (i1 i2 : Instr) (st': State),
    bexp ⇓ TRUE → i1 / st ⇓ st' →
    IF bexp THEN i1 ELSE i2 END / st ⇓ st'

if_eval_false : ∀ (bexp: BoolExp) (i1 i2: Instr) (st': State),
    bexp ⇓ FALSE → i2 / st ⇓ st' →
    IF bexp THEN i1 ELSE i2 END / st ⇓ st'
```

Instrucción WHILE Esta instrucción recibe como parámetro una expresión booleana y una instrucción. Realizamos la evaluación dependiendo si el resultado de evaluar la expresión booleana es verdadero o falso, por lo cual necesitamos dos constructores los cuales son `while_eval_true` y `while_eval_flase`. Si el resultado es verdadero esta instrucción nos lleva del estado `st` al `st'` ejecutando la instrucción `i`, y en ese estado `st'` nos lleva al estado `st''` volviendo a ejecutar el loop. Si el resultado de evaluar la expresión es falso nos deja en el mismo estado inicial `st`.

```
while_eval_true : ∀ (bexp: BoolExp) (i: Instr) (st' st'': State),
    bexp ⇓ TRUE → i / st ⇓ st' →
    WHILE bexp DO i END / st' ⇓ st'' →
    WHILE bexp DO i END / st ⇓ st''
```

```
while_eval_false : ∀ (bexp: BoolExp) (i: Instr),
    bexp ⇓ FALSE → WHILE bexp DO i END / st ⇓ st
```

Instrucción FOR Esta instrucción recibe como parámetro un identificador con una variable de tipo entero, una expresión aritmética y una instrucción. Realizamos la evaluación dependiendo si el valor de la variable es mayor o igual que el valor de la expresión artimética, separando en dos casos: `for_eval_step` y `for_eval_end`. Si el resultado es mayor o igual, esto es `for_eval_end`, nos quedamos en el mismo estado. Si es menor, pasamos del estado `st` al `st''`, sabiendo que, ejecutando una vez la instrucción —cuerpo del for— pasamos del estado `st` al estado `st'`. Partiendo del estado `st'` y ejecutando el for con la variable de control incrementada en uno pasamos del estado `st` al `st''`.

```
for_eval_end : ∀ (i: Id) (n: Integer) (instr: Instr),
    i ≥ n ⇓ TRUE → FOR i TO n DO instr END / st ⇓ st

for_eval_step : ∀ (i: Id) (x n: Integer)
                  (instr: Instr) (st' st'': State),
    i < n ⇓ TRUE → instr / st ⇓ st' →
    FOR i TO n DO instr END / st' ⇓ st'' →
    st' i = Some (int_val x) →
    st'' i = Some (int _val (x + ONE)) →
    FOR i TO n DO instr END / st ⇓ st''
```

Instrucción SET Matrix Esta instrucción recibe como parámetro un identificador que contiene la matriz a la que se le quiere setear un elemento, dos expresiónes aritmetícas que representan los índices de la matriz, y el byte que se desea setear. En la relación se exige que el resultado de aplicar la función que modifica el valor de la matriz sea válido, es decir que el resultado sea de tipo `Some`, luego la evaluación de la instrucción nos lleva del estado inicial `st` a un estado igual pero donde la variable que contenía la matriz inicial ahora contiene la matriz luego de modificar la posición especificada por los índices, con el byte pasado como parámetro.

```
matrix_set_eval : ∀ (m n i j: Integer) (e1 e2: ArithExp) (e: ByteExp)
                    (mid: Id) (matr matr': Matrix Byte m n) (b: Byte),
    mid ⇓ matr → e1 ⇓ i → e2 ⇓ j → e ⇓ b →
    matrix_set matr i j b = Some matr' →
    mid[e1][e2] ::= e / st ⇓ st[mid ← matr']
```

2.3. Matrices

Para poder implementar los algoritmos AES y Mini-AES necesitamos contar con un tipo de dato para las matrices. Como la librería estándar de Coq no cuenta con este tipo, intentamos utilizar una implementación existente [20] pero su enfoque no resultó útil para este trabajo. Decidimos entonces implementar nuestro propio módulo de matrices. Este módulo nos permite definir matrices de tamaño y tipo paramétricos, y cuenta con los respectivos operadores `get` y `set`. También en este módulo desmostramos ciertas propiedades sobre las matrices que serán utilizadas en la prueba de corrección del Mini-AES.

2.3.1. Representación

Representamos una matriz de tamaño $m \times n$ como un vector de largo m, donde cada posición del mismo es a su vez un vector de largo n correspondiéndose con una fila de la matriz. Implementamos estos vectores en Coq como listas con tipo paramétrico y dependientes en el tamaño, sea `A` un tipo de datos cualquiera, definimos un vector de la siguiente forma:

```
vect A :=
  vnil  : vect A 0
  vcons : ∀ (n: ℕ), A → vect A n → vect A (Suc n)
```

Fragmento 16: Definición inductiva de vectores.

Luego la definición de matriz $m \times n$ es:

```
Matrix A (m n: ℕ) := vect (vect A m) n
```

Fragmento 17: Definición del tipo de las matrices.

2.3.2. Operaciones

En este módulo de matrices implementamos las operaciones `get`, `set` y `new_matr` que se corresponden con las operaciones `matrix_get`, `matrix_set` y `matrix_zero`, las cuales fueron definidas abstractamente en la sintaxis del lenguaje.

La función `get` retorna el valor correspondiente a la posición indicada por los índices `i` y `j`:

```
get (A: Set)(m n i j: ℕ)(matr: Matrix A m n): option A
```

Fragmento 18: Firma de get.

Por otro lado, `set` retorna una nueva mariz que contiene a `elem` en la posición indicada por los índices `i` y `j`:

```
set (A:Set)(m n i j: ℕ)(elem: A)(matr: Matrix A m n): option (Matrix A n m)
```

Fragmento 19: Firma de set.

Tanto `get` como `set` retornan un opcional para indicar el caso de fallo. Para ambas funciones, la única precondición para que retornen un valor de tipo `Some` es:

$$(0 \leq i \leq m-1) \wedge (0 \leq j \leq n-1)$$

Y esto es debido a que los índices varían desde 0 hasta $n-1$ siendo n el largo del vector.

La función `new_matr` genera una nueva matriz de `m` filas por `n` columnas donde todas sus posiciones contienen al valor `elem` :

```
new_matr (A: Set)(elem: A)(m n: ℕ): Matrix A n m
```

Fragmento 20: Firma de new_matr.

2.3.3. Propiedades sobre Matrices

Como vimos en la sección anterior, las operaciones `get` y `set` requieren ciertas precondiciones para no fallar, en cuyo caso retornan `None`. Resulta interesante por otro lado verificar que en el caso de cumplirse las precondiciones entonces el resultado es el esperado.

Para el acceso a una posición, basta con probar que siempre que el tamaño de la matriz sea válido y los índices de acceso se encuentren en un rango también válido, entonces existe un resultado. Enunciamos esto mediante el lema **get_some**:

$$\forall\ (i\ j\ m\ n :\ \mathbb{N})\ (matr :\ Matrix\ A\ m\ n),$$
$$m \geq 1\ \rightarrow\ n \geq 1\ \rightarrow\ 0 \leq i < m\ \rightarrow\ 0 \leq j < n\ \rightarrow\ \exists\ a,\ matr[i][j]\ =\ Some\ a$$

Al modificar una posición, nos interesa comprobar que siempre que modifiquemos una posición válida debe existir una matriz como resultado de la operación. El lema **set_matrix** expresa dicho comportamiento:

$$\forall\ (i\ j\ m\ n :\ \mathbb{N})\ (matr :\ Matrix\ A\ m\ n)\ (b :\ A),$$
$$0 \leq i < m\ \rightarrow\ 0 \leq j < n\ \rightarrow\ \exists\ matr',\ (matr[i][j]\ ::=\ b)\ =\ Some\ matr'$$

Una vez probados estos dos lemas, necesitamos profundizar más y probar que al modificar una posición en una matriz y luego acceder a la misma posición obtenemos el valor modificado. Para ello demostramos el lema **set_matrix_get**:

$$\forall\ (i\ j\ m\ n :\ \mathbb{N})\ (matr\ matr' :\ Matrix\ A\ m\ n)\ (a :\ A),$$
$$m \geq 1\ \rightarrow\ n \geq 1\ \rightarrow\ 0 \leq i < m\ \rightarrow\ 0 \leq j < n\ \rightarrow$$
$$(matr[i][j]\ ::=\ a)\ =\ Some\ matr'\ \rightarrow\ matr'[i][j]\ =\ Some\ a$$

Finalmente nos resultó útil contar con un lema que nos garantice que al modificar una posición en una matriz, las demás posiciones no varían. Para ello demostramos que si modificamos una posición y luego accedemos a otra posición distinta, el valor obtenido es el mismo que se encontraba en la matriz inicial. A este lema lo llamamos **get_set_other_matrix_get**:

$$\forall\ (i\ j\ q\ s\ m\ n :\ \mathbb{N})\ (matr\ matr' :\ Matrix\ A\ m\ n)\ (a\ b :\ A),$$
$$0 \leq i < m\ \rightarrow\ 0 \leq j < n\ \rightarrow$$
$$0 \leq q < m\ \rightarrow\ 0 \leq s < n\ \rightarrow$$
$$\neg(i\ =\ q\ \wedge\ j\ =\ s)\ \rightarrow$$
$$matr[i][j]\ =\ Some\ a\ \rightarrow$$
$$(matr[q][s]\ ::=\ b)\ =\ Some\ matr'\ \rightarrow$$
$$matr'[i][j]\ =\ Some\ a$$

3. AES

En esta sección veremos de forma más detallada el funcionamiento de AES, así como también la versión simplificada del mismo.

3.1. Aritmética de cuerpos finitos

Este algoritmo se basa fundamentalmente en la teoría de campos finitos, que puede ser estudiada en mayor profundidad en [19]. Sin embargo, introducimos aquí brevemente las nociones fundamentales para comprender la operativa de AES.

Un *cuerpo* es un conjunto de elementos junto con la definición dos operaciones llamadas *adición* y *multiplicación*, que cumplen las propiedades asociativa, conmutativa y distributiva de la multiplicación respecto de la adición. A partir de dichas operaciones se debe poder definir un inverso aditivo y otro multiplicativo así como un elemento neutro para la adición y uno para la multiplicación. Estos elementos permiten efectuar las operaciones de sustracción y división. Un *cuerpo finito* es por definición un *cuerpo* definido sobre un conjunto finito de elementos.

Un byte puede ser interpretado como un elemento de un cuerpo finito si usamos una representación polinomial. Sea un byte b definido por sus ocho bits $b_i,\ 0 \leq i \leq 7$, su representación polinomial es la siguiente:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

Por ejemplo el byte 10010111 identifica el elemento $x^7 + x^4 + x^2 + x + 1$ del cuerpo finito.

Definimos entonces las operaciones de adición y multiplicación que, además de ser necesarias para la definición del cuerpo finito, serán aplicadas por las operaciones del algoritmo y resultan de importancia para nuestro trabajo.

Adición La suma de dos elementos en este cuerpo finito se logra sumando cada coeficiente utilizando la suma exclusiva, también conocida como XOR y para la cual utilizaremos la notación $\oplus$. Esta operación es muy económica de implementar en hardware y es muy utilizada en AES, sus cuatro posibles resultados son: $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 1 = 1$ y $0 \oplus 0 = 0$. Para todo par de bytes a y b entonces se define la adición c como $c_i = a_i \oplus b_i,\ 0 \leq i \leq 7$.

Por ejemplo el siguiente es el resultado de una adición en el cuerpo finito:

$$(x^6 + x^4 + x^2 + 1) \oplus (x^7 + x^6 + x^5 + x^4) = x^7 + x^5 + x^2 + 1$$

Que puede ser representado en la notación binaria de bytes como:

$$01010101 \oplus 11110000 = 10100101$$

Multiplicación En esta representación polinómica de bytes, la multiplicación se define igual que la multiplicación de polinomios pero tomando el resultado módulo un polinomio irreducible de grado 8. Un polinomio es irreducible si sus únicos divisores son 1 y él mismo, y en este cuerpo finito que estamos estudiando existen varios que cumplen dicha condición. Su justificación puede encontrarse en [14] y por lo tanto no la veremos aquí, pero para AES fue elegido como polinomio irreducible para utilizar en las operaciones el siguiente: $x^8 + x^4 + x^3 + x + 1$. Utilizaremos la notación $\bullet$ para referirnos a esta operación.

Por ejemplo podemos realizar la siguiente multiplicación:

$$(x^6 + x^4 + x^2 + x + 1) \bullet (x^7 + x + 1) \; = \\ (x^{13} + x^{11} + x^9 + x^8 + x^7) + (x^7 + x^5 + x^3 + x^2 + x) + x^6 + x^4 + x^2 + x + 1 \; = \\ x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

Y al reducirla módulo el polinomio irreducible:

$$(x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) \; mod \; (x^8 + x^4 + x^3 + x + 1) \; = \; x^7 + x^6 + 1$$

Esta reducción nos garantiza que el resultado siempre será de grado menor o igual a 7 y por lo tanto pertenece al cuerpo finito. Existen demostraciones de que efectivamente el cuerpo finito aquí definido cumple las propiedades requeridas, para más detalle ver [19]. Este cuerpo finito tiene un nombre que es $GF(2^8)$ y será el nombre que utilizaremos en adelante.

Rijndael S-box La *S-box* (substitution box) de Rijndael es una matriz cuadrada de tamaño 16 por 16 bytes que será utilizada por el algortimo para proveer no linealidad. Como veremos en algunas operaciones, esta se utiliza como tabla de búsqueda, es decir que será utilizada como una función que dado un byte retorna otro. Utilizaremos en este documento una notación similar a la aplicación de funciones, por ejemplo `S-box(b)` es el byte correspondiente a `b`, aunque la operación real que se está realizando es buscar la posición i, j de la matriz donde i se obtiene de los 4 bits de mayor orden y j de los siguientes 4; por ejemplo para el byte 11001100 el correspondiente es el byte en la posición $12, 12$ de la matriz S-box.

Esta matriz fue diseñada para que el algoritmo sea resistente a criptoanálisis lineal y diferencial, la forma de generarla así como una mejor explicación de sus fundamentos de resistencia al criptoanálisis pueden verse en detalle en [14].

3.2. Rijndael

Como ya fue mencionado previamente, AES trabaja sobre bloques de la entrada, aplicando varias rondas de distintas operaciones sobre cada uno de ellos, por lo cual se caracteriza como una red de sustitución-permutación.

Tamaño de la clave (bits)	Cantidad de rondas	Nk
128	10	4
192	12	6
256	14	8

Cuadro 1: Cantidad de rondas dependiendo del tamaño de la clave.

Como puede apreciarse en la tabla 1, se aplican 10, 12 o 14 rondas para claves de 128, 192 o 256 bits respectivamente; el valor de `Nk` será mejor comprendido al ver la sección 3.2.1. Nos concentraremos en ver las operaciones que se realizan a un único bloque, sin perder generalidad ya que el proceso aplicado a cada uno de ellos es exactamente el mismo. Dicho bloque se representa como una matriz de 4 filas por 4 columnas de bytes y de aquí en adelante nos referiremos a éste como *matriz de estado* o simplemente *estado*. Como veremos más adelante, una de las operaciones a aplicar en cada ronda depende de la clave, pero es requerido que en cada una la clave sea diferente. Por dicho motivo existe un paso previo a la encriptación misma del texto, que es la *expansión de la clave*, donde a partir de la clave inicial, se genera una clave nueva para cada ronda.

Los algoritmos 1 y 2 muestran en alto nivel el orden en que son aplicadas las operaciones sobre el estado. El parámetro `state` representa la matriz de estado y por lo tanto es una matriz de 4 por 4 bytes, mientras que `key` es la clave expandida, que es representada como un arreglo de largo `Nr+1` donde cada entrada del mismo es a su vez una matriz de exactamente el mismo tamaño y tipo que la matriz de estado. La constante `Nr` contiene el número de rondas que deben ser aplicadas y es dependiente del tamaño de la clave.

Algorithm 1 AES encrypt

```
procedure AES_ENCRYPT(state, sbox, key)
    ADD_ROUND_KEY(state, key[0])
    for round = 1 to Nr-1 do
        SUB_BYTES(state, sbox)
        SHIFT_ROWS(state)
        MIX_COLUMNS(state)
        ADD_ROUND_KEY(state, key[round])
    end for
    SUB_BYTES(state, sbox)
    SHIFT_ROWS(state)
    ADD_ROUND_KEY(state, key[Nr])
end procedure
```

Algorithm 2 AES decrypt

```
procedure AES_DECRYPT(state, sboxInv, key)
    ADD_ROUND_KEY(state, key[0])
    for round = 1 to Nr-1 do
        SUB_BYTES(state, sboxInv)
        INV_SHIFT_ROWS(state)
        INV_MIX_COLUMNS(state)
        ADD_ROUND_KEY(state, key[round])
    end for
    SUB_BYTES(state, sboxInv)
    INV_SHIFT_ROWS(state)
    ADD_ROUND_KEY(state, key[Nr])
end procedure
```

En la figura 1 mostramos un diagrama de [28] que muestra claramente la aplicación de cada paso del algoritmo al estado y donde se puede ver cómo los procedimientos de encriptar y desencriptar realizan las operaciones inversas en el orden exactamente inverso.

A continuación mostramos la expansión de la clave así como cada una de las operaciones referenciadas en el pseudocódigo.

3.2.1. Expansión de la clave

En el algoritmo 3 se puede observar la forma de expandir la clave. Este toma como argumento la clave inicial que es una matriz de 4 filas y `Nk` columnas de bytes y escribe en `expanded_key`, la clave expandida, que es una matriz de 4 filas y $4 * (Nr + 1)$ columnas de bytes. Para simplificar la notación, tomamos los accesos a las matrices de forma que retornen la i-ésima columna comenzando en 0; por ejemplo `expanded_key[0]` retorna la primera columna, que son 4 bytes,

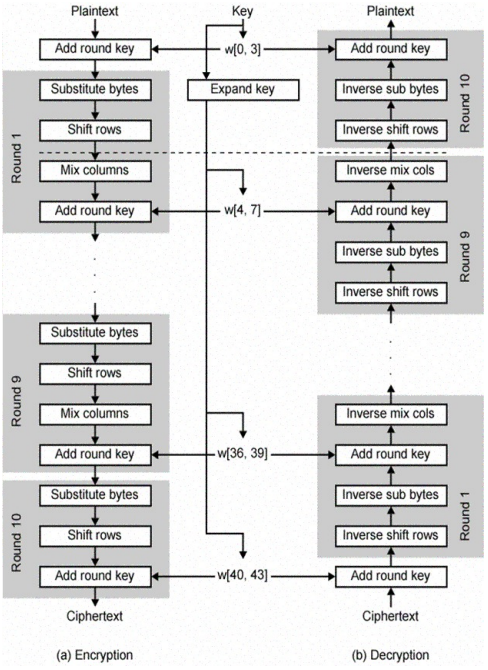

Figura 1: Diagrama de funcionamiento de AES [28].

de `expanded_key`. En el primer `for` simplemente se copia la clave inicial al principio de la expandida y en el segundo se generan las columnas restantes, obteniendo $Nr + 1$ claves de 4 por 4 bytes, que es el tamaño del bloque. Dado que la cantidad de filas es fija y vale 4, se puede calcular el valor de `Nk` en función del tamaño de la clave. Por ejemplo para una clave de 128 bits, que son 16 bytes, debemos dividirlos en 4 filas y el resultado es 4 columnas. El resto de los resultados se encuentra en la tabla 1.

RotWord

Este procedimiento simplemente toma una palabra de 32 bits —que puede ser vista como un arreglo de 4 bytes— y los rota 8 posiciones a la izquierda —o 1 sola si son vistos como bytes. Se puede ver claramente en un ejemplo, si la entrada a `RotWord` es una tira de bits escrita en forma hexadecimal como $2A00F110$ entonces el resultado sería $00F1102A$.

SubWord

Dada una entrada de 4 bytes, `SubWord` reemplaza cada uno de dichos bytes por su correspondiente en la `S-box`. Por lo tanto podemos definir al procedimiento de la siguiente forma:

```
SubWord(in) = [S-box(in[0]), S-box(in[1]), S-box(in[2]), S-box(in[3])]
```

Siendo `in` un arreglo de 4 bytes.

Rcon

Esta operación se puede especificar de la siguiente forma:

$$Rcon(i) = x^{i-1} \text{ mód } x^8 + x^4 + x^3 + x + 1$$

La exponenciación es realizada en el cuerpo finito $GF(2^8)$. En la práctica se necesita como máximo el valor de $Rcon(10)$ para una clave de 128 bits, $Rcon(8)$ y $Rcon(7)$ para 192 y 256

Algorithm 3 AES key expansion

```
procedure KEY_EXPANSION(initial_key, expanded_key)
    for i = 0 to Nk − 1 do
        expanded_key[i] ← initial_key[i]
    end for
    for i ← Nk to 4 ∗ (Nr + 1) − 1 do
        temp ← expanded_key[i − 1]
        if i%4 = 0 then
            ROT_WORD(temp)
            SUB_WORD(temp)
            temp[0] ← temp[0] ⊕ Rcon(i/Nk)
        else if Nk > 6 and i%Nk = 4 then
            SUB_WORD(temp)
        end if
        expanded_key[i] ← expanded_key[i − Nk] ⊕ temp
    end for
end procedure
```

bits respectivamente. Por lo tanto es sencillo de implementar con una tabla de búsqueda con máximo 10 posiciones distintas.

3.2.2. Procedimiento AddRoundKey

En `AddRoundKey` lo que se hace es combinar la clave con el estado. Esto sirve para lograr que la clave afecte la salida del algoritmo y, combinado con el resto de las operaciones, se producirá el efecto deseado de que al cambiar una pequeña parte de la clave esto afecte la mayor parte de la salida. El algoritmo 4 muestra en alto nivel el funcionamiento del procedimiento. Cada

Algorithm 4 AddRoundKey

```
procedure ADD_ROUND_KEY(state, key)
    for i = 0 to 3 do
        for j = 0 to 3 do
            state[i][j] ← state[i][j] ⊕ key[i][j]
        end for
    end for
end procedure
```

posición i, j es un byte y la operación $\oplus$ retorna el `XOR` bit a bit.

A la hora de desencriptar un texto cifrado se utiliza esta misma operación ya que se cumple $\forall a,\ a \oplus a = 0$ y $\forall a,\ a \oplus 0 = a$, y por lo tanto se cumple $\forall a\ b,\ a \oplus b \oplus b = a$, lo cual indica que aplicar AddRoundKey dos veces con la misma clave nos devuelve el texto inicial.

3.2.3. Procedimiento SubBytes

Este procedimiento, que puede ser observado en el algoritmo 5, sustituye cada byte del estado por su correspondiente en la operación de sustitución implementada por la tabla de búsqueda `S-Box` y funciona de la misma forma que el procedimiento `SubWord` de la expansión de la clave, explicado en la sección 3.2.1. El objetivo de esta operación es claramente aportar no linealidad al algoritmo debido a las propiedades ya mencionadas de la tabla de sustitución.

La operación inversa que se realiza para desencriptar es exactamente la misma en cuanto al algoritmo, pero la tabla de búsqueda que se usa contiene los valores que corresponden a la función inversa de `S-box`.

Algorithm 5 SubBytes

```
procedure SUB_BYTES(state, sbox)
    for i = 0 to 3 do
        for j = 0 to 3 do
            state[i][j] ← sbox(state[i][j])
        end for
    end for
end procedure
```

3.2.4. Procedimiento ShiftRows

Si nos fijamos hasta ahora las operaciones que fueron vistas afectan las cuatro columnas de bytes del estado de manera independiente, lo cual degeneraría en cuatro algoritmos que aplican operaciones a las columnas independientemente en lugar de al bloque completo, por lo cual el algoritmo no produciría los efectos de confusión y difusión. Para lograr estos efectos, se transforma la matriz de estado rotando a la izquierda cada fila, tantas veces como el número mismo de la fila. De esta forma un cambio en el texto de entrada o en la clave se distribuye en la salida generando confusión y difusión.

Algorithm 6 ShiftRows

```
procedure SHIFT_ROWS(state)
    for i = 0 to 3 do
        shift(state, i)
    end for
end procedure
procedure SHIFT(state, n)
    for i = 0 to n do
        aux ← state[i][0]
        state[i][0] ← state[i][1]
        state[i][1] ← state[i][2]
        state[i][2] ← state[i][3]
        state[i][3] ← aux
    end for
end procedure
```

Se puede ver en el algoritmo 6 cada fila i es rotada a la izquierda i veces, notar que la primera fila cuyo índice es 0 no es rotada en absoluto. Cabe notar que pueden haber implementaciones más eficientes pero aquí estamos mostrando el funcionamiento conceptual del algoritmo.

La operación inversa a ShiftRows utilizada para desencriptar —denominada INV_SHIFT_ROWS en el algoritmo 2— funciona de la misma forma a excepción de que las rotaciones se realizan a la derecha, por lo cual componer ShiftRows con su inversa trivialmente mantienen a la entrada sin modificar.

3.2.5. Procedimiento MixColumns

Esta operación junto con ShiftRows aporta a generar difusión, ya que al aplicarla los cuatro valores de cada columna de la entrada se combinan para formar cada valor de la salida. Para ello el procedimiento consiste en multiplicar cada columna por una matriz fija que es la siguiente:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Aunque su fundamento se encuentra mejor explicado en [14], podemos mencionar que esta matriz fue elegida especialmente para asegurar la mezcla de los bytes de cada columna y después de pocas rondas al combinarse con ShiftRows, todos los bits de la salida dependerán de todos los bits de la entrada. Además sus coeficientes fueron elegidos teniendo en cuenta su implementación, ya que con estos coeficientes la multiplicación puede implicar como máximo un shift y un XOR de bytes.

Algorithm 7 MixColumns

procedure MIX_COLUMNS(state)
  for $i = 0$ to 3 **do**
    $mix(state[i])$
  end for
end procedure
procedure MIX($state$)
  $state[0] \leftarrow (2 \bullet state[0]) \oplus (3 \bullet state[1]) \oplus state[2] \oplus state[3]$
  $state[1] \leftarrow state[0] \oplus (2 \bullet state[1]) \oplus (3 \bullet state[2]) \oplus state[3]$
  $state[2] \leftarrow state[0] \oplus state[1] \oplus (2 \bullet state[2]) \oplus (3 \bullet state[3])$
  $state[3] \leftarrow (3 \bullet state[0]) \oplus state[1] \oplus state[2] \oplus (2 \bullet state[3])$
end procedure

El algoritmo 7 resume el funcionamiento indicado. Al igual que se ha tratado hasta ahora, la multiplicación y la suma se interpretan en el cuerpo finito $GF(2^8)$ y por eso los símbolos $\bullet$ y $\oplus$.

Para el procedimiento de desencriptar se realiza la misma operación a diferencia de la matriz de multiplicación —$INV\_MIX\_COLUMNS$ en el algoritmo 2, en cuyo caso es la siguiente:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Si realizamos la multiplicación entre ambas matrices obtenemos la siguiente igualdad:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \bullet \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix}$$

Lo cual implica que ambas operaciones son inversas y al componerlas la salida resulta igual a la entrada.

3.3. Mini-AES

3.3.1. Algoritmo

Por ser una simplificación del AES, el algoritmo Mini-AES funciona de la misma forma pero aplicándose en bloques de menor tamaño y utilizando una clave de menor tamaño además de aplicar menor cantidad de rondas. Para el estudio del Mini-AES nos centraremos entonces en sus diferencias con el original.
El tamaño de cada bloque y, por lo tanto, de la matriz de estado en el Mini-AES es de 16 bits. Tanto el estado como la clave consisten en matrices de 2 filas por 2 columnas y, en lugar de contener cada posición un byte, estas contienen unidades de 4 bits denominadas *nibble*s. El número de rondas es exactamente 2 y, al igual que en el AES original, en la última ronda no se aplica la transformación `MixColumns`.

Algorithm 8 Mini-AES encrypt

procedure ENCRYPT($state, sbox, multTable, key$)
    KEY_ADDITION($state, key[2]$)
    NIBBLE_SUB($state, sbox$)
    SHIFT_ROW($state$)
    MIX_COLUMN($state, multTable$)
    KEY_ADDITION($state, key[1]$)
    NIBBLE_SUB($state, sbox$)
    SHIFT_ROW($state$)
    KEY_ADDITION($state, key[0]$)
end procedure

Algorithm 9 Mini-AES decrypt

procedure DECRYPT($state, sboxInv, multTable, key$)
    KEY_ADDITION($state, key[0]$)
    SHIFT_ROW($state$)
    NIBBLE_SUB($state, sboxInv$)
    KEY_ADDITION($state, key[1]$)
    MIX_COLUMN($state, multTable$)
    SHIFT_ROW($state$)
    NIBBLE_SUB($state, sboxInv$)
    KEY_ADDITION($state, key[2]$)
end procedure

En los algoritmos 8 y 9 mostramos el funcionamiento en alto nivel del Mini-AES, allí se puede observar que éste ejecuta solamente 2 rondas de transformaciones. Describimos al mismo aplicando las transformaciones una a una. Como veremos de aquí en adelante, cada transformación se verá simplificada debido al tamaño reducido de la matriz de estado.

3.3.2. Expansión de la clave

Al igual que en el AES, la primera parte de la clave expandida es la misma que la clave de entrada al algoritmo general, luego dos claves más son generadas, una para cada ronda. Se utilizan las transformaciones `NibbleSub` y `Rcon`, la primera es la equivalente a `SubBytes` y será explicada más adelante mientras que la segunda ya fue explicada en la sección 3.2.1 donde se

Algorithm 10 Mini-AES key expansion

procedure MINI_KEY_EXPANSION(init_key, exp_key)
$exp\_key[0][0] \leftarrow init\_key[0][0]$
$exp\_key[0][1] \leftarrow init\_key[0][1]$
$exp\_key[1][0] \leftarrow init\_key[1][0]$
$exp\_key[1][1] \leftarrow init\_key[1][1]$

$exp\_key[0][2] \leftarrow exp\_key[0][0] \oplus$ NIBBLE_SUB(exp_key[1][1]) $\oplus Rcon(1)$
$exp\_key[0][3] \leftarrow exp\_key[0][1] \oplus exp\_key[0][2]$
$exp\_key[1][2] \leftarrow exp\_key[1][0] \oplus exp\_key[0][3]$
$exp\_key[1][3] \leftarrow exp\_key[1][1] \oplus exp\_key[1][2]$

$exp\_key[0][4] \leftarrow exp\_key[0][2] \oplus$ NIBBLE_SUB(exp_key[1][3]) $\oplus Rcon(2)$
$exp\_key[0][5] \leftarrow exp\_key[0][3] \oplus exp\_key[0][4]$
$exp\_key[1][4] \leftarrow exp\_key[1][2] \oplus exp\_key[0][5]$
$exp\_key[1][5] \leftarrow exp\_key[1][3] \oplus exp\_key[1][4]$
end procedure

habla de la expansión de la clave. El algoritmo 10 muestra paso a paso la expansión de cada posición de la clave.

3.3.3. Procedimiento KeyAddition

Siendo análoga a `AddRoundKey`, en esta transformación, como se puede observar en el algoritmo 11, se aplica `XOR` con la clave correspondiente en cada una de las entradas de la matriz de estado. Al igual que en AES, esta operación es inversa de si misma y por lo tanto se usa en

Algorithm 11 KeyAddition

procedure KEY_ADDITION(state, key)
$state[0][0] \leftarrow state[0][0] \oplus key[0][0]$
$state[0][1] \leftarrow state[0][1] \oplus key[0][1]$
$state[1][0] \leftarrow state[1][0] \oplus key[1][0]$
$state[1][1] \leftarrow state[1][1] \oplus key[1][1]$
end procedure

encrypt y decrypt.

3.3.4. Procedimiento NibbleSub

Esta es la transformación equivalente a `SubBytes` y se realiza de la misma forma, sustituyendo en la matriz de estado cada entrada por su correspondiente en la `S-box`, podemos observar la S-box en el cuadro 2 y su inversa en el cuadro 3. La única diferencia aquí es que la tabla tiene un tamaño menor y como podemos observar en el algoritmo 12 escribimos la sustitución de cada entrada de la matriz en una sentencia sin usar bucles.

Para desencriptar, la operación que se utiliza es la misma pero cambiando la `S-box` por otra que es exactamente la inversa.

Algorithm 12 Nibble sub

procedure NIBBLE_SUB(state, sbox)
$state[0][0] \leftarrow sbox(state[0][0])$
$state[0][1] \leftarrow sbox(state[0][1])$
$state[1][0] \leftarrow sbox(state[1][0])$
$state[1][1] \leftarrow sbox(state[1][1])$
end procedure

Input	Output
0000	1110
0001	0100
0010	1101
0011	0001
0100	0010
0101	1111
0110	1011
0111	1000

Input	Output
1000	0011
1001	1010
1010	0110
1011	1100
1100	0101
1101	1001
1110	0000
1111	0001

Cuadro 2: Mini-AES S-Box

3.3.5. Procedimiento ShiftRow

En `ShiftRow` se realiza la rotación de filas al igual que en la original `ShiftRows`. Notar que la diferencia en el nombre es simplemente pasar a singular debido a que, al tener solamente 2 filas la matriz, solamente se termina rotando una fila —recordar que la primera fila, cuyo índice es 0, no es rotada. Podemos implementar esta transformación simplemente como un `swap` entre las dos entradas de la segunda fila de la matriz, como se muestra en el algoritmo 13.

Algorithm 13 Shift row

procedure SHIFT_ROW(state, sbox)
$aux \leftarrow state[1][0]$
$state[1][0] \leftarrow state[1][1]$
$state[1][1] \leftarrow aux$
end procedure

Como finalmente esta operación es un intercambio de posiciones, ella es inversa de sí misma y para desencriptar no es necesario implementar otra función inversa.

3.3.6. Procedimiento MixColumns

En esta transformación se aplica lo mismo que en la original `MixColumns` pero con una matriz más reducida que es la siguiente:

$$\begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$$

Debido a que los posibles valores de entrada no son demasiados y para simplificar la implementación de esta operación, decidimos implementar la función mediante una tabla de búsqueda. Esta tabla abstrae la función $multTable(i, j) = (3 \bullet i) \oplus (2 \bullet j)$. Notar que como la operación $\oplus$ es conmutativa, si damos vuelta los índices tenemos $multTable(j, i) = (2 \bullet i) \oplus (3 \bullet j)$. Entonces `MixColumns` puede ser implementado como en el algoritmo 14.

Input	Output	Input	Output
0000	1110	1000	0111
0001	0011	1001	1101
0010	0100	1010	1001
0011	1000	1011	0110
0100	0001	1100	1011
0101	1100	1101	0010
0110	1010	1110	0000
0111	1111	1111	0101

Cuadro 3: Mini-AES S-Box Inversa

Algorithm 14 Mix columns

procedure MIX_COLUMNS(state, multTable)
$aux00 \leftarrow state[0][0]$
$aux10 \leftarrow state[1][0]$
$state[0][0] \leftarrow multTable(aux00, aux10)$
$state[1][0] \leftarrow multTable(aux10, aux00)$

$aux01 \leftarrow state[0][1]$
$aux11 \leftarrow state[1][1]$
$state[0][1] \leftarrow multTable(aux01, aux11)$
$state[1][1] \leftarrow multTable(aux11, aux01)$
end procedure

Esta tabla en particular cumple una propiedad que más adelante será importante:

$$multTable(x_1,\ y_1) = x_2 \wedge multTable(y_1,\ x_1) = y_2$$
$$\rightarrow$$
$$multTable(x_2,\ y_2) = x_1 \wedge multTable(y_2,\ x_2) = y_1$$

Lo cual quiere decir que la misma tabla se utilizará para la operación `MixColumns` así como para su operación inversa al desencriptar.

3.4. Implementación de Mini-AES

A continuación veremos una a una las traducciones de los pseudocódigos vistos en la sección 3.3 a código Coq utilizando nuestro lenguaje. Podrá apreciarse una transformación bastante directa y sencilla. Por la naturaleza imperativa del lenguaje y por la forma de funcionar del algoritmo que ya fue explicada previamente, todos los procedimientos reciben como entrada un identificador que apunta a la variable donde se encuentra la matriz de estado y modifican la misma en el lugar.

3.4.1. Procedimiento KeyAddition

Este procedimiento se puede escribir de forma casi idéntica al pseudocódigo, utilizamos las notaciones de acceso a una posición de matriz, operación de xor y asignación a posiciones de matriz.

Los parámetros `text` y `key` son identificadores que contienen la matriz de estado y la clave respectivamente.

```
Definition key_addition(text key: Id) :=
  text[0][0] ::= text[0][0] xor key[0][0];
  text[0][1] ::= text[0][1] xor key[0][1];
  text[1][0] ::= text[1][0] xor key[1][0];
  text[1][1] ::= text[1][1] xor key[1][1];
.
```

3.4.2. Procedimiento Nibble sub

Es acá donde se pueden encontrar mayores diferencias con el pseudocódigo. Para poder obtener el valor correspondiente a un byte en la tabla de sustitución `sbox`, necesitamos primero guardar el byte original en una variable. Esto se debe a que la posición de una matriz se indexa utilizando expresiones aritméticas y no bytes. Si quisiéramos acceder directamente, por ejemplo `sbox[0][text[0][0]]` tendríamos un error sintáctico ya que `text[0][0]` es una expresión de byte. Por el contrario, los identificadores sirven tanto como expresiones de bytes así como aritméticas, de esta forma guardamos un valor que es un byte, que luego será leido como número entero para acceder a la matriz.

Los parámetros `text` y `sbox` son identificadores que contienen la matriz de estado y la tabla de sustitución respectivamente.

```
Definition nibble_sub (text sbox: Id) :=
  byte a00   ::= text[0][0];
  byte b00   ::= sbox[0][a00];
  text[0][0] ::= b00;

  byte a01   ::= text[0][1];
  byte b01   ::= sbox[0][a01];
  text[0][1] ::= b01;

  byte a10   ::= text[1][0];
  byte b10   ::= sbox[0][a10];
  text[1][0] ::= b10;

  byte a11   ::= text[1][1];
  byte b11   ::= sbox[0][a11];
  text[1][1] ::= b11;
.
```

3.4.3. Procedimiento Shift row

Este es el procedimiento más sencillo, solo requiere el parámetro `text` que contiene a la matriz de estado. La traducción a nuestro lenguaje es directa.

```
Definition shift_row (text : Id) :=
  byte aux   ::= text[1][0];
  text[1][0] ::= text[1][1];
  text[1][1] ::= aux;
.
```

3.4.4. Procedimiento Mix columns

En `mix_columns` nuevamente tenemos el problema de acceder a una posición de matriz utilizando el resultado de un acceso a matriz previo. Es por ello que también aquí utilizamos variables auxiliares donde almacenamos valores intermedios para poder realizar las sustituciones.

Los parámetros `text` y `mult_table` son identificadores que contienen la matriz de estado y la tabla de multiplicación, respectivamente.

```
Definition mix_columns (text mult_table: Id) :=
  byte a00   ::= text[0][0];
  byte a10   ::= text[1][0];
  text[0][0] ::= mult_table[a00][a10];
  text[1][0] ::= mult_table[a10][a00];

  byte a01   ::= text[0][1];
  byte a11   ::= text[1][1];
  text[0][1] ::= mult_table[a01][a11];
  text[1][1] ::= mult_table[a11][a01];
.
```

3.4.5. Procedimiento Encrypt

Para implementar el procedimiento de encriptación simplemente debemos hacer las llamadas correspondientes a los métodos definidos previamente. Para esto es necesario tener todos los parámetros que son exigidos por dichos procedimientos, estos son la matriz de estado — inicialmente es el texto plano, la tabla de sustitución, la tabla de multiplicación y las tres claves que debieron ser generadas en una etapa de expansión de clave.

```
Definition encrypt (text sbox mult_table key_0 key_1 key_2: Id) :=
  key_addition text key_2;
  nibble_sub   text sbox;
  shift_row    text;
  mix_columns  text mult_table;
  key_addition text key_1;
  nibble_sub   text sbox;
  shift_row    text;
  key_addition text key_0;
.
```

3.4.6. Procedimiento Decrypt

Al igual que al encriptar, al realizar el proceso inverso son necesarios todos los parámetros. Llamamos `cipher_text` al texto cifrado que debió ser el resultado de aplicar `encrypt`, y llamamos `sbox_inv` a la tabla de sustitución inversa a la utilizada para encriptar; `mult_table` recibe el mismo nombre ya que ella es su propia tabla inversa. Las claves mantienen su nombre ya que para que esta operación cumpla la propiedad de ser inversa de la de encriptación, es necesario que sean exactamente las mismas claves.

```
Definition decrypt (cipher_text sbox_inv mult_table
                    key_0 key_1 key_2: Id) :=
  key_addition cipher_text key_0;
```

```
  shift_row    cipher_text;
  nibble_sub   cipher_text sbox_inv;
  key_addition cipher_text key_1;
  mix_columns  cipher_text mult_table;
  shift_row    cipher_text;
  nibble_sub   cipher_text sbox_inv;
  key_addition cipher_text key_2;
.
```

4. Lógica de Hoare

La lógica de Hoare (también conocida como logica de Floyd-Hoare o Reglas de Hoare) es un sistema formal desarrollado por C.A.R. Hoare — y posteriormente refinado por otros investigadores — que proporciona un conjunto de reglas de inferencia para razonar sobre distintas propiedades (especialmente corrección y terminación) de programas imperativos con el rigor de la lógica matemática.

Esta lógica fue publicada por Hoare en 1969 en [17] donde mencionó las contribuciones de Robert Floyd, que había publicado un sistema similar para los diagramas de flujo.

En este trabajo, utilizaremos la lógica de Hoare para probar la corrección del algoritmo Mini-AES. Una forma para poder determinar la corrección de un programa es hacer afirmaciones sobre los valores de determinadas variables en determinado punto de ejecución.

4.1. Afirmaciones

Para poder razonar sobre los programas necesitamos poder realizar afirmaciones sobre propiedades que cumplen los programas en un punto dado de ejecución, mas precisamente necesitamos poder realizar afirmaciones sobre el estado de la memoria en un punto dado del programa. Para esto definimos el siguiente tipo:

$$Assertion \ := \ state \ \rightarrow \ Prop$$

Prop es en Coq el tipo de todas las proposiciones lógicas. Por ejemplo: $True$ y $False$ son proposiciones lógicas; así mismo, dadas dos proposiciones A y B, se puede construir una nueva proposición $A \wedge B$, que semánticamente es la conjunción de A y B. De esta forma y con todos los operadores vistos en la sección 1.5, se definen todas las proposiciones lógicas.

Diremos entonces que una afirmación es una familia de proposiciones lógicas indizadas por estado. Esto puede ser visto más claramente como una función que dado un estado retorna una propiedad sobre el mismo, escrita como proposición lógica. Esto nos resultara sumamente útil para poder definir pre y postcondiciones que se cumplen antes y después de ejecutar una o más instrucciones.

De aquí en adelante utilizaremos el término *afirmación* para referirnos a una *Assertion*.

4.2. Ternas de Hoare

El principal fundamento de esta lógica es la terna o triplete. Una terna describe como la ejecución de un bloque de código cambia el estado del programa. Las ternas tienen la siguiente forma: $\{P\}\ S\ \{Q\}$, donde P y Q son afirmaciones que deben cumplirse para que se cumpla la terna. Es decir, que si el programa S comienza en un estado st en el que se cumple P —lo cual quiere decir que se puede probar $P\ st$, entonces luego de ejecutar el programa, la memoria queda en un estado st' para el cual se cumple Q, y por lo tanto se puede demostrar $Q\ st'$. A P le llamamos precondición y a Q postcondición.

4.3. Reglas

El objetivo de la lógica de Hoare es probar de manera composicional la validez de una terna. De esta forma, la estructura de la prueba debe reflejar la estructura del programa. A continuación vamos a introducir reglas para razonar sobre cada una de las instrucciones del lenguaje: asignación, condicional, loops, etc.

4.3.1. Skip

Si bien esta regla es sencilla, es necesario tenerla ya que contamos con una regla para cada una de las instrucciones del lenguaje y por lo tanto `skip` también cuenta con una. Esta regla nos dice que la instrucción `skip` no cambia el estado del programa. Por lo tanto si una proposición es válida antes de ejecutar la instrucción, también es válida después:

$$\overline{\{Q\}\ skip\ \{Q\}}$$

4.3.2. Asignación

Esta es la regla principal de la lógica de Hoare. Ella nos dice que, para que la propiedad P sea válida en el estado siguiente a la asignación de una expresión a una variable, la propiedad debe ser válida en el estado anterior, pero sustituyendo todas las ocurrencias de la variable por la expresión. Dada una variable X y una expresión exp la regla tiene la siguiente forma:

$$\overline{\{P[X\ \rightarrow\ exp]\}\ X\ ::=\ exp\ \{P\}}$$

Donde $P[X\ \rightarrow\ exp]$ se lee como "P en donde todas las ocurrencias de X se sustituyen por exp". Los siguientes son ejemplos de ternas válidas:

- $\{x + 1 = 2\}\ y := x + 1\ \{y = 2\}$
- $\{x + 1 \leq N\}\ x := x + 1\ \{x \leq N\}$

4.3.3. Asignación de una posición de una matriz

Es de fundamental importancia para este trabajo contar con una regla análoga a la de la asignación vista anteriormente, pero para la asignación de una posición de una matriz.

Supongamos que contamos con un identificador mid cuyo valor en memoria es una matriz $matr$ y sabemos también que $set\ i\ j\ exp\ matr\ =\ matr'$ para cualquier par de enteros i y j y una expresión de byte exp. Expresamos la regla de la siguiente forma:

$$\overline{\{Q[mid\ \rightarrow\ matr']\}\ mid[i][j]\ ::=\ exp\ \{Q\}}$$

Con esta regla, sabemos que Q se cumple luego de ejecutar la instrucción si se cumplía también en el estado anterior pero sustituyendo las ocurrencias de mid por $matr'$.

El siguiente es un ejemplo donde se puede aplicar la regla de la asignación de una posición de la matriz:

$$\{mid = \begin{pmatrix} 1 & m_{01} \\ m_{10} & m_{11} \end{pmatrix}\}\ mid[0][0] := 1\ \{mid = \begin{pmatrix} 1 & m_{01} \\ m_{10} & m_{11} \end{pmatrix}\}$$

4.3.4. Secuencia

Un programa está formado generalmente por una secuencia de instrucciones separadas por punto y coma, que se ejecutan una tras otra. La regla de la secuencia nos permite concatenar dos instrucciones, y obtener las pre y postcondiciones para la concatenación de ambas. Esta regla es fundamental para poder razonar sobre programas que cuentan con más de una instrucción.

Supongamos que tenemos una instrucción `i1` que nos lleva de un estado donde se cumple P a un un estado donde se cumple Q. Supongamos también que tenemos otra instrucción `i2` que

nos lleva de cualquier estado donde se cumple Q a un estado donde se cumple R. Entonces si partimos de cualquier estado donde se cumple P y ejecutamos `i1` seguido de `i2`, llegaremos a un estado donde se cumple R. Formalmente:

$$\frac{\{P\}\ i1\ \{Q\} \qquad \{Q\}\ i2\ \{R\}}{\{P\}\ i1;\ i2\ \{R\}}$$

Por ejemplo, consideremos la siguiente terna que cuenta con dos instrucciones:

- $\{x + 1 = 2\}\ y := x + 1; z := y\ \{z = 2\}$

Podemos aplicar la regla de la secuencia, tomando Q = $\{y = 2\}$; esto nos lleva a tener que probar las siguientes dos ternas:

- $\{x + 1 = 2\}\ y := x + 1\ \{y = 2\}$
- $\{y = 2\}\ z := y\ \{z = 2\}$

4.3.5. Consecuencia

A veces puede pasar que las precondiciones o las postcondiciones que nos quedan en las reglas de Hoare no sean exactamente las que necesitamos, pero sean lógicamente equivalentes. En particular esto se da cuando tenemos una precondición más fuerte o una postcondición más débil. Para resolver este problema, formulamos las siguientes reglas:

$$\frac{\{\ P'\ \}\quad i\quad \{\ Q\ \} \qquad P \rightarrow P'}{\{\ P\ \}\quad i\quad \{\ Q\ \}}\ \text{(regla para la precondición)}$$

$$\frac{\{\ P\ \}\ \ i\ \ \{\ Q'\ \} \qquad Q' \rightarrow Q}{\{\ P\ \}\ \ i\ \ \{\ Q\ \}}\ \text{(regla para la postcondición)}$$

Consideremos la siguiente terna:

$\{0 \leq x < 2\}$ `x := x + 1` $\{0 \leq x \leq 2\}$

Sabiendo que $0 \leq x \rightarrow -1 \leq x$, entonces podemos aplicar la regla para la precondicion, y esto nos lleva a tener que probar la siguiente terna:

$\{-1 \leq x < 2\}$ `x := x + 1` $\{0 \leq x \leq 2\}$ que es lógicamente igual a

$\{0 \leq x+1 \leq 2\}$ `x := x + 1` $\{0 \leq x \leq 2\}$ que sabemos se cumple por la regla de la asignación.

Continuaremos ahora con el ejemplo para la regla de la postcondición. Consideremos la siguiente terna:

$\{0 \leq 2\}$ `x := 0` $\{x = 0\}$

Sabiendo que $x = 0 \rightarrow x \leq 2$, entonces podemos aplicar la regla para la postcondición, y esto nos lleva a tener que probar la siguiente terna:

$\{0 \leq 2\}$ `x := 0` $\{x \leq 2\}$ que sabemos se cumple por la regla de la asignación.

4.3.6. Condicional

Dada una instrucción de la forma `IF b THEN i1 ELSE i2 END`, la regla para el `IF` dice que dada una postcondición Q común a `i1` e `i2` entonces ésta es postcondición de la instrucción `IF`. Para la precondicion debemos considerar el caso en donde la condición del `IF` es verdadera y se ejecuta el cuerpo del `IF`, y el caso donde la condición es falsa y se ejecuta el cuerpo del `ELSE`. Por lo tanto, dada una precondicion P y una expresión booleana `b`, consideramos como precondición del cuerpo del `IF` a la conjunción de P y `b`, y como precondición del cuerpo del `ELSE` a la conjunción de P y la negación de `b`, dando lugar a la siguiente regla:

$$\frac{\begin{array}{c}\{\ P \wedge b\ \}\ i1\ \{\ Q\ \} \\ \{\ P \wedge \neg b\ \}\ i2\ \{\ Q\ \}\end{array}}{\{\ P\ \}\ \text{IF b THEN i1 ELSE i2}\ \{\ Q\ \}}$$

Por ejemplo, supongamos que queremos probar lo siguiente:

$\{0 \leq x \leq 2\}$ `IFB x <2 THEN x := x + 1 ELSE x := 0` $\{0 \leq x \leq 2\}$

Aplicamos la regla condicional y luego debemos probar dos cosas:

$\{0 \leq x \leq 2 \wedge x < 2\}$ `x := x + 1` $\{0 \leq x \leq 2\}$, que simplificado nos queda:

$\{0 \leq x < 2\}$ `x := x + 1` $\{0 \leq x \leq 2\}$ correspondiente al cuerpo del IF-THEN, y

$\{0 \leq x \leq 2 \wedge x \geq 2\}$ `x := 0` $\{0 \leq x \leq 2\}$, que simplificado nos queda:

$\{x = 2\}$ `x := 0` $\{0 \leq x \leq 2\}$ correspondiente al cuerpo del ELSE.

4.3.7. While

Dada una instrucción de la forma `WHILE b DO i END`, donde `b` es una expresión booleana e `i` es una instrucción o secuencia de instrucciones que representa el cuerpo del `WHILE`. Esta regla dice que: dada una condición P; sabiendo que antes del cuerpo del `WHILE` se cumple P y `b` es verdadero, y después del cuerpo del `WHILE` también se cumple P, entonces podemos afirmar que P se mantendrá invariante para cualquier número de repeticiones, por lo tanto sabemos que P es precondición y postcondición de la instrucción `WHILE`, y ademas sabemos que al finalizar el `WHILE` el valor de `b` es falso.

$$\frac{\{\ P \wedge b\ \}\ i\ \{\ P\ \}}{\{\ P\ \}\ \text{WHILE b DO i END}\ \{\ P \wedge \neg b\ \}}$$

Consideremos el siguiente ejemplo:

$\{x \leq 2\}$ `WHILE x <2 DO x := x + 1 END` $\{x = 2\}$

Sabiendo que $\neg x < 2 \wedge x \leq 2 \rightarrow x = 2$, aplicamos la regla para la postcondición, que nos lleva a tener que probar lo siguiente:

$\{x \leq 2\}$ `WHILE x <2 DO x := x + 1 END` $\{\neg x < 2 \wedge x \leq 2\}$

Donde ahora sí podemos aplicar la regla del `WHILE`, que nos lleva a tener que probar la siguiente terna:

$\{x \leq 2 \wedge x < 2\}$ `x := x + 1` $\{x \leq 2\}$, que de forma simplificada nos queda:

$\{x < 2\}$ `x := x + 1` $\{x \leq 2\}$, que es lógicamente equivalente a:

$\{x + 1 \leq 2\}$ `x := x + 1` $\{x \leq 2\}$, que sabemos se cumple por la regla de la asignación.

4.3.8. For

Dada una instrucción de la forma `FOR i TO n DO instr END`, esta regla dice que si se cumple $i < n$ y P es pre y postcondición de `instr`, entonces sabemos que P es pre y postcondición de la instrucción `FOR`, y además sabemos que al finalizar la iteración el valor de i es mayor o igual a n.

$$\frac{\{\ P \wedge i < n\ \}\ \mathit{instr}\ \{\ P\ \}}{\{\ P\ \}\ \text{FOR i TO n DO instr END}\ \{\ P \wedge i \geq n\ \}}$$

Consideremos el siguiente ejemplo:

$\{x \leq 2\}$ `FOR x TO 2 DO y := y + x END` $\{x = 2\}$

Sabiendo que $x = 2 \rightarrow x \geq 2 \wedge x \leq 2$, aplicamos la regla para la postcondición, que nos lleva a tener que probar lo siguiente:

$\{x \leq 2\}$ `FOR x TO 2 DO y := y + x END` $\{\ x \geq 2 \wedge x \leq 2\}$

donde ahora sí podemos aplicar la regla del `FOR`, que nos lleva a tener que probar la siguiente terna:

$\{x \leq 2 \wedge x < 2\}$ `y := y + x` $\{x \leq 2\}$, que de forma simplificada nos queda:

$\{x < 2\}$ `y := y + x` $\{x \leq 2\}$, sabiendo que $x < 2 \rightarrow x \leq 2$, aplicamos la regla para la postcondición, que nos lleva a tener que probar lo siguiente:

$\{x < 2\}$ `y := y + x` $\{x < 2\}$, que sabemos se cumple por la regla de la asignación.

5. Prueba de corrección del algoritmo Mini-AES

En esta sección presentamos la prueba formal de corrección del algoritmo Mini-AES. Probamos que para todo bloque de texto plano (representado por una matriz) y para toda clave, si encriptamos el texto y luego desencriptamos el resultado, el bloque no cambia.

Realizamos la prueba de manera composicional utilizando la lógica de Hoare. Para esto creamos un lema para cada uno de los pasos del algoritmo (KeyAddition, NibbleSub, ShifRow, MixColumns), que luego utilizamos en la prueba principal.

Utilizamos la siguiente notación: escribimos los enunciados de los lemas de corrección con anotaciones indicando las pre y las postcondiciones de cada lema. A las pruebas de los lemas las escribimos con anotaciones luego de cada línea de código, indicando los nuevos valores de las variables que cambiaron de valor. En algunos casos, para que la prueba no sea tan compleja mostramos solo los valores de las variables que cambiaron de valor y omitimos todo lo que permanece invariante; esto es aclarado en cada caso en particular. Donde no tenemos que hacer referencia a ninguna precondición en particular las llamamos `{pre}`. Muchas de estas precondiciones son invariantes que se pasan de un lema a otro para usar en uno en particular. Hablaremos más de esto en la sección correspondiente a las invariantes de este capítulo. A la variable de tipo matriz que representa el bloque de texto la llamamos `mid`. Denotamos con `mid_pre` al valor de dicha variable en las precondiciones de cierto lema. Llamamos `kid` a una variable que representa una clave y `kid0`, `kid1` y `kid2` a cada una de las claves generadas para cada una de las rondas del algoritmo.

A continuación presentamos una descripción de las invariantes. Luego presentaremos primero la prueba final y después los lemas auxiliares que se usan. En todos los casos, para pasar de una línea de código a otra, ya sea en los lemas auxiliares o en la prueba final, se utiliza la regla de Hoare para la secuencia; por lo tanto la omitiremos en las demostraciones.

5.1. Invariantes

La prueba principal `hoare_miniaes` está hecha de manera composicional, dividida en varios lemas: `hoare_key_addition`, `hoare_shift_row`, etc. La forma que tenemos de componer estos lemas es utilizando la regla de Hoare para la secuencia. Esta regla nos dice que las precondiciones de la segunda instrucción deben ser postcondiciones de la primera. Para esto nos definimos un conjunto de invariantes, que son un conjunto de predicados o afirmaciones que son pre y postcondiciones de la prueba principal y de cada uno de los lemas antes mencionados. Cada una de estas invariantes se utiliza en uno o más de los lemas.

- La matriz S-box está bien definida; en cada una de sus 16 posiciones tiene el valor correcto.
- La matriz S-box inversa está bien definida; en cada una de sus 16 posiciones tiene el valor correcto.
- La matriz de multiplicación está bien definida; en cada una de sus 256 posiciones tiene el valor correcto.
- Cada una de las matrices que representan la claves de cada ronda están bien definidas; en cada una de sus posiciones tienen valores entre 0 y 15.
- El bloque de texto está bien definido; en cada una de sus posiciones tiene valores entre 0 y 15.

5.2. Prueba del lema hoare_miniaes

Cuando decimos que vamos a probar la corrección del algoritmo Mini-AES, lo que vamos a probar realmente es que los procedimientos `encrypt` y `decrypt` son inversos, es decir, si aplicamos `encrypt` y luego `decrypt` sobre un bloque cualquiera, este bloque no cambia. Dicho de otra forma: que los valores de la matriz que representa el bloque son los mismos antes y después de aplicar los dos procedimientos. Sabiendo que `mid` es la variable donde se almacena el bloque, nuestra prueba consiste en probar la siguiente terna de Hoare:

```
Theorem hoare_miniaes:
{mid = ( m00  m01 )}
       ( m10  m11 )
encrypt mid;
decrypt mid;
{mid = ( m00  m01 )}
       ( m10  m11 )
```

Hay varios enfoques que se pueden utilizar para demostrar esto. El más directo sería utilizar las reglas de Hoare básicas (asignación, secuencia, etc.) directamente. Es decir, extender la definición de `encrypt` y `decrypt` y luego extender la definición de cada uno de los procedimientos que estos contienen (MixColumns, ShiftRow, KeyAdditon y NibbleSub) para luego aplicar la correspondiente regla de Hoare. Este enfoque nos llevaría a una prueba muy larga y poco estructurada, donde serían repetidas muchas pruebas que en realidad son análogas. Por lo tanto, vemos necesario dividir este problema en subproblemas más pequeños.

Otro enfoque sería partir cada terna en dos y crear lemas para cada una de estas partes y luego probar el lema utilizando estos dos sublemas y la regla de Hoare para la secuencia. Por ejemplo, un lema para `encrypt` y para `decrypt` y así sucesivamente. Este otro enfoque nos llevaría a tener que crear demasiados lemas auxiliares.

Nuestra prueba toma un enfoque intermedio y bastante intuitivo, donde creamos un lema de Hoare para cada uno de los pasos del algoritmo (MixColumns, ShiftRow, KeyAdditon y NibbleSub) y en algunos casos algún otro lema auxiliar. Por lo tanto, lo primero que haremos es extender la definición de `encrypt` y de `decrypt`, es decir sustituirlos por sus correspondientes definiciones, lo que nos deja el teorema de la siguiente forma:

```
Theorem hoare_miniaes:
{mid = ( m00  m01 )}
       ( m10  m11 )
key_addition mid kid2;
nibble_sub mid mini_sbox;
shift_row mid;
mix_columns mid mult_table;
key_addition mid kid1;
nibble_sub mid mini_sbox;
shift_row mid;
key_addition mid kid0;
key_addition mid kid0;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid1;
mix_columns mid mult_table;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid2;
```

$\{mid = \begin{pmatrix} m00 & m01 \\ m10 & m11 \end{pmatrix}\}$

Ahora nuestra terna de Hoare consta de una secuencia de llamadas a procedimientos que, como dijimos anteriormente, la demostramos utilizando la regla de Hoare para la secuencia, además de una regla de Hoare para cada componente del algoritmo. También utilizamos la regla de Hoare para la consecuencia para poder debilitar las precondiciones. Comenzamos la prueba aplicando la regla de la secuencia, que nos lleva a tener que probar las siguientes dos ternas:

$\{mid = \begin{pmatrix} m00 & m01 \\ m10 & m11 \end{pmatrix}\}$
```
key_addition mid kid2;
{Q}
```

```
{Q}
nibble_sub mid mini_sbox;
shift_row mid;
mix_columns mid mult_table;
key_addition mid kid1;
nibble_sub mid mini_sbox;
shift_row mid;
key_addition mid kid0;
key_addition mid kid0;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid1;
mix_columns mid mult_table;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid2;
```
$\{mid = \begin{pmatrix} m00 & m01 \\ m10 & m11 \end{pmatrix}\}$

Ahora, para la primera terna podemos aplicar nuestro lema definido en 5.3, `hoare_key_addition`, que nos instancia la variable `Q` en la segunda terna y nos lleva a tener que demostrar lo siguiente:

$\{mid = \begin{pmatrix} m00 \oplus kid2\_00 & m01 \oplus kid2\_01 \\ m10 \oplus kid2\_10 & m11 \oplus kid2\_11) \end{pmatrix}\}$
```
nibble_sub mid mini_sbox;
shift_row mid;
mix_columns mid mult_table;
key_addition mid kid1;
nibble_sub mid mini_sbox;
shift_row mid;
key_addition mid kid0;
key_addition mid kid0;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid1;
mix_columns mid mult_table;
shift_row mid;
nibble_sub mid mini_sbox_inv;
key_addition mid kid2;
```

$\{mid = \begin{pmatrix} m00 & m01 \\ m10 & m11 \end{pmatrix}\}$

Procedemos de la misma forma con el resto de las instrucciones; aplicamos la regla de la secuencia, la regla de la consecuencia para poder relajar las precondiciones en el caso que corresponda y luego el correspondiente lema para cada procedimiento.

Pero esto no es todo, a medida que vamos aplicando las reglas para cada uno de los pasos, el valor de cada una de las posiciones de la matriz `mid` va cambiando. Antes de empezar, el valor de la posición [0][0] de `mid` es `m00`, luego de aplicar `key_addition mid kid2` el valor de la posición [0][0] es `m00` $\oplus$ `kid2_00`, luego aplicamos `nibble_sub mid miniSboxId` y el valor de [0][0] pasa a ser sbox[0][m00 $\oplus$ kid2_00], y así sucesivamente con cada uno de los pasos del algoritmo y para cada posición de la matriz. Vemos de esta forma que al llegar al final, lo que nos queda en la posición [0][0] de `mid` es equivalente a `m00`, pero no es sintácticamente igual. Para poder reducir esta expresión y terminar la prueba, necesitamos lemas que prueben que la función `xor` es inversa de sí misma, que la S-box y su inversa son efectivamente inversas, y que la tabla de multiplicación es inversa de sí misma. La prueba completa, con mayor, detalle se encuentra en el anexo C.

5.3. Prueba del lema hoare_key_addition

```
Lema hoare_key_addition:

{pre}
key_addition mid kid
{∀ i,j ∈ {0,1} : mid[i][j] = mid_pre[i][j] ⊕ kid[i][j]}
```

Extendemos la definición de `key_addition` y escribimos el código con anotaciones:

```
{pre}
mid[0][0] ::= mid_pre[0][0] xor  kid[0][0]
{mid[0][0] = mid_pre[0][0] ⊕ kid[0][0]}    (1)
mid[0][1] ::= mid_pre[0][1] xor  kid[0][1]
{mid[0][0] = mid_pre[0][0] ⊕ kid[0][0] ∧
 mid[0][1] = mid_pre[0][1] ⊕ kid[0][1]}    (2)
mid[1][0] ::= mid_pre[1][0] xor  kid[1][0]
{mid[0][0] = mid_pre[0][0] ⊕ kid[0][0] ∧
 mid[0][1] = mid_pre[0][1] ⊕ kid[0][1] ∧
 mid[1][0] = mid_pre[1][0] ⊕ kid[1][0]}    (3)
mid[1][1] ::= mid_pre[1][1] xor  kid[1][1]
{mid[0][0] = mid_pre[0][0] ⊕ kid[0][0] ∧
 mid[0][1] = mid_pre[0][1] ⊕ kid[0][1] ∧
 mid[1][0] = mid_pre[1][0] ⊕ kid[1][0] ∧
 mid[1][1] = mid_pre[1][1] ⊕ kid[1][1]}    (4)
```

En los casos (1) (2) (3) y (4) aplicamos la regla de Hoare para la asignación de una posición de la matriz. Vemos que al final se cumple la postcondición del lema.

5.4. Prueba del lema hoare_shift_row

```
Lema hoare_shift_row:
```

```
{pre}
shift_row mid
{mid[1][0] = mid_pre[1][1] ∧ mid[1][1] = mid_pre[1][0]}
```

Extendemos la definición de `shift_row` y escribimos el código con anotaciones:

```
{pre}
byte aux ::= mid_pre[1][0]
{aux = mid_pre[1][0]}           (1)
mid[1][0] ::= mid_pre[1][1]
{aux = mid_pre[1][0] ∧ mid[1][0] = mid_pre[1][1]}       (2)
mid[1][1] ::= aux
{mid[1][0] = mid_pre[1][1] ∧ mid[1][1] = mid_pre[1][0]}         (3)
```

En el paso (1) aplicamos la regla de asignación de Hoare. En los pasos (2) y (3) aplicamos la regla de Hoare para la asignación de una posición de la matriz.

5.5. Prueba del lema hoare_nibble_sub

```
Lema hoare_nibble_sub:

{pre}
nibble_sub mid sbox
{∀ i,j ∈ {0,1} : mid[i][j] = sbox[0][mid_pre[i][j]]}
```

Extendemos la definición de `nibble_sub` y escribimos el código con anotaciones:

```
{pre}
byte a00 ::= mid_pre[0][0]
{a00 = mid_pre[0][0]}                   (1)
byte b00 ::= sbox[0][a00]
{b00 = sbox[0][mid_pre[0][0]]}          (2)
mid[0][0] ::= b00
{mid[0][0] = sbox[0][mid_pre[0][0]]}    (3)

byte a01 ::= mid_pre[0][1]
{a01 = mid_pre[0][1]}                   (4)
byte b01 ::= sbox[0][a01]
{b01 = sbox[0][mid_pre[0][1]]}          (5)
mid[0][1] ::= b01
{mid[0][1] =sbox[0][mid_pre[0][1]]}     (6)

byte a10 ::= mid_pre[1][0]
{a10 = mid_pre[1][0]}                   (7)
byte b10 ::= sbox[0][a10]
{b10 = sbox[0][mid_pre[1][0]]}          (8)
mid[1][0] ::= b10
{mid[1][0] =sbox[0][mid_pre[1][0]]}     (9)

byte a11 ::= mid_pre[1][1]
{a11 = mid_pre[1][1]}                   (10)
byte b11 ::= sbox[0][a11]
```

```
{b11 = sbox[0][mid_pre[1][1]]}          (11)
mid[1][1] ::= b11
{mid[1][1] =sbox[0][mid_pre[1][1]]}     (12)
```

En los pasos (1), (2), (4), (5), (7), (8), (10) y (11) aplicamos la regla de Hoare para la asignación a una variable. En los pasos (3), (6), (9) y (12) aplicamos la regla de Hoare para la asignación de una posición de la matriz. En todos los pasos omitimos anotar las variables y posiciones de la matriz que no cambian de valor, para que la prueba no que la prueba no sea muy compleja.

5.6. Prueba del lema hoare_mix_columns

```
Lema hoare_mix_columns:

{pre}
mix_columns mid sbox
{∀ i,j ∈ {0,1} : mid[i][j] = mult_table[mid_pre[i][j]][mid_pre[¬i][j]]}
```

Extendemos la definición de `mix_columns` y escribimos el código con anotaciones:

```
{pre}
byte a00 ::= mid[0][0]
{a00 = mid_pre[0][0]}                                   (1)
byte a10 ::= mid[1][0]
{a10 = mid_pre[1][0]}                                   (2)

mid[0][0] ::= (mult_table[a00][a10])
{mid[0][0] = mult_table[mid_pre[0][0]][mid_pre[1][0]]}  (3)
mid[1][0] ::= (mult_table[a10][a00])
{mid[1][0] = mult_table[mid_pre[1][0]][mid_pre[0][0]]}  (4)

byte a01 ::= mid[0][1]
{a01 = mid_pre[0][1]}                                   (5)
byte a11 ::= mid[1][1]
{a11 = mid_pre[1][1]}                                   (6)

mid[0][1] ::= (mult_table[a01][a11])
{mid[0][1] = mult_table[mid_pre[0][1]][mid_pre[1][1]]}  (7)
mid[1][1] ::= (mult_table[a11][a01])
{mid[1][1] = mult_table[mid_pre[1][1]][mid_pre[0][1]]}  (8)
```

En los casos (1) (2) (5) y (6) aplicamos la regla de Hoare para la asignación, y en los pasos (3) (4) (7) y (8) aplicamos la regla de Hoare para la asignación de una posición de la matriz. Como las posiciones que no cambian las omitimos en cada paso, vemos que al final se cumple la postcondición del lema.

5.7. Lemas auxiliares

En esta sección describimos algunos de los lemas auxiliares que se usan tanto en la prueba principal así como en los lemas que usa esta prueba.

mini_sbox_result

Este lema nos dice que la S-box está compuesta por nibbles, es decir que en todas sus posiciones contiene bytes entre 0 y 15:

```
Lema mini_sbox_result:

∀ x,y : 0<=x<16 → miniSbox[0][x] = y → 0<=y<16.
```

mini_sbox_inv_result

Este lema nos dice que la S-box inversa está compuesta por nibbles, es decir que en todas sus posiciones contiene bytes entre 0 y 15:

```
Lema mini_sbox_inv_result:

∀ x,y : 0<=x<16 → miniSboxInv[0][x] = y → 0<=y<16.
```

miniSboxInv

Este lema nos dice que la S-box y la S-box inversa son efectivamete inversas:

```
Lema miniSboxInv:

∀ x,y : 0<=x<16 → miniSbox[0][x] = y → miniSboxInv[0][y] = x
```

mult_table_inv

Este lema nos dice que la tabla de multiplicación es inversa de sí misma:

```
Lema mult_table_inv:

∀ x1, x2, x3, x4 : 0<=x1<16 → 0<=x2<16 → mult_table[x1][x2] = x3 →
mult_table[x2][x1] = x4 → (mult_table[x3][x4] = x1 ∧ mult_table[x4][x2] = x2)
```

mult_table_result

Este lema nos dice que la tabla de multiplicación está compuesta por nibbles, es decir que en todas sus posiciones contiene bytes entre 0 y 15:

```
Lema mult_table_result:

∀ x,y : 0<=x<16 → 0<=y<16 → mult_table[x][y] = b → 0<=b<16.
```

nibble_xor

Este lema nos dice que el xor de dos bytes menores que 16 es menor que 16:

```
Lema nibble_xor:

∀ x, y : 0<=x<16 → 0<=y<16 → (x xor y) < 16
```

6. Conclusiones y trabajo a futuro

En este trabajo se presentó el diseño de un lenguaje de programación imperativo, que permite implementar algoritmos criptográficos. El lenguaje cuenta con los tipos primitivos: entero, byte, booleano y matrices de bytes, asi como declaración de variables e instrucciones de asignación, loops y condicionales. Sobre este lenguaje fueron implementados AES y Mini-AES. Por último fue realizada con éxito la prueba de corrección de Mini-AES, utilizando la lógica de Hoare. Esta prueba garantiza que la implementación brindada funciona correctamente: al encriptar un texto plano y desencriptar el resultado se obtiene el texto original.

Este trabajo resulta relevante en cuanto a la criticidad de los temas tratados. El algoritmo AES es sumamente utilizado a nivel internacional y por organizaciones que operan con información sensible de usuarios en todo el mundo. Por ello consideramos importante contar con implementaciones certificadas, es decir, que junto con ellas se presente una prueba formal de su correcto funcionamiento. Se aprovechó el aporte de Mini-AES, que si bien no es utilizado en la práctica, sirve como primer paso hacia la especificación y verificación de AES. Debido a que no fueron encontradas pruebas formales para las implementaciones actuales ni resultados análogos al presente, consideramos a éste como un primer acercamiento al tema.

Se plantea que este trabajo sirva como base para realizar más pruebas, que verifiquen la seguridad de la implementación presentada de Mini-AES. Por ejemplo, se puede probar el correcto manejo de memoria, de forma de no revelar ningún tipo de información sensible [29]. Para esto, es necesario implementar las primitivas de acceso a memoria del lenguaje aquí diseñado en una plataforma —como la considerada en [16], por ejemplo— que permita ejecutar las instrucciones.

También se propone realizar las mismas pruebas sobre la implementación de AES aquí presentada. Éstas pueden resultar más complejas debido a la propia complejidad del algoritmo. AES, a diferencia de Mini-AES, contiene sentencias condicionales y loops, además de utilizar bytes completos en lugar de nibbles. Sin embargo, en este trabajo se brinda un marco de trabajo. Contando con la prueba de Mini-AES aquí desarrollada, se puede seguir un enfoque muy similar de prueba, dividiendo en lemas por operaciones y probando dichos lemas por separado. Además de ello se desarrolló un conjunto de lemas útiles para utilizar la lógica de Hoare, que pueden ser reutilizados. Incluso es posible realizar el mismo conjunto de pruebas sobre algoritmos criptográficos de clave pública, como RSA [27].

Otro posible trabajo a futuro puede tratarse de extender al lenguaje aquí presentado. Por ejemplo agregar tipos primitivos —entre otros, caracteres y cadenas de caracteres— y procedimientos con retorno de valores. Esto permitiría que el lenguaje sea más similar a los lenguajes con los que usualmente se implementan los algoritmos criptográficos, permitiendo generar implementaciones de referencia más realistas y fáciles de traducir a otros lenguajes. Además ello simplificaría la implementación de nuevos algoritmos, que pueden ser más complejos y requerir las herramientas mencionadas.

Referencias

[1] Botan. http://botan.randombit.net/. (Último acceso Noviembre 2014).

[2] Bouncy castle. http://bouncycastle.org/java.html. (Último acceso Noviembre 2014).

[3] Crypto++. http://www.cryptopp.com/. (Último acceso Noviembre 2014).

[4] Código fuente Coq desarrollado en este trabajo. http://www.fing.edu.uy/inco/grupos/gsi/index.php?page=proygrado&locale=es.

[5] Libgcrypt. http://www.gnu.org/software/libgcrypt/. (Último acceso Noviembre 2014).

[6] Openssl. https://www.openssl.org/. (Último acceso Noviembre 2014).

[7] Oracle crypto spec. http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html. (Último acceso Noviembre 2014).

[8] Polarssl. https://polarssl.org/. (Último acceso Noviembre 2014).

[9] The ynot project. http://ynot.cs.harvard.edu/. (Último acceso Noviembre 2014).

[10] Walter Brainerd. *Theory of computation*. Wiley, New York, 1974.

[11] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[12] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[13] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. AES submission, 1999.

[14] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[15] F. Olmedo G. Barthe, B. Grégoire, S. Zanella-Béguelin, and al. Certicrypt: Computer-aided cryptographic proofs in coq. http://certicrypt.gforge.inria.fr/. (Último acceso Noviembre 2014).

[16] C. Luna y J.D. Campo G. Betarte. Virtual cert, descripción del modelo, 2010.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[18] Xavier Leroy and al. Compcert: compilers you can formally trust, 2010.

[19] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, USA, 1986.

[20] N. Magaud. Ring properties for square matrices. http://coq.inria.fr/pylons/pylons/contribs/view/Matrices/trunk. (Último acceso Noviembre 2014).

[21] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

[22] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[23] National Institute of Standards and Technology. Data encryption standard. FIPS Publication 46-2, December 1993.

[24] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001.

[25] National Institute of Standards and Technology. Advanced encryption standard algorithm validation list, Oct. 2014.

[26] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.

[27] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[28] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2011.

[29] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures, 2009.

[30] Raphael Chung wei Phan. Mini advanced encryption standard (mini-aes): A testbed for cryptanalysis. *Students, Cryptologia*, pages 283–306, 2002.

A. Lenguaje completo

En este apéndice se presenta la definición completa del lenguaje diseñado. El código presentado es prácticamente igual al código Coq original, a excepción de algunas palabras claves y símbolos de puntuación. Estos fueron quitados para lograr un código más sencillo de leer.

A.1. Sintaxis

En esta subsección se presentan solamente las definiciones mediante las cuales se pueden construir expresiones e instrucciones válidas del lenguaje.

A.1.1. Expresiones

Utilizando los tipos aquí presentados se construyen las expresiones aritméticas, booleanas, de bytes o de matrices. Se definen de forma recursiva pudiendo ser combinadas para generar expresiones complejas.

```
Id := id :
  nat -> Id

ArithExp :=
  arith_exp_num   : Integer  -> ArithExp
  arith_exp_id    : Id       -> ArithExp
  arith_exp_plus  : ArithExp -> ArithExp -> ArithExp
  arith_exp_minus : ArithExp -> ArithExp -> ArithExp
  arith_exp_mult  : ArithExp -> ArithExp -> ArithExp
  arith_exp_div   : ArithExp -> ArithExp -> ArithExp
  arith_exp_mod   : ArithExp -> ArithExp -> ArithExp

BoolExp :=
  bool_exp_lit : Bool     -> BoolExp
  bool_exp_id  : Id       -> BoolExp
  bool_exp_not : BoolExp  -> BoolExp
  bool_exp_eq  : ArithExp -> ArithExp -> BoolExp
  bool_exp_lt  : ArithExp -> ArithExp -> BoolExp
  bool_exp_le  : ArithExp -> ArithExp -> BoolExp
  bool_exp_gt  : ArithExp -> ArithExp -> BoolExp
  bool_exp_gtb : ByteExp  -> ByteExp  -> BoolExp
  bool_exp_eqb : ByteExp  -> ByteExp  -> BoolExp
  bool_exp_ge  : ArithExp -> ArithExp -> BoolExp
  bool_exp_and : BoolExp  -> BoolExp  -> BoolExp
  bool_exp_or  : BoolExp  -> BoolExp  -> BoolExp

ByteExp :=
  byte_exp_num    : Byte      -> ByteExp
  byte_exp_id     : Id        -> ByteExp
  byte_exp_xor    : ByteExp   -> ByteExp  -> ByteExp
  byte_exp_andbb  : ByteExp   -> ByteExp  -> ByteExp
  byte_exp_shiftl : ByteExp   -> ArithExp -> ByteExp
  byte_exp_shiftr : ByteExp   -> ArithExp -> ByteExp
  byte_exp_matrix : MatrixExp -> ArithExp -> ArithExp -> ByteExp

MatrixExp :=
  matrix_exp_init : ArithExp -> ArithExp -> MatrixExp
```

```
  matrix_exp_lit  : forall A m n, Matrix A m n -> MatrixExp
  matrix_exp_id   : Id -> MatrixExp
```

A.1.2. Instrucciones

Las instrucciones son ejecutadas una a una al ser evaluadas. Estas son las que, al ser evaluadas, realizan cambios de estado.

```
Instr :=
  instr_skip          : Instr
  instr_assign_bool   : Id      → BoolExp   → Instr
  instr_assign_int    : Id      → ArithExp  → Instr
  instr_assign_byte   : Id      → ByteExp   → Instr
  instr_assign_matrix : Id      → MatrixExp → Instr
  instr_seq           : Instr   → Instr     → Instr
  instr_if            : BoolExp → Instr     → Instr    → Instr
  instr_while         : BoolExp → Instr     → Instr
  instr_for           : Id      → ArithExp  → Instr    → Instr
  instr_matrix_set    : Id      → ArithExp  → ArithExp → ByteExp → Instr
```

Notación

Esta notación puede ser utilizada al escribir programas en el lenguaje presentado. En la columna *Constructor* se muestra la notación utilizando directamente el constructor de Coq, en *Símbolo* se muestra su contraparte con notación simplificada. Notar que en muchos casos la notación es infija, es decir que el símbolo se coloca entre los operandos.

Constructor	Símbolo
arith_exp_num a	a
arith_exp_id a	a
arith_exp_plus a b	a + b
arith_exp_minus a b	a - b
arith_exp_mult a b	a * b
arith_exp_div a b	a / b
arith_exp_mod a b	a % b

Cuadro 4: Notación para expresiones aritméticas.

A.2. Semántica

A continuación se presenta la definición formal completa de la evaluación del lenguaje. Al evaluar las expresiones e instrucciones del lenguaje estamos dándole semántica, esto es, estamos definiendo el resultado de ejecutar el programa.

A.2.1. Expresiones

La evaluación de una expresión siempre se puede ver como un resultado de una operación, que puede implicar leer un valor de la memoria, pero que no realiza cambios en la misma.

Constructor	Símbolo
bool_exp_lit a	a
bool_exp_id a	a
bool_exp_eq a b	a `==` b
bool_exp_lt a b	a `lt` b
bool_exp_le a b	a `le` b
bool_exp_gt a b	a `gt` b
bool_exp_ge a b	a `ge` b
bool_exp_not a	`!a`
bool_exp_and a b	a `&&` b
bool_exp_or a b	a `||` b

Cuadro 5: Notación para expresiones booleanas.

Constructor	Símbolo
byte_exp_num a	a
byte_exp_id a	a
byte_exp_xor a b	a `xor` b
byte_exp_andbb a b	a `and` b
byte_exp_shiftl a b	a `shl` b
byte_exp_shiftr a b	a `shr` b
byte_exp_matrix matr i j	`matr[i][j]`

Cuadro 6: Notación para expresiones de bytes.

```
ArithEval :=
  arith_eval_num   : ∀ (n: Integer),
      arith_exp_num n ⇓ n

  arith_eval_id    : ∀ (i: Id)(z: Integer),
      st i = Some (int_val z) → arith_exp_id i ⇓ z

  arith_eval_plus  : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 + e2 ⇓ int_plus n1 n2

  arith_eval_minus : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 - e2 ⇓ int_minus n1 n2

  arith_eval_mult  : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 * e2 ⇓ int_mult n1 n2

  arith_eval_div   : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 / e2 ⇓ int_div n1 n2

  arith_eval_mod   : ∀ (e1 e2: ArithExp) (n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 % e2 ⇓ int_modulo n1 n2
```

Fragmento 21: Relación que define la evaluación de expresiones aritméticas.

```
BoolEval :=
  bool_eval_lit : ∀ (b: Bool),
```

```
      bool_exp_lit b ⇓ b

  bool_eval_id  : ∀ (i:Id)(b:Bool),
       st i = Some (bool_val b) → (bool_exp_id i) ⇓ b

  bool_eval_eq  : ∀ (e1 e2: ArithExp)(n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 == e2 ⇓ int_eq n1 n2

 bool_eval_eqb  : ∀ (e1 e2: ByteExp)(n1 n2: Byte),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 == e2 ⇓ byte_eq n1 n2

  bool_eval_lt  : ∀ (e1 e2: ArithExp)(n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 < e2 ⇓ int_lt n1 n2

  bool_eval_le  : ∀ (e1 e2: ArithExp)(n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 ≤ e2 ⇓ int_le n1 n2

  bool_eval_gt  : ∀ (e1 e2: ArithExp)(n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 > e2 ⇓ int_gt n1 n2

 bool_eval_gtb  : ∀ (e1 e2: ByteExp)(n1 n2: Byte),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 > e2 ⇓ byte_gt n1 n2

  bool_eval_ge  : ∀ (e1 e2: ArithExp)(n1 n2: Integer),
      e1 ⇓ n1 → e2 ⇓ n2 → e1 ≥ e2 ⇓ int_ge n1 n2

  bool_eval_not : ∀ (e1: BoolExp)(b1: Bool),
      e1 ⇓ b1 → bool_exp_not e1 ⇓ ¬b1

  bool_eval_and : ∀ (e1 e2: BoolExp)(b1 b2: Bool),
      e1 ⇓ b1 → e2 ⇓ b2 → e1 ∧ e2 ⇓ bool_and b1 b2

  bool_eval_or  : ∀ (e1 e2: BoolExp)(b1 b2: Bool),
      e1 ⇓ b1 → e2 ⇓ b2 → e1 ∨ e2 ⇓ bool_or b1 b2
```

Fragmento 22: Relación que define la evaluación de expresiones booleanas.

```
ByteEval :=
  byte_eval_lit : ∀ (b:Byte),
      byte_exp_num b ⇓ b

  byte_eval_id : ∀ (i:Id)(b:Byte),
      st i = Some (byte_val b) → byte_exp_id i ⇓ b

  byte_eval_xor : ∀ (e1 e2: ByteExp)(b1 b2: Byte),
      e1 ⇓ b1 → e2 ⇓ b2 → e1 xor e2 ⇓ byte_xor b1 b2

  byte_eval_andbb : ∀ (e1 e2: ByteExp)(b1 b2: Byte),
      e1 ⇓ b1 → e2 ⇓ b2 → e1 and e2 ⇓ byte_and b1 b2

  byte_eval_shiftl : ∀ (e: ByteExp)(x: Integer)(n: ArithExp)(b: Byte),
      e ⇓ b → n ⇓ x → e shl n ⇓ byte_shl b x

  byte_eval_shiftr : ∀ (e: ByteExp)(x: Integer)(n: ArithExp)(b: Byte),
      e ⇓ b → n ⇓ x → e shr n ⇓ byte_shr b x
```

```
  byte_eval_matrix :
      ∀ (e1 e2: ArithExp) (m n i j: Integer)
        (mexp: MatrixExp) (matr: Matrix Byte m n),
      mexp ⇓ matr → e1 ⇓ i → e2 ⇓ j → mexp[e1][e2] ⇓ matrix_get i j matr
```

Fragmento 23: Relación que define la evaluación de expresiones de bytes.

```
MatrixEval :=
  matrix_eval_lit : ∀ (matr: Matrix Byte m n),
      matrix_exp_lit matr ⇓ matr

  matrix_eval_id : ∀ (i: Id)(matr: Matrix Byte m n),
      st i = Some (byte_matrix_val mx) → matrix_exp_id i ⇓ matr

  matrix_eval_init : ∀ (e1 e2: ArithExp),
      e1 ⇓ m → e2 ⇓ n → matrix_exp_init e1 e2 ⇓ matrix_zero m n
```

Fragmento 24: Relación que define la evaluación de expresiones de matrices.

A.2.2. Instrucciones

La evaluación de instrucciones, a diferencia de la de expresiones, puede realizar cambios en los valores en memoria. Al cambiar el valor de una variable en memoria decimos que se realiza una transición de un estado a otro. Por esto, la evaluación de instrucciones se define como una relación entre estados.

```
InstrEval :=
  skip_eval :
    instr_skip / st ⇓ st

  assign_bool_eval : ∀ (bid: Id) (bexp: BoolExp) (b: Bool),
    bexp ⇓ b → bool bid ::= bexp / st ⇓ st[bid ← b]

  assign_int_eval : ∀ (zid: Id) (zexp: ArithExp) (z: Integer),
    zexp ⇓ z → int zid ::= zexp / st ⇓ st[zid ← z]

  assign_byte_eval : ∀ (bid: Id) (bexp: ByteExp) (b: Byte),
    bexp ⇓ (Some b) → byte bid ::= bexp / st ⇓ st[bid ← b]

  assign_matrix_eval : ∀ (mid: Id) (m n: Integer)
                       (matr: Matrix Byte m n) (mexp : MatrixExp),
    mexp ⇓ matr → matrix mid ::= mexp / st ⇓ st[mid ← matr]

  seq_eval : ∀ (i1 i2: Instr) (st' st'': State),
      i1 / st ⇓ st' → i2 / st' ⇓ st'' → i1 ; i2 / st ⇓ st''

  if_eval_true : ∀ (bexp: BoolExp) (i1 i2 : Instr) (st': State),
      bexp ⇓ TRUE → i1 / st ⇓ st' →
      IF bexp THEN i1 ELSE i2 END / st ⇓ st'

  if_eval_false : ∀ (bexp: BoolExp) (i1 i2: Instr) (st': State),
      bexp ⇓ FALSE → i2 / st ⇓ st' →
```

```
      IF bexp THEN i1 ELSE i2 END / st ⇓ st'

  while_eval_true : ∀ (bexp: BoolExp) (i: Instr) (st' st'': State),
      bexp ⇓ TRUE → i / st ⇓ st' →
      WHILE bexp DO i END / st' ⇓ st'' →
      WHILE bexp DO i END / st ⇓ st''

  while_eval_false : ∀ (bexp: BoolExp) (i: Instr),
      bexp ⇓ FALSE → WHILE bexp DO i END / st ⇓ st

  for_eval_end : ∀ (i: Id) (n: Integer) (instr: Instr),
      i ≥ n ⇓ TRUE → FOR i TO n DO instr END / st ⇓ st

  for_eval_step : ∀ (i: Id) (x n: Integer)
                    (instr: Instr) (st' st'': State),
      i < n ⇓ TRUE → instr / st ⇓ st' →
      FOR i TO n DO instr END / st' ⇓ st'' →
      st' i = Some (int_val x) →
      st'' i = Some (int _val (x + ONE)) →
      FOR i TO n DO instr END / st ⇓ st''

  matrix_set_eval : ∀ (m n i j: Integer) (e1 e2: ArithExp) (e: ByteExp)
                      (mid: Id) (matr matr': Matrix Byte m n) (b: Byte),
      mid ⇓ matr → e1 ⇓ i → e2 ⇓ j → e ⇓ b →
      matrix_set matr i j b = Some matr' →
      mid[e1][e2] ::= e / st ⇓ st[mid ← matr']
```

Fragmento 25: Definición inductiva de la evaluación de instrucciones.

B. Implementación de AES

Aquí se presenta la implementación completa del estándar AES, implementada sobre el lenguaje anteriormente presentado. Se divide este anexo en una subsección por cada operación que incluye el algoritmo.

B.1. Key expansion

Sub word

```
Definition sub_word(w sbox: Id) :=
  int i ::= 1;
  FOR i TO 4 DO
    byte b ::= w[i][1];
    byte row ::= b band 240;(*11110000*)
    byte row ::= row shr 4;
    byte col ::= b band 15;(*00001111*)
    byte aux ::= sbox[row][col];
    w[i][1] ::= aux;
  END
.
```

Rot word

```
Definition rot_word(w: Id) :=
  int k ::= 1;
  byte aux ::= w[1][1];
  FOR k TO (Nb -1) DO
    byte sig ::= w[k + 1][1];
    w[k][1] ::= sig;
  END;
  w[4][1] ::= aux;
.
```

Key expansion

```
Definition key_expansion(key: Id):=
  int i ::= 1;
  WHILE (i le Nk) DO
    (* w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]) *)
    byte aux ::= key[1][i];
    w[1][i] ::= aux;

    byte aux ::= key[2][i];
    w[2][i] ::= aux;

    byte aux ::= key[3][i];
    w[3][i] ::= aux;

    byte aux ::= key[4][i];
    w[4][i] ::= aux;

    int i ::= i + 1;
  END;

  int i ::= Nk;
  int auxint ::= Nk * (Nr + 1);

  WHILE (i le auxint) DO
    byte aux ::= w[1][i - 1];
    temp[1][1] ::= aux;
    byte aux ::= w[2][i - 1];
    temp[1][2] ::= aux;
    byte aux ::= w[3][i - 1];
    temp[1][3] ::= aux;
    byte aux ::= w[4][i - 1];
    temp[1][4] ::= aux;

    IFB (i mod Nk == 1) THEN
      rot_word temp;
      sub_word temp sbox;

      byte temp ::= temp xor rcon[1][i / Nk];
    ELSE
      sub_word temp sbox;
```

```
    END;

    (* w[i] = w[i-Nk] xor temp *)
    byte aux ::= w[1][i - Nk] xor temp;
    w[i - 1][1] ::= aux;
    byte aux ::= w[2][i - Nk] xor temp;
    w[i - 1][2] ::= aux;
    byte aux ::= w[3][i - Nk] xor temp;
    w[i - 1][3] ::= aux;
    byte aux ::= w[4][i - Nk];
    w[i - 1][4] ::= aux;

    int i ::= i + 1;
  END
.
```

B.2. AES cipher

Sub bytes

```
Definition sub_bytes(matr: Id)(sbox: Id):=
  int i ::= 1;
  int j ::= 1;

  FOR i TO Nb DO
    FOR j TO 4 DO
      byte b ::= matr[i][j];
      byte row ::= b band 240;(*11110000*)
      byte row ::= row shr 4;
      byte col ::= b band 15;(*00001111*)
      byte aux ::= sbox[row][col];
      matr[i][j] ::= aux;
    END
  END
.
```

AddRoundKey

```
Definition add_round_key(matr: Id)(key: Id)(offset: ArithExp) :=
  int i ::= offset + 1;
  int j ::= 1;

  FOR i TO Nb DO
    FOR j TO 4 DO
      byte x ::= matr[i][j];
      byte y ::= key[i][j];
      matr[i][j] ::= x xor y;
    END
  END
.
```

Galois multiplication 2

```
(* Galois mult 2 *)
Definition gmul2(b: Id) :=
  byte b ::= b shl 1;
  IFB (b gt 255) THEN
    byte b ::= b xor 27;
  END
.
```

Galois multiplication 3

```
(* Galois mult 3 *)
Definition gmul3(b: Id) :=
  byte baux ::= b;
  byte b ::= b shl 1;

  IFB (b gt 255) THEN
    byte b ::= b xor 27;
  END;
  byte b ::= b xor baux;
.
```

Mix columns

```
Definition mix_columns(matr: Id) :=
  int i ::= 1;

  FOR i TO Nb DO
    byte a0 ::= matr[i][1];
    byte a1 ::= matr[i][2];
    byte a2 ::= matr[i][3];
    byte a3 ::= matr[i][4];

    byte gmul2a0 ::= a0;
    byte gmul3a0 ::= a0;
    byte gmul2a1 ::= a1;
    byte gmul3a1 ::= a1;
    byte gmul2a2 ::= a2;
    byte gmul3a2 ::= a2;
    byte gmul2a3 ::= a3;
    byte gmul3a3 ::= a3;

    gmul2 gmul2a0;
    gmul3 gmul3a0;
    gmul2 gmul2a1;
    gmul3 gmul3a1;
    gmul2 gmul2a2;
    gmul3 gmul3a2;
    gmul2 gmul2a3;
```

```
    gmul3 gmul3a3;

    matr[i][1] ::= gmul2a0 xor gmul3a1 xor a2 xor a3;
    matr[i][2] ::= a0 xor gmul2a1 xor gmul3a2 xor a3;
    matr[i][3] ::= a0 xor a1 xor gmul2a2 xor gmul3a3;
    matr[i][4] ::= gmul3a0 xor a1 xor a2 xor gmul2a3;
  END
.
```

Shift rows

```
Definition shift_rows(matr: Id) :=
  int i ::= 1;
  int j ::= 1;

  FOR i TO Nb DO
    FOR j TO i DO
      byte aux ::= matr[1][j];

      FOR k TO 3 DO
        byte sig ::= matr[i + 1][k];
        matr[i][k] ::= sig;
      END;

      matr[i][j] ::= aux;
    END
  END
.
```

AES cipher

```
Definition aes_cipher (state: Id) :=
  add_round_key state w 1;
  int round ::= 1;
  FOR round TO Nr DO
    sub_bytes state sbox;
    shift_rows state;
    mix_columns state;
    add_round_key state w (round * Nb);
  END;

  sub_bytes state sbox;
  shift_rows state;
  add_round_key state w (round * Nb);
.
```

C. Prueba de Mini-AES

Se presenta a continuación una prueba completa de la relación inversa que existe entre los procedimientos de encriptar y desencriptar. Esta prueba se aleja de la prueba realizada en Coq en

el trabajo, ya que no se encuentra escrita en ningún lenguaje formal. Sin embargo, la estructura de la prueba y la lógica utilizada es análoga a la realizada en este trabajo. El código fuente de la prueba formal, verificada en Coq, se encuentra en las referencias de la sección 1.6.

Aclaración: nos resulta útil saber el valor de la matriz de estado en un estado del programa; por lo tanto denotamos como `mid_i` el valor de dicha variable en el estado `i` del programa.

```
Lema hoare_miniaes:
{pre}
encrypt mid
decrypt mid
{∀ i,j : mid[i][j] = mid_pre[i][j]}
```

Para la prueba de corrección, extendemos la definición de encrypt y de decrypt y nos queda lo siguiente:

```
{pre}
key_addition mid kid2
{mid_1[0][0] = mid_0[0][0] ⊕ kid2[0][0] ∧
 mid_1[0][1] = mid_0[0][1] ⊕ kid2[0][1] ∧
 mid_1[1][0] = mid_0[1][0] ⊕ kid2[1][0] ∧
 mid_1[1][1] = mid_0[1][1] ⊕ kid2[1][1]} (s1)
nibble_sub mid miniSboxId
{mid_2[0][0] = sbox[0][mid_1[0][0]] ∧
 mid_2[0][1] = sbox[0][mid_1[0][1]] ∧
 mid_2[1][0] = sbox[0][mid_1[1][0]] ∧
 mid_2[1][1] = sbox[0][mid_1[1][1]]} (s2)
shift_row mid
{mid_3[1][0] = mid_2[1][1] ∧
 mid_3[1][1] = mid_2[1][0]} (s3)
mix_columns mid mult_table
{mid_4[0][0] = mult_table[mid_3[0][0]][mid_3[1][0]] ∧
 mid_4[0][1] = mult_table[mid_3[0][1]][mid_3[1][1]] ∧
 mid_4[1][0] = mult_table[mid_3[1][0]][mid_3[0][0]] ∧
 mid_4[1][1] = mult_table[mid_3[1][1]][mid_3[0][1]]} (s4)

key_addition mid kid1
{mid_5[0][0] = mid_4[0][0] ⊕ kid1[0][0] ∧
 mid_5[0][1] = mid_4[0][1] ⊕ kid1[0][1] ∧
 mid_5[1][0] = mid_4[1][0] ⊕ kid1[1][0] ∧
 mid_5[1][1] = mid_4[1][1] ⊕ kid1[1][1]} (s5)
shift_row mid
{mid_6[1][0] = mid_5[1][1] ∧
 mid_6[1][1] = mid_5[1][0]} (s6)
nibble_sub mid miniSboxId
{mid_7[0][0] = sbox[0][mid_6[0][0]] ∧
 mid_7[0][1] = sbox[0][mid_6[0][1]] ∧
 mid_7[1][0] = sbox[0][mid_6[1][0]] ∧
 mid_7[1][1] = sbox[0][mid_6[1][1]]} (s7)

key_addition mid kid0
{mid_8[0][0] = mid_7[0][0] ⊕ kid0[0][0] ∧
 mid_8[0][1] = mid_7[0][1] ⊕ kid0[0][1] ∧
 mid_8[1][0] = mid_7[1][0] ⊕ kid0[1][0] ∧
 mid_8[1][1] = mid_7[1][1] ⊕ kid0[1][1]} (s8)
```

```
key_addition mid kid0
{mid_9[0][0] = mid_8[0][0] ⊕ kid0[0][0] ∧
 mid_9[0][1] = mid_8[0][1] ⊕ kid0[0][1] ∧
 mid_9[1][0] = mid_8[1][0] ⊕ kid0[1][0] ∧
 mid_9[1][1] = mid_8[1][1] ⊕ kid0[1][1]} (s9)

nibble_sub mid sboxInv
{mid_10[0][0] = sboxInv[0][mid_9[0][0]] ∧
 mid_10[0][1] = sboxInv[0][mid_9[0][1]] ∧
 mid_10[1][0] = sboxInv[0][mid_9[1][0]] ∧
 mid_10[1][1] = sboxInv[0][mid_9[1][1]]} (s10)

shift_row mid
{mid_11[1][0] = mid_10[1][1] ∧
 mid_11[1][1] = mid_10[1][0]} (s11)

key_addition mid kid1
{mid_12[0][0] = mid_11[0][0] ⊕ kid1[0][0] ∧
 mid_12[0][1] = mid_11[0][1] ⊕ kid1[0][1] ∧
 mid_12[1][0] = mid_11[1][0] ⊕ kid1[1][0] ∧
 mid_12[1][1] = mid_11[1][1] ⊕ kid1[1][1]} (s12)

mix_columns mid mult_table
{mid_13[0][0] = mult_table[mid_12[0][0]][mid_12[1][0]] ∧
 mid_13[0][1] = mult_table[mid_12[1][0]][mid_12[0][0]] ∧
 mid_13[1][0] = mult_table[mid_12[0][1]][mid_12[1][1]] ∧
 mid_13[1][1] = mult_table[mid_12[1][1]][mid_12[0][1]]} (s13)

shift_row mid
{mid_14[1][0] = mid_13[1][1] ∧
 mid_14[1][1] = mid_13[1][0]} (s14)

nibble_sub mid sboxInv
{mid_15[0][0] = sboxInv[0][mid_14[0][0]] ∧
 mid_15[0][1] = sboxInv[0][mid_14[0][1]] ∧
 mid_15[1][0] = sboxInv[0][mid_14[1][0]] ∧
 mid_15[1][1] = sboxInv[0][mid_14[1][1]]} (s15)
(s1) Esto se cumple por el lema hoare_key_addition mid\_0 kid2 \\
(s2) lema hoare_nibble_sub\\
(s3) hoare_shift_row\\
(s4) hoare_mix_columns
key_addition mid kid2
{mid_16[0][0] = mid_15[0][0] ⊕ kid2[0][0] ∧
 mid_16[0][1] = mid_15[0][1] ⊕ kid2[0][1] ∧
 mid_16[1][0] = mid_15[1][0] ⊕ kid2[1][0] ∧
 mid_16[1][1] = mid_15[1][1] ⊕ kid2[1][1]} (s16)
```

En cada uno de los pasos s1..s16 aplicamos el correspondiente lema, de los lemas anteriormente vistos: key_addition, nibble_sub, shift_row y mix_columns.

Ahora tenemos que probar por que $mid_{16}[i][j] = mid_0[i][j] \; \forall \; i, j \in \{0, 1\}$

Comenzaremos por sustituir (s8) en (s9) y nos queda lo siguiente:

$$\begin{aligned} mid_9[0][0] &= mid_7[0][0] \oplus kid0[0][0] \oplus kid0[0][0] \\ &= mid_7[0][0] \\ mid_9[0][1] &= mid_7[0][1] \oplus kid0[0][1] \oplus kid0[0][1] \\ &= mid_7[0][1] \\ mid_9[1][0] &= mid_7[1][0] \oplus kid0[1][0] \oplus kid0[1][0] \\ &= mid_7[1][0] \\ mid_9[1][1] &= mid_7[1][1] \oplus kid0[1][1] \oplus kid0[1][1] \\ &= mid_7[0][0] \end{aligned} \tag{2}$$

En (s10) sustituyendo con lo obtenido en (2) tenemos:

$$\begin{aligned} mid_10[0][0] &= sboxInv[mid_7[0][0]] \\ mid_10[0][1] &= sboxInv[mid_7[0][1]] \\ mid_10[1][0] &= sboxInv[mid_7[1][0]] \\ mid_10[1][1] &= sboxInv[mid_7[1][1]] \end{aligned} \tag{3}$$

Sustituyendo en (3) con las equaciones de (s7) nos queda

$$\begin{aligned} mid_10[0][0] &= sboxInv[sbox[mid_6[0][0]]] \\ mid_10[0][1] &= sboxInv[sbox[mid_6[0][1]]] \\ mid_10[1][0] &= sboxInv[sbox[mid_6[1][0]]] \\ mid_10[1][1] &= sboxInv[sbox[mid_6[1][1]]] \end{aligned}$$

Sabiendo que sbox y sboxInv son inversas llegamos a que:

$$\begin{aligned} mid_{10}[0][0] &= mid_6[0][0] \\ mid_{10}[0][1] &= mid_6[0][1] \\ mid_{10}[1][0] &= mid_6[1][0] \\ mid_{10}[1][1] &= mid_6[1][1] \end{aligned} \tag{4}$$

En (s11) y utilizando 4 tenemos:

$$\begin{aligned} mid_{11}[0][0] &= mid_{10}[0][0] = mid_6[0][0] \\ mid_{11}[0][1] &= mid_{10}[0][1] = mid_6[0][1] \\ mid_{11}[1][0] &= mid_{10}[1][1] = mid_6[1][1] \\ mid_{11}[1][1] &= mid_{10}[1][0] = mid_6[1][0] \end{aligned} \tag{5}$$

En (s6):

$$\begin{aligned} mid_6[0][0] &= mid_5[0][0] \\ mid_6[0][1] &= mid_5[0][1] \\ mid_6[1][0] &= mid_5[1][1] \\ mid_6[1][1] &= mid_5[1][0] \end{aligned} \tag{6}$$

Sustituimos 6 en 5 y nos queda:

$$\begin{aligned} mid_{11}[0][0] &= mid_5[0][0] \\ mid_{11}[0][1] &= mid_5[0][1] \\ mid_{11}[1][0] &= mid_5[1][0] \\ mid_{11}[1][1] &= mid_5[1][1] \end{aligned} \tag{7}$$

Por (s12) sabemos:

$$\begin{aligned} mid_{12}[0][0] &= mid_{11}[0][0] \oplus kid1[0][0] \\ mid_{12}[0][1] &= mid_{11}[0][1] \oplus kid1[0][1] \\ mid_{12}[1][0] &= mid_{11}[1][0] \oplus kid1[1][0] \\ mid_{12}[1][1] &= mid_{11}[1][1] \oplus kid1[1][1] \end{aligned} \tag{8}$$

Sustituimos (7) en (8) y obtenemos:

$$\begin{aligned} mid_{12}[0][0] &= mid_5[0][0] \oplus kid1[0][0] \\ mid_{12}[0][1] &= mid_5[0][1] \oplus kid1[0][1] \\ mid_{12}[1][0] &= mid_5[1][0] \oplus kid1[1][0] \\ mid_{12}[1][1] &= mid_5[1][1] \oplus kid1[1][1] \end{aligned} \tag{9}$$

Luego sustituimos (s5) en (9):

$$\begin{aligned} mid_{12}[0][0] &= mid_4[0][0] \oplus kid1[0][0] \oplus kid1[0][0] \\ &= mid_4[0][0] \\ mid_{12}[0][1] &= mid_4[0][1] \oplus kid1[0][1] \oplus kid1[0][1] \\ &= mid_4[0][1] \\ mid_{12}[1][0] &= mid_4[1][0] \oplus kid1[1][0] \oplus kid1[1][0] \\ &= mid_4[1][0] \\ mid_{12}[1][1] &= mid_4[1][1] \oplus kid1[1][1] \oplus kid1[1][1] \\ &= mid_4[1][1] \end{aligned} \tag{10}$$

En (s13) y utilizando (10) tenemos:

$$\begin{aligned} mid_{13}[0][0] &= mult\_table[mid_4[0][0]][mid_4[1][0]] \\ mid_{13}[0][1] &= mult\_table[mid_4[0][1]][mid_4[1][1]] \\ mid_{13}[1][0] &= mult\_table[mid_4[1][0]][mid_4[0][0]] \\ mid_{13}[1][1] &= mult\_table[mid_4[1][1]][mid_4[0][1]] \end{aligned} \tag{11}$$

Por (s4) sabemos que:

$$\begin{aligned} mid_4[0][0] &= mult\_table[mid_3[0][0]][mid_3[1][0]] \\ mid_4[0][1] &= mult\_table[mid_3[0][1]][mid_3[1][1]] \\ mid_4[1][0] &= mult\_table[mid_3[1][0]][mid_3[0][0]] \\ mid_4[1][1] &= mult\_table[mid_3[1][1]][mid_3[0][1]] \end{aligned} \tag{12}$$

Sustituimos (12) en (11) y nos queda lo siguiente:

$$\begin{aligned} mid_{13}[0][0] &= mult\_table[mult\_table[mid_3[0][0]][mid_3[1][0]]][mult\_table[mid_3[1][0]][mid_3[0][0]]] \\ &= mid_3[0][0] \\ mid_{13}[0][1] &= mult\_table[mult\_table[mid_3[0][1]][mid_3[1][1]]][mult\_table[mid_3[1][1]][mid_3[0][1]]] \\ &= mid_3[0][1] \\ mid_{13}[1][0] &= mult\_table[mult\_table[mid_3[1][0]][mid_3[0][0]]][mult\_table[mid_3[0][0]][mid_3[1][0]]] \\ &= mid_3[1][0] \\ mid_{13}[1][1] &= mult\_table[mult\_table[mid_3[1][1]][mid_3[0][1]]][mult\_table[mid_3[0][1]][mid_3[1][1]]] \\ &= mid_3[1][1] \end{aligned} \tag{13}$$

Las segundas igualdades se derivan del lema mult_table_inv.

En (s14) y utilizando 13 tenemos:

$$\begin{aligned} mid_{14}[0][0] &= mid_{13}[0][0] = mid_3[0][0] \\ mid_{14}[0][1] &= mid_{13}[0][1] = mid_3[0][1] \\ mid_{14}[1][0] &= mid_{13}[1][1] = mid_3[1][1] \\ mid_{14}[1][1] &= mid_{13}[1][0] = mid_3[1][0] \end{aligned} \tag{14}$$

En (s3):

$$\begin{aligned} mid_3[0][0] &= mid_2[0][0] \\ mid_3[0][1] &= mid_2[0][1] \\ mid_3[1][0] &= mid_2[1][1] \\ mid_3[1][1] &= mid_2[1][0] \end{aligned} \tag{15}$$

Sustituimos 15 en 14 y nos queda:

$$\begin{aligned} mid_{14}[0][0] &= mid_2[0][0] \\ mid_{14}[0][1] &= mid_2[0][1] \\ mid_{14}[1][0] &= mid_2[1][0] \\ mid_{14}[1][1] &= mid_2[1][1] \end{aligned} \tag{16}$$

En (s15) sustituyendo con lo obtenido en (16) tenemos:

$$\begin{aligned} mid_15[0][0] &= sboxInv[mid_2[0][0]] \\ mid_15[0][1] &= sboxInv[mid_2[0][1]] \\ mid_15[1][0] &= sboxInv[mid_2[1][0]] \\ mid_15[1][1] &= sboxInv[mid_2[1][1]] \end{aligned} \tag{17}$$

Sustituyendo en (17) con las equaciones de (s2) nos queda

$$\begin{aligned} mid_15[0][0] &= sboxInv[sbox[mid_1[0][0]]] \\ &= mid_1[0][0] \\ mid_15[0][1] &= sboxInv[sbox[mid_1[0][1]]] \\ &= mid_1[0][1] \\ mid_15[1][0] &= sboxInv[sbox[mid_1[1][0]]] \\ &= mid_1[1][0] \\ mid_15[1][1] &= sboxInv[sbox[mid_1[1][1]]] \\ &= mid_1[1][1] \end{aligned} \tag{18}$$

Ahora sustituyendo (18) en (s16):

$$\begin{aligned} mid_{16}[0][0] &= mid_1[0][0] \oplus kid2[0][0] \\ mid_{16}[0][1] &= mid_1[0][1] \oplus kid2[0][1] \\ mid_{16}[1][0] &= mid_1[1][0] \oplus kid2[1][0] \\ mid_{16}[1][1] &= mid_1[1][1] \oplus kid2[1][1] \end{aligned} \tag{19}$$

Finalmente sustituimos (s1) en (19) y llegamos a:

$$
\begin{aligned}
mid_{16}[0][0] &= mid_0[0][0] \oplus kid2[0][0] \oplus kid2[0][0] \\
&= mid_0[0][0] \\
mid_{16}[0][1] &= mid_0[0][1] \oplus kid2[0][1] \oplus kid2[0][1] \\
&= mid_0[0][1] \\
mid_{16}[1][0] &= mid_0[1][0] \oplus kid2[1][0] \oplus kid2[1][0] \\
&= mid_0[1][0] \\
mid_{16}[1][1] &= mid_0[1][1] \oplus kid2[1][1] \oplus kid2[1][1] \\
&= mid_0[1][1]
\end{aligned}
$$