TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Desarrollo de librerías de firma ciega para OpenSSL

TITULACIÓN: Grado en Ingeniería Telemática

AUTOR: Fernando Román García

DIRECTOR: Juan Hernández Serrano

FECHA: 8 de febrero de 2016

Título: Desarrollo de librerías de firma ciega para OpenSSL

Autor: Fernando Román García

Director: Juan Hernández Serrano

Data: 8 de febrero del 2016

Resumen

OpenSSL es una suite criptográfica inicialmente dedicada a proveer una implementación de los protocolos Transport Layer Security (TLS) y Secure Sockets Layer (SSL), pero que actualmente se ha convertido en una suite criptográfica de carácter generalista. OpenSSL está pensado para poder ser utilizado como librería, su código es abierto y está desarrollado por voluntarios. A día de hoy, incorpora mucho de los mecanismos de seguridad "clásica", muy relacionados con conceptos como la autenticación y la confidencialidad. Sin embargo, éstos no son suficientes para afrontar otros retos de seguridad más complejos, sobre todo los relacionados con la privacidad y el anonimato, que requieren de nuevos mecanismos.

El objetivo principal de este TFG es el de implementar en OpenSSL uno de estos mecanismos: la firma ciega. Este tipo concreto de firma permite que el firmante no conozca el contenido del mensaje que firma, ofreciendo privacidad al propietario. La realización del proyecto para OpenSSL es una motivación extra, ya que se puede considerar uno de los programas más importantes de seguridad a nivel mundial, en el que participan reconocidos expertos.

Como objetivo secundario, se plantea la implementación de un sistema de moneda electrónica que utilice la librería OpenSSL de firma ciega y que sirva como prueba de funcionamiento.

El último objetivo del proyecto es el de ampliar conocimientos, en la medida de lo posible, sobre temas relacionados con seguridad y programación.

Como resultado del trabajo realizado, hemos obtenido una librería plenamente funcional para OpenSSL que permite realizar una firma válida sin necesidad de saber el contenido de lo que se firma. Además, se ha conseguido la correcta implementación de un sistema de moneda electrónica que aplica de forma satisfactoria la librería de firma ciega implementa.

Title: Implementation of blind-signature libraries for OpenSSL

Author: Fernando Román García

Director: Juan Hernández Serrano

Date: Febrary 8 2016

Overview

OpenSSL is a cryptographic suite that was initially aimed at providing an implementation of the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols; although has now become a cryptographic suite with a much broader scope. OpenSSL is intended to be used as a library, it's open source and it's developed by a worldwide community of volunteers. Today, OpenSSL incorporates many of the “classic” security mechanisms, mainly related to provide authentication and confidentiality. However, these mechanisms are not sufficient to deal with other more-complex security challenges, especially those related to privacy and anonymity, which require new specific mechanisms.

The main objective of this TFG is the implementation of one of these mechanisms, the blind signature, for OpenSSL. This particular type of signature allows the signer to sign a message without actually knowing its contents. The fact of developing this TFG within the OpenSSL umbrella is an added motivation to the authors since OpenSSL is probably the most important and widely-use cryptographic suite.

Another objective of this TFG is the implementation of an electronic currency system that uses the proposed blind-signature library for OpenSSL. This system acts as a proof of proper performance of this library.

As a result of the work done, we have got a fully functional library that allows the generation of a valid signature without knowing the contents of the message to be signed. Furthermore, we have deployed an electronic currency system that makes proper use of the provided OpenSSL blind-signature libraries.

ÍNDICE

INTRODUCCIÓN

OpenSSL es un proyecto de código abierto que en un principio implementaba todas las funcionalidades para los protocolos *Transport Layer Security* (TLS) y *Secure Sockets Layer* (SSL). Con el tiempo se ha convertido en la suite criptográfica más utilizada y ha evolucionado hacia una librería de propósito general que incluye multitud de algoritmos criptográficos y formatos de datos de seguridad.

El uso de las librerías de OpenSSL permite integrar la seguridad “clásica” (autenticación, cifrado) de forma sencilla y eficiente en el desarrollo de aplicaciones y/o servicios. Sin embargo, OpenSSL todavía necesita integrar mecanismos “avanzados” de seguridad que a día de hoy son necesarios para garantizar no sólo la seguridad “clásica” sino también la privacidad y el anonimato. Entre estos mecanismos avanzados se encuentra la firma ciega que, como veremos, es esencial para garantizar anonimato en servicios como, por ejemplo, votaciones electrónicas o moneda electrónica.

La motivación principal de este proyecto es implementar un librería de firma ciega para OpenSSL plenamente funcional. Además, como prueba de concepto, se pretende desarrollar un sistema de moneda electrónica que utilice dichas librerías. A nivel personal, también es una gran motivación el hecho de poder ahondar en el funcionamiento de una suite que puede considerarse como una de las biblias de la seguridad y en la que participan de forma abierta multitud de expertos de seguridad.

Para empezar a poner en contexto del proyecto, el primer capítulo explica como ha evolucionado la historia de la criptografía y más en detalle aquello que es necesario para la implementación que he realizado.

Una vez ya tengamos los conocimientos necesarios de criptografía podremos, en el segundo capítulo, explicar la implementación de herramientas y librerías que permiten integrar firma ciega en OpenSSL.

En el tercer y cuarto capítulo veremos, respectivamente, la descripción e implementación de un sistema de moneda electrónica. Se trata de una prueba de concepto en la que el sistema desarrollado hace uso de las librerías de firma ciega implementadas en el capítulo anterior.

Por último, en el quinto capítulo presentaremos las conclusiones y líneas futuras para este trabajo de fin de grado.

CAPÍTULO 1. Antecedentes

Contiene todo lo necesario para poder entender cómo se ha implementado la firma ciega en OpenSSL y el sistema de moneda electrónica diseñado.

1.1. Fundamentos criptográficos

En este apartado explicaremos las bases de la criptografía. Para ello primero veremos qué es la seguridad de la información y en concreto qué problema hay cuando se quiere ofrecer confidencialidad y anonimato. A continuación, veremos cómo ha evolucionado la criptografía hacia los métodos de criptografía simétrica y asimétrica. Finalmente, explicaremos en que consisten las funciones hash criptográficamente seguras.

1.1.1. Seguridad de la información

Según Wikipedia [1], “La seguridad de la información es el conjunto de medidas preventivas y reactivas de las organizaciones y de los sistemas tecnológicos que permiten resguardar y proteger la información buscando mantener la confidencialidad, la disponibilidad e integridad de la misma.”

Concretamente, cuando se habla de seguridad se pretende ofrecer a los usuarios confidencialidad, integridad, disponibilidad y autenticación.

- **Confidencialidad.** Consiste en garantizar que sólo las personas autorizadas son las que pueden acceder al contenido del mensaje.
- **Integridad.** Impide que personas no autorizadas modifiquen el contenido los datos.
- **Disponibilidad.** Garantiza que la información esté accesible por las personas autorizadas en el momento que deseen acceder.
- **Autenticación.** Consiste en identificar a la persona que ha generado la información.

Existe un problema en cuanto a la seguridad y es que si se quiere garantizar que una persona tenga acceso a una determinada información es necesario que ésta esté antes autenticada y por lo tanto pierde el anonimato. Esto complica los sistemas en los que se quiere ofrecer confidencialidad y anonimato como, por ejemplo, el boto electrónico y la moneda electrónica.

1.1.2. Criptografía simétrica/asimétrica

A lo largo de la historia se han desarrollado muchos algoritmos criptográficos. En un inicio se ponía toda la seguridad en el secreto del algoritmo, es decir,

que lo más importante era que el atacante no conociera como estaban cifrados los mensajes.

Esto conlleva un gran problema de escalabilidad, ya que si toda la seguridad está en el algoritmo, cada comunicación tendría que mantener en secreto su propio algoritmo para que fuera seguro. El problema es que, si fuera descubierto, habría que modificar el algoritmo o implementar uno nuevo. Además, no se podrían hacer estándares porque deberían mantenerse en secreto.

Por lo tanto, un buen sistema criptográfico pone toda su seguridad en la clave y ninguna en el algoritmo. Esto quiere decir que para poder cifrar mensajes por otra persona hay que adivinar su clave. Por este motivo, uno de los factores más importantes esta en la cantidad de posibles claves que tiene un algoritmo. Otro factor a tener en cuenta es que cuando se usa el algoritmo para cifrar no debe reducir el abanico de las posibles claves, ya sea porque dependiendo de la clave, tarde un tiempo diferente en ser ejecutado o porque el criptograma dé información acerca de la clave.

Según el funcionamiento de la clave existen tres métodos criptográficos de criptografía, la simétrica, la asimétrica.

1.1.2.1. Criptografía simétrica

El primer método que se ideó fue la criptografía simétrica. Consiste en utilizar la misma clave para cifrar y descifrar los mensajes y por ello es conocida también como criptografía de una clave.

También se le conoce como criptografía de clave secreta ya que las dos partes de la comunicación han de ponerse de acuerdo de antemano en que clave desean utilizar y solo la pueden conocer ellos.

Para ofrecer confidencialidad, una vez que ambas partes comparten la clave, cuando una quiere enviar un mensaje a la otra, lo cifra utilizando la clave secreta, lo envía al destinatario y este ultimo utiliza la misma clave para descifrar el contenido del mensaje.

Si Alice quiere garantizar la autenticidad de los mensajes enviados a Bob en criptografía simétrica, tendrá que calcular un resumen del mensaje y cifrarlo con la clave secreta que solo conocen Alice y Bob. El resumen cifrado se conoce como MAC (Message Autentication Code). Cuando Bob reciba el mensaje, descifrará el MAC y lo comparará con el resumen del mensaje. Si son iguales, quiere decir que solo Alice le ha podido enviar este mensaje. El problema de este sistema es que no protege contra no repudio, es decir que Bob no puede demostrar que Alice es la que le ha enviado el mensaje porque el podría haberlo generado también.

El principal problema de estos sistemas está en el intercambio de claves, que requiere de un intercambio previo por un canal seguro.

1.1.2.2. Criptografía asimétrica y/o de clave pública

El segundo método de criptografía que surgió fue la criptografía asimétrica. Este método, a diferencia del basado en clave simétrica, utiliza una clave para cifrar y otra para descifrar y por ello también se le conoce como criptografía de dos claves.

La criptografía de clave pública es un tipo de criptografía asimétrica en que las dos claves pertenecen a la misma persona pero una de ellas se publica (es pública) y la otra se mantiene en secreto (es privada).

Si el mensaje se cifra utilizando la clave pública, la única persona que puede descifrar el mensaje es la dueña de la clave privada de dicha clave pública con lo que se otorgaría confidencialidad al mensaje (sólo el receptor puede descrifrar el mensaje).

Si por otro lado el remitente cifra utilizando la clave privada, cualquier persona podrá descifrar el mensaje utilizando su clave pública pero solo el poseedor del par de claves podría haber cifrado ese mensaje. De esta manera se otorga autenticidad. Simplificando y de forma generalista, a este método se conoce como firma digital; si bien la firma generalmente se hace sobre un resumen del mensaje y no el mensaje directamente.

Sus inconvenientes son que son más lentos, las claves deben ser más grandes que las utilizadas en criptografía simétrica y que el mensaje cifrado ocupa más espacio que el original. Para solventar el problema de la velocidad de cálculo, primero se suele utilizar criptografía asimétrica para obtener una clave secreta común y poder utilizarla como clave simétrica para comunicaciones confidenciales.

1.1.3. Funciones hash

Una función de hash es una función h que recibe como entrada un valor M con una longitud determinada (en número de cifras, tamaño en bits,…) y lo transforma en otro $H=h(M)$ de menor tamaño. Por este motivo también se las conoce como funciones resumen.

Estas funciones son utilizadas en muchos ámbitos de la computación, como por ejemplo, para generar índices para algoritmos de búsqueda. En nuestro caso nos interesan las funciones de hash criptográficamente seguras que son aquellas que cumplen las siguientes propiedades:

- **Unidireccionalidad.** Conocido un resumen $h(M)$, debe ser computacionalmente imposible encontrar M a partir de dicho resumen.
- **Compresión**: A partir de un mensaje de cualquier longitud, el resumen $h(M)$ debe tener una longitud fija. Lo normal es que la longitud de $h(M)$ sea menor que la de M.
- **Facilidad de cálculo.** Debe ser fácil calcular $h(M)$.

- **Difusión.** El resumen $h(M)$ debe ser una función compleja de todos los bits del mensaje M. Si se modifica un bit del mensaje M, el hash $h(M)$ debería cambiar aproximadamente la mitad de sus bits.
- **Colisión simple.** Conocido M, será computacionalmente imposible encontrar otro M' tal que $h(M)=h(M')$. Se conoce como resistencia débil a las colisiones.
- **Colisión fuerte.** Será computacionalmente difícil encontrar un par (M,M') de forma que $h(M)=h(M')$. Se conoce como resistencia fuerte a las colisiones.

Como ejemplos de funciones de hash criptográficamente seguras tenemos SHA256 o MD5.

1.2. Formatos de datos

En este apartado explicaremos los formatos que hemos utilizado a lo largo del proyecto. Primero de todo, hablaremos de ASN.1 y su codificación en los formatos BER, DER y PEM. A continuación, comentaremos el otro formato que hemos utilizado: JSON.

1.2.1. ASN.1

"El formato ASN.1 (Abstract Syntax Notation One) es una norma para representar datos independientemente de la máquina que se esté usando y sus formas de representación internas. Es un protocolo de nivel de presentación en el modelo OSI." [2]

Por ejemplo, imaginemos que queremos transmitir la información del estado de las diferentes partes de un coche a un ordenador. Estos dispositivos son totalmente diferentes y para tener un formato estándar sobre como ha de viajar la información podríamos utilizar ASN.1 para definir nuestras estructuras de datos.

Por lo tanto, ASN.1 provee de la representación lógica, pero no de la física que tendrán los datos, es decir, define una notación abstracta de la información pero no restringe cómo esta información está codifica.

1.2.1.1. Valores y tipos

Un tipo es una clase de dato. Los tipos de datos que define se pueden clasificar en tipos primarios, tipos construidos y tipos definidos. Por otro lado, un valor cuantifica el tipo.

Los tipos primarios o simples son los que almacenan un único valor. Los más importantes son:

- **INTEGER.** Se usa para representar números enteros.
- **OCTET STRING.** Almacena una secuencia de bytes.
- **OBJECT IDENTIFIER.** Para representar los identificadores de objetos dentro de el árbol de la MIB. La MIB es una base de datos que mantiene en un sistema jerárquico los identificadores de objetos estándar.
- **BOOLEAN.** Almacena valores que solo pueden tomar verdadero o falso.
- **NULL.** Representa la ausencia de tipo.

Los tipos construidos generan listas y tablas. Destacan los siguientes:

- **SEQUENCE.** Es una lista ordenada de tipos de datos diferentes.
- **SEQUENCE OF.** Es una lista ordenada de tipos de datos iguales.
- **SET.** Es equivalente al "SEQUENCE" pero la lista no está ordenada. Todos los campos tienen que ser diferentes, sino seria ambigua.
- **SET OF.** Es equivalente al "SEQUENCE OF" pero la lista no está ordenada.
- **CHOICE.** Es un tipo de datos en el que hay que elegir uno de entre los tipos disponibles en una lista.

Por último, los tipos definidos son nombres alternados de tipos ASN.1 simples o complejos y generalmente son más descriptivos.

1.2.1.2. Ejemplo

Para ver un ejemplo de como funciona ASN.1 nos basaremos en el mismo ejemplo que aparece en Wikipedia [3], ya que es suficientemente explicativo. Trata sobre la implementación de la capa de presentación de un protocolo Foo que sirve para estandarizar el formato de preguntas de si/no y el formato de las respuestas.

El protocolo se definiría en un modulo de la siguiente manera:

```
FooProtocol DEFINITIONS ::= BEGIN

  FooQuestionIdentifier ::= INTEGER

  FooQuestion ::= SEQUENCE {
    trackingNumber  FooQuestionIdentifier,
    question        IA5String
  }

  FooAnswer ::= SEQUENCE {
    questionNumber  FooQuestionIdentifier,
    answer          BOOLEAN
  }

END
```

Lo primero que podemos ver en el ejemplo es la declaración del módulo. Éste se llama FooProtocol. Dentro se define cada una de las representaciones de los datos que necesita este protocolo.

El primero que declara es un tipo simple FooQuestionIdentifier que equivale al tipo INTEGER.

El tipo FooQuestion esta declarado como tipo SEQUENCE. Esto indica que nuestros datos han de estar ordenados de forma que el primer valor es un identificador de pregunta y, el segundo, es un texto que contiene la pregunta.

Por último, el tipo FooAwnser es otra SEQUENCE cuyo primer valor es el identificador de la pregunta a la cual da respuesta y el segundo contiene un booleano con la respuesta.

1.2.1.3. Codificación BER

Con tal de poder representar físicamente las estructuras de datos ASN.1 en binario, la ITU-T creó el estándar X.690.

El primero de los formatos que presenta es BER (Basic Encoding Rules) que especifica una estructura de codificación auto-contenida y auto-descriptiva. Tiene el siguiente formato:

Identificador	Tamaño	Contenido	Final de contenido

El primer octeto guarda el identificador que contiene la información del tipo, como por ejemplo si es un tipo construido o primario. También contiene que tipo en concreto es (INTEGER, SEQUENCE, ...).

Los siguientes octetos contienen la información del tamaño que ocupa el contenido. Dependiendo del primer octeto, este campo se comporta de diferentes maneras.

- **Forma corta.** Si el primer bit es un 0 se trata de la forma corta. Esto quiere decir que el tamaño sólo ocupa un octeto e indica el número de octetos que contiene. Por ejemplo, 0x4D (01001101 en binario o 77 en decimal) quiere decir tamaño 77 octetos.
- **Forma larga.** Se trata de forma larga si el primer bit es un 1 y contiene algún otro bit a 1. Esto quiere decir que el primer octeto marcará el número de octetos necesarios para indicar el tamaño. Por ejemplo, 0x820112 quiere decir que el campo tamaño tiene 2 octetos y que el tamaño total es de 274 octetos.
- **Forma indefinida.** Finalmente se trata de forma indefinida si el primer bit es un 1 y todos lo demás son 0. Esto quiere decir que para cerrar el contenido encontraremos dos octetos de final de contenido.

El campo *“contenido”* contiene, propiamente, los datos que se quieren transmitir. Este campo puede ser omitido si por ejemplo se trata de tipo NULL.
Finalmente se repite la misma estructura que el identificador indicando el tipo EOC.

1.2.1.4. Codificación DER

También en el estándar X.690 se definió DER (*Distinguished Encoding Rules*). Esta codificación funciona exactamente igual que BER, pero limitando a una las posibilidades de codificar las estructura de datos.

Este formato es muy útil en criptografía, ya que cuando queremos realizar firmas calculamos el hash. Un cambio en un bit hace que el hash sea totalmente diferente; por lo que si utilizamos BER para representar los datos, como se pueden representar de formas diferentes, el hash puede valer cosas totalmente diferentes dando como resultado firmas diferentes.

Por lo tanto, DER se podría considerar como la forma canónica de BER.

1.2.1.5. Codificación PEM

Hay muchos protocolos, como por ejemplo HTTP y MIME, que solo pueden transmitir cadenas de caracteres. Los formatos anteriores están pensados para ser eficientes, y por lo tanto, guardan los datos en binario y no se podrían enviar utilizando esos protocolos.

Para poderlos utilizar se creó otro formato llamado PEM, que no es más que la representación base64 del objeto codificado en DER. La codificación base64 consiste en convertir cada grupo de seis bits y representarlo con un carácter imprimible por ASCII, lo que hace posible la representación en texto.

Para hacer que los humanos lo puedan entender de forma más simple añade también dos límites de encapsulación, uno al principio y otro al final. Un ejemplo seria:

-----BEGIN FOO-----
Codificación base64 de la representación DER de una estructura de datos llamada FOO
-----END FOO-----

1.2.2. JSON

Según Wikipedia [4] “El acrónimo de JavaScript Object Notation y es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia

adopción como alternativa a XML, se considera un formato de lenguaje independiente.”
Las principales ventajas de JSON respecto a XML son: que es mucho más sencillo para el analizador sintáctico procesar la estructura, y que añade menos redundancia, lo que reduce la cantidad de bytes a transmitir.

1.2.2.1. Tipos

La sintaxis de JSON permite representar los siguientes tipos:

- **Números**. Se permiten números y opcionalmente pueden contener decimales separados por punto.
- **Cadenas**. Representa una secuencia de cero o más caracteres. Se ponen entre doble comilla y se permiten cadenas de escape.
- **Null**. Representa el valor nulo.
- **Vector**. Representa una lista ordenada de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes.
- **Objetos**. Son colecciones no ordenadas de pares de la forma <clave>:<valor> separados por comas y puestas entre llaves. La clave tiene que ser una cadena. El valor puede ser de cualquier tipo.

1.2.2.2. Ejemplo

Si quisiéramos representar a un estudiante podríamos hacerlo de la siguiente manera:

```
{
  "name": "Alice",
  "surname": "Foo",
  "birth": "20/04/1996",
  "gender": "female",
  "subjects": [
    {
      "name": "English",
      "mark": 7
    },
    {
      "name": "Security",
      "mark": 9
    }
  ]
}
```

1.3. RSA

Este apartado trata sobre el algoritmo RSA, un ejemplo de criptografía asimétrica. También veremos como se añaden esquemas de relleno para hacerlo más seguro, como se realiza la firma y, finalmente, en qué consiste la firma ciega RSA.

1.3.1. Algoritmo RSA

RSA es un algoritmo de clave pública creado en 1978 por Rivest, Shamir y Adlman, y es el sistema criptográfico asimétrico más conocido y usado.

Para comprender bien cómo funciona RSA tenemos que ver cómo se generan las claves, cuál es el procedimiento para poder cifrar y descifrar y, finalmente, cómo firmar y cómo validar las firmas.

1.3.1.1. Generación de claves

Suponiendo que Alice quiere generar un par de claves RSA, los pasos que debería seguir son los siguientes:

1. Busca un par de números p y q que cumplen la condición de ser primos fuertes.
   - Se entiende por primo fuerte un número primo muy grande.
2. Calcula $n=p\cdot q$.
   - El valor de n es el modulo de la clave privada y pública.
3. Calcula $\phi(n)$, donde ϕ es la función de Euler.
   - La función de Euler ϕ es una función que cuenta el número de coprimos a n menores que n. Para el caso en que n está compuesto por dos números primos, se puede calcular fácilmente como $\phi(n)=(p-1)\cdot(q-1)$.
4. Busca un número e coprimo a $\phi(n)$.
   - El valor de e es el exponente publico y forma la clave pública (n,e).
   - Si e es pequeño puede suponer un riesgo de seguridad. Se suele escoger 65537 (0x10001) como valor.
5. Se determina el valor de d de forma que $e\cdot d\equiv 1 \mod \phi(n)$.
   - Puesto que e y $\phi(n)$ son coprimos, seguro que existe el multiplicador inverso d , que se puede calcular utilizando el algoritmo de Euclides extendido.
   - El valor de d es el exponente privado y forma parte de la clave privada (n,d).

La seguridad de RSA recae en que para poder calcular $\phi(n)$ de forma rápida hay que factorizar n y para valores grandes de n es un problema computacionalmente muy difícil. Gracias a este hecho se hace “imposible” calcular la clave privada a partir de la pública.

Para añadir seguridad a RSA, una de las medidas que se han añadido a esta primera versión de claves es la de dar la posibilidad de que el módulo n esté compuesto por más de dos factores. De esta forma complica aún más la factorización de n.

1.3.1.2. Cifrado y descifrado

Ahora, si Alice quiere enviar la representación del mensaje m de forma confidencial a Bob, y las claves privada y pública de Bob son (n,d) y (n,e) respectivamente, tendrían que seguir el siguiente procedimiento:

1. Alice calcula $c \equiv m^e \mod n$.
2. Alice envía c a Bob.
3. Bob calcula $m \equiv c^d \equiv (m^d)^m \equiv m^{d \cdot e} \mod n$.
   - Esto es debido a que el teorema de Euler dice que $a^{\phi(n)} \equiv 1 \mod n$ y por tanto, se puede demostrar que $m^{d \cdot e} \mod n \equiv m \mod n$

Bob es la única persona que puede descifrar el mensaje ya que es la única que conoce d por lo que se garantiza la confidencialidad.

1.3.1.3. Firma y verificación

En este caso, Alice quiere garantizar que ella es la persona que ha enviado la representación del mensaje m a Bob. Suponiendo que las claves privada y pública de Alice son (n,d) y (n,e) respectivamente, Alice seguiría el siguiente procedimiento:

1. Alice calcula $s = m^d \mod n$.
2. Alice envía s a Bob.
3. Bob recibe la firma y calcula $m \equiv s^e \equiv (m^e)^d \equiv m^{e \cdot d} \mod n$.

El mensaje que recibe Bob sólo puede haber sido enviado por Alice, porque es la única que conoce el exponente privado.

1.3.2. Esquemas de relleno

Los esquemas de relleno son métodos que introducen información irrelevante con cierto objetivo. El rellenado no tiene por qué ser hecho con datos aleatorios (relleno aleatorio). Dependiendo del contexto, el esquema de relleno será distinto para cumplir con el objetivo apropiado a ese contexto.

RSA se debe usar combinado con algún esquema de relleno, ya que si no el valor de m puede llevar a textos cifrados inseguros. No es nada fácil diseñar un esquema de relleno, ya que deben ser cuidadosamente diseñados para prevenir ataques sofisticados.

Como ejemplo de esquemas de relleno para RSA tenemos los definidos en la familia de estándars PKCS (*Public-Key Cryptography Standards*) #1. A continuación detallamos los esquemas de relleno recomendados tanto para cifrado como para firma.

1.3.2.1. EME-PKCS1-v1_5-Encode (M, k)

EME-PKCS1-v1_5 es un esquema que se utiliza para el cifrado RSA. Si queremos añadir relleno a M, cuyo tamaño en octetos es $mLen$, y siendo k el tamaño en octetos del modulo n de las claves RSA, habría que realizar los siguientes pasos:

1. Generar una cadena pseudoaleatoria (sin ceros) de octetos PS de tamaño $k - mLen - 3$. El tamaño mínimo de PS es de ocho octetos.
2. Construir la estructura $EM = 0x00 \,||\, 0x02 \,||\, PS \,||\, 0x00 \,||\, M$.
3. Devolver EM.

1.3.2.2. EMSA-PKCS1-v1_5-Encode $(M, emLen)$

Por otro lado, EMSA-PKCS1-v1_5 se utiliza para la firma. Consiste grosso modo en utilizar una función de hash antes del procedimiento realizado en apartado anterior.

El mecanismo tiene como opciones una función $Hash$ con una salida de $hLen$ octetos. Los parámetros de entrada son el mensaje M y $emLen$ que será el tamaño en octetos de la salida. Seguiría los siguientes pasos:

1. Calcular $H = \mathrm{Hash}(M)$.
2. Codificar en ASN.1 utilizando DER el DigestInfo en la variable T.

```
DigestInfo ::= SEQUENCE {
    digestAlgorithm  AlgorithmIdentifier,  -- Identificador de Hash
    digest           OCTET STRING          -- H
}
```

3. Si $emLen < tLen - 11$, devolver error.
4. Generar una cadena pseudoaleatoria (sin ceros) de octetos PS de tamaño $emLen - tLen - 3$.
5. Construir $EM = 0x00 \,||\, 0x01 \,||\, PS \,||\, 0x00 \,||\, T$.
6. Devolver EM.

1.3.3. Firma RSA

El algoritmo planteado en el apartado 1.3.1.3 es aún ineficiente e inseguro. Para mejorarlo hay que añadir un esquema de relleno que utilice una función

de hash. Esto hace que el mensaje sea de tamaño inferior, lo que agilizará los cálculos, y protegerá contra posibles ataques.

1.3.3.1. Firma

De forma general, para realizar una firma segura, lo primero que tendríamos que hacer es calcular el resumen del mensaje que queremos firmar utilizando una función de hash. Después, se añade el relleno necesario utilizando un esquema de relleno. Para finalizar la firma, como las claves pueden tener dos formatos, se utilizaría la primitiva RSASP1 (explicada en el anexo A.1).

1.3.3.2. Verificación

Como las funciones de hash (criptográficamente seguras) son irreversibles, ya no se puede enviar solamente la firma para obtener el mensaje. Por lo tanto, para mantener integridad de los datos habrá que enviar el mensaje seguido de la firma de este.

Para verificarlo, primero habrá que utilizar la primitiva RSAVP1 (explicada en el A.2). Al valor retornado se le quita el relleno y se compara con el resultado del hash aplicado al mensaje. Si son iguales quiere decir que el mensaje es integro.

1.3.4. Firma ciega RSA

La firma ciega consiste en conseguir que la entidad que firma el mensaje no sea capaz de conocer el contenido real de lo que esta firmado y de esta forma garantizar la privacidad de lo que contiene.

Para hacerlo posible, en RSA se aplica la propiedad de la potencia de un producto para poder, sin necesidad de utilizar la clave privada y modificando la firma, obtener otra firma válida para otro valor conocido. Puesto que esta propiedad de RSA podría ser explotada por un atacante, es anulada cuando se utilizan esquemas de relleno. Por tanto, a continuación vamos a explicar este proceso obviando el proceso “estandarizado” de firma RSA y asumiendo el concepto de que un mensaje cifrado con una clave privada está firmado por el emisor (cualquiera puede descifrarlo con la pública pero sólo el emisor podría haberlo cifrado).

Suponiendo que Bob tiene una clave privada (n,d) y una pública (n,e) válidas, si Alice quiere que Bob calcule la representación de la firma s de la representación de un mensaje m, y no quiere que Bob sepa el contenido, Alice y Bob tendrían que realizar el siguiente procedimiento:

1. Alice escoge un factor de cegado r que cumpla que $mcd(r,n)=1$.
2. Alice calcula dR resolviendo la ecuación modular $r \cdot dR \equiv 1 \mod n$.

- Tiene solución debido a la condición anterior de $mcd(r,n)=1$.
- Para resolverlo se puede aplicar el algoritmo de Euclides extendido.

3. Alice calcula $m_b \equiv r^e \cdot M \mod n$.
   - El valor de m_b es la representación del mensaje cegado.
4. Alice envía la representación del mensaje cegado m_b a Bob.
5. Bob firma el mensaje calculando $s_b \equiv {m_b}^d \equiv (r^e \cdot m)^d \equiv r \cdot m^d \mod n$.
   - El valor de s_b es la representación de la firma del mensaje cegado.
   - Bob realiza la firma como si se tratara de un mensaje sin cegar.
6. Bob devuelve s_b a Alice.
7. Alice calcula $s \equiv dR \cdot m_b \equiv dR \cdot r \cdot m^d \equiv m^d \mod n$.
   - El valor de s es la representación de la firma de m.

1.4. Moneda electrónica

Este apartado trata en qué consiste la moneda electrónica. Primero, veremos un poco de historia sobre cómo ha aparecido. Después, conoceremos los fundamentos básicos en los cuales se construyen las monedas electrónicas y, por último, que monedas hay en la actualidad.

1.4.1. Historia

Por dinero electrónico se entiende dinero que se emite de forma electrónica. Se pueden distinguir dos tipos de moneda electrónica dependiendo de si para hacer el pago hay que estar conectado a internet o se puede hacer fuera de línea.

El dinero electrónico ha sido un problema interesante para la criptografía. Pero su acogida por los usuarios fue bastante lenta. Uno de los primeros éxitos fue el sistema de tarjeta Octopus en Hong Kong, que comenzó como un sistema de pago de tránsito masivo y se ha utilizado ampliamente como un sistema de dinero electrónico. Singapur también ha implementado un sistema de dinero electrónico para su sistema de transporte público (tren, autobús, etc), que es muy similar al de Hong Kong. Otras aplicaciones exitosas las podemos encontrar en los Países Bajos.

Posteriormente, aparecieron sistemas con acogida más global, como por ejemplo, Bitcoin y LiteCoin. Ambos utilizan mecanismos de *proof-of-work* para dar valor a la moneda realizando trabajos computacionalmente complicados.

1.4.2. Fundamentos

En este apartado veremos en qué se basa la moneda electrónica y qué es lo necesario para que sea segura y anónima. Para ello veremos qué relaciones tiene con la moneda física, cómo aplicar firma ciega para hacerla anónima y

finalmente qué hay que hacer para que puedan coexistir monedas con varios valores al mismo tiempo.

1.4.2.1. Sustitución de la moneda física

El objetivo de la moneda electrónica es el de sustituir la moneda física con una moneda digital. La primera de las ventajas es la de liberar a las personas de llevar encima peso. También, por ejemplo, facilita el pago a sus usuarios y lo hace más seguro, además de hacer que las monedas sean más difíciles de falsificar.

Como todo sistema de pago necesita un elemento de intercambio. Este elemento es la moneda electrónica. Para dar valor a la moneda electrónica es necesario que una entidad la firme. En el momento de la firma se descuenta el valor de la moneda electrónica de la cuenta bancaria del usuario. Por el contrario, cuando esta moneda electrónica vuelve al banco, éste devuelve el dinero a la cuenta del usuario que ha depositado la moneda, exactamente como si se tratara de dinero en efectivo.

Para evitar que una moneda sea depositada más de una vez, es necesario diferenciarlas y, por lo tanto, deben tener un identificador único en el sistema.

1.4.2.2. Anónima

Hay una propiedad más de la moneda física que hay que tener en cuenta y es que las transacciones son intrazables, es decir, que no se puede saber por que manos ha pasado una moneda, ni tampoco quien la sacó del banco. El necesitar un identificador único, hace que si el banco es quién distribuye las monedas electrónicas, también puede controlar por donde pasa.

Por lo tanto, para obtener una moneda electrónica, no es el banco quien ha de generar los identificadores sino el usuario, y para garantizar anonimato es necesaria la firma ciega. Una vez el usuario ha escogido un identificador de moneda ha de cegarla y, como quiere que tenga valor, la envía para ser firmada. El banco es incapaz de saber cuál es el identificador de la moneda porque no sabe que factor de cegado ha escogido el usuario. Una vez firmada, el usuario la puede descegar y podrá disfrutar de una moneda válida.

Una complicación que añade este hecho, es que la generación de los identificadores de moneda se hace de forma distribuida, y esto añade la posibilidad de que varios usuarios generen el mismo identificador de moneda. Para evitarlo hay varias estrategias, como por ejemplo añadir una entidad en la cual se registren las monedas dadas de alta.

1.4.2.3. Múltiples valores de moneda electrónica

Con el sistema planteado hasta ahora, solo podemos generar monedas con un único valor. Esto hace que las transacciones de grandes cantidades de dinero sean pesadas, debido a que se tendrían que transmitir cantidades muy grandes de datos. También las hace más difíciles de manejar. Este hecho se podría comparar con qué pasaría si solo existiesen monedas de un céntimo de euro.

Como hace la moneda convencional, hay que permitir más de un posible valor de moneda y para ello hay que hacer que los bancos firmen con más de una clave privada. Cada una de las claves está relacionada con cada posible valor y, por lo tanto, cada firma dará un valor diferente permitiendo una mayor granularidad.

De esta forma a la hora de pedir una moneda, dependiendo de la cantidad deseada, el banco firmará con una clave u otra y restará este dinero de la cuenta del usuario. Y, finalmente, a la hora de depositar el dinero, tendrá que comprobar con qué clave se ha realizado la firma para ingresar esa misma cantidad.

1.4.3. Moneda electrónica en la actualidad

A continuación veremos ejemplos de monedas que funcionan en la actualidad, como los ya mencionados Bitcoin y LiteCoin, y Peercoin.

1.4.3.1. Bitcoin

Según la propia Wiki de Bitcoin [5], “Bitcoin es una moneda electrónica descentralizada creada por el programador Satoshi Nakamoto, quien también desarrolló el cliente original Bitcoin. Permite una participación anónima y segura así como la transferencia de cantidades sin depender de ningún servidor central para procesar las transacciones ni almacenar información de las cuentas.”

También afirma que en la actualidad se acepta a cambio de otras monedas (como dólares o euros), así como directamente para el pago de servicios y bienes tangibles.

1.4.3.2. LiteCoin

LiteCoin utiliza el mismo sistema que Bitcoin, pero es diferente en aspectos como la confirmación de las transacciones, que se produce con mayor rapidez en poco menos de tres minutos.

Adicionalmente, el proceso de mining puede realizarse con equipos que no requieren de gran capacidad y son más comunes entre los usuarios. Esta moneda está en circulación desde el 2011.

1.4.3.3. Peercoin

Esta otra moneda propone seguridad y eficiencia energética como las partes más importantes. Ambos elementos están relacionados con el método que incluye, conocido como *proof-of-stake*, el cual obliga a quienes realizan operaciones con la moneda, a probar que son los legítimos propietarios de la misma.

El sistema *proof-of-shake* es más ecológico que el *proof-of-work* descrito en las otras monedas, ya que los cálculos que realiza son más simples y hacen que la computadora trabaje a menor rendimiento.

Otros de los controles que ofrece ayudan a combatir el proceso colectivo de mining, que se ha catalogado como un fallo para Bitcoin. Funciona desde 2012.

1.5. OpenSSL

OpenSSL es un proyecto de código abierto que provee de una serie de herramientas robustas, de grado comercial y con todas las funcionalidades para los protocolos *Transport Layer Security* (TLS) y *Secure Sockets Layer* (SSL). Además esta desarrollada con el fin de proveer una librería de criptografía con propósitos generales.

La licencia de OpenSSL es estilo Apache, que básicamente significa que eres libre usar el software con fines comerciales o no comerciales mientras respetes unas simples condiciones.

Una vez compilado, OpenSSL ofrece, por un lado, las headers y las librerías estáticas para poder acceder a las funciones directamente desde otras aplicaciones y, por otro lado, provee de una aplicación para acceder vía terminal a funcionalidades de alto nivel, como por ejemplo, cifrar u obtener el hash para una entrada.

1.5.1. Comandos de OpenSSL

Se puede acceder a las funcionalidades de seguridad mediante una aplicación de consola que ofrece. Los comandos más importantes de OpenSSL son los siguientes:

- **rsa.** Procesa las claves RSA. Pueden ser convertidas en varios tipos. Este comando utiliza el cifrado tradicional utilizado en SSLeay.
- **rsautl.** Puede ser usado para firmar, verificar, cifrar y descifrar datos utilizando el algoritmo RSA.
- **dgst.** Sirve para calcular el hash de ficheros o un texto introducido por consola. También se puede utilizar para firmar o validar firmas.

- **req.** Crea y procesa peticiones de certificado en formato PKCS#10. Adicionalmente puede crear certificados auto-firmados para utilizarlos, por ejemplo, en las autoridades de certificación.
- **x509.** Ofrece, de forma genérica, herramientas para certificados. Puede ser usado para imprimir la información de un certificado por pantalla, convertir certificados a varios formatos, firmar peticiones de certificado o editar las configuraciones de confianza de los certificados.
- **pkcs12.** Sirve para crear y parsear ficheros PKCS#12.
- **pkcs8.** Procesa claves privadas en formato PKCS#8.
- **ca.** Es una “mini aplicación” para CAs. Puede firmar peticiones de certificado en una variedad de formatos y mantener una base de datos de texto con los certificados emitidos y sus estados.
- **dsa.** Igual que el comando rsa pero para claves DSA.

1.5.2. Librerías de OpenSSL

Si se quiere acceder a funcionalidades más complejas o, directamente, se quiere incorporar con otro programa, se puede hacer mediante las librerías que ofrece.

1.5.2.1. libcrypto

La librería libcrypto implementa una variedad muy amplía de algoritmos criptográficos usados en varios estándares de Internet. Los servicios provistos por esta librería se utilizan por OpenSSL para las implementaciones de SSL, TLS y S/MIME, y se han utilizado por otros para implementar SSH, OpenPGP y otros estándars criptrográficos.

Una de las estructuras más importante son los BIGNUM. En criptografía se trabaja con números realmente grandes. Por ello, esta estructura sirve para poder representar un número entero de forma general, sin importar su tamaño.

1.5.2.2. libssl

Esta librería implementa los protocolos *Secure Sokets Layer* (SSL v2/v3) y *Transport Layer Security* (TLS v1) y provee de una API con una documentación en línea.

Para poder funcionar, lo primero hay que hacer es inicializarla. Luego, se inicializa el contexto con el cual se quiere trabajar. Aquí se indica que versión de SSL se va a utilizar, certificados, algoritmos, etc. Después se construye la estructura SSL a la que hay que asignarle un socket y finalmente se utilizan las funciones SSL_write y SSL_read para recibir y transmitir información.

Esta librería tiene como dependencia libcrypto.

CAPÍTULO 2. Implementación de firma ciega para OpenSSL

En este capítulo se explican los algoritmos implementados en OpenSSL para que fuera capaz de incorporar firma ciega.

2.1. Clave de cegado

Primero hay que definir la clave de cegado y cómo la generaremos, teniendo en cuenta cuáles son los campos necesarios para cegar y descegar.

Como ambas operaciones las realizaremos en ejecuciones diferentes, es necesario guardar la clave en un fichero. Para ello realizaremos una representación ASN.1 que nos permitirá finalmente guardarla en formato PEM.

2.1.1. Generación de la clave

A simple vista podemos ver que la clave de cegado se puede derivar a partir de la clave pública. Aun así, como la clave pública también la podemos obtener a partir de la privada (para más información consultar el anexo B.1), la clave de cegado se puede derivar a su vez de la clave privada.

Asumiendo que la clave pública asociada a la privada que firmará ciegamente es (e,n) , para generar la clave de cegado seguiremos los siguientes pasos:

1. Escoger un entero aleatorio r que cumpla $\mathrm{mcd}(r,n)=1.$
2. Calcular el factor de cegado como $\beta \equiv r^e \mod n.$
3. Calcular el factor de descegado μ como $\mu \equiv r^{-1} \mod n.$

Para facilitar los cálculos a la hora de cegar y descegar, es más eficiente que la clave de cegado esté compuesta con los valores precalculados, es decir, de β y de $\mu.$

Para su generación, lo más importante es escoger un entero aleatorio r que cumple la condición de ser coprimo a $n.$ La forma más rápida de comprobar que $mcd(a,n)=1$ es utilizar el algoritmo de Euclides. Por lo tanto, aprovecharemos los cálculos para obtener la Identidad de Bezout y obtener $\mu.$ Para calcular β tan solo tenemos que calcular la potencia e modulo n de $r.$

La implementación de este algoritmo ya existe en OpenSSL porque es un cálculo muy recurrido en criptografía. Dicho código lo podemos consultar en el anexo C.1.

2.1.2. Representación ASN.1

La representación de la clave de cegado tendría el siguiente formato:

```
RSABlindKey ::= SEQUENCE {
        blindFactor             INTEGER, -- b
        inverseBlindFactor      INTEGER, -- u
        publicExponent          INTEGER, -- e
        modulus                 INTEGER, -- n
}
```

OpenSSL contiene una serie de macros que permiten crear representaciones de datos en ASN.1. De forma automática permiten también la exportación a formato DER sin tener que añadir nada más. Este código se puede ver en el anexo C.2.1.

2.1.3. Codificación en formato PEM

Para poder guardar la clave de cegado en formato PEM hace falta definir un texto que la encapsule. El que utilizaremos será “RSA BLIND KEY”. El fichero quedará de la siguiente forma:

-----BEGIN RSA BLIND KEY-----
(Codificación base64 de la clave de cegado)
-----END RSA BLIND KEY-----

Para implementar estos cambios hay que utilizar las macros que provee OpenSSL, tal como indica el anexo C.2.2.

2.2. Primitivas de cegado y descegado

Ahora que ya tenemos la clave de cegado, podemos definir las primitivas RSABP1 (RSA blind primitive 1) y RSAUP1 (RSA unblind primitive 1).

2.2.1. RSABP1 (K_b, m)

Entrada:	K_b	clave de cegado RSA. Su formato es (n, β, μ)
	m	representación del mensaje, un número entero entre 0 y $n-1$.
Salida:	m_b	representación del mensaje cegado, un número entero entre 0 y $n-1$.
Error:	“representación del mensaje fuera de rango”.	
Presunciones:	La clave de cegado K_b es valida.	

Pasos:

1. Si la representación del mensaje m no está entre 0 y $n-1$ devuelve, “representación del mensaje fuera de rango” y para.
2. Evalúa $m_b \equiv \beta \cdot m \mod n$.
3. Devuelve m_b.

2.2.2. RSAUP1 (K_b, s_b)

Entrada:	K_b	clave de cegado RSA. Su formato es (n, β, μ)
	s_b	representación de la firma cegada, un número entero entre 0 y $n-1$.
Salida:	s	representación de la firma, un número entero entre 0 y $n-1$.
Error:	“representación de la firma cegada fuera de rango”.	
Presunciones:	La clave de cegado K_b es valida.	

Pasos:

1. Si la representación de la firma cegada s_b no esta no esta entre 0 y $n-1$ devuelve, “representación de la firma cegada fuera de rango” y para.
2. Evalúa $s \equiv \mu \cdot s_b \mod n$.
3. Devuelve s.

2.2.3. Implementación

El hecho de que RSA se base en cálculo de potencias lo hace vulnerable a ataques basados en el tiempo que se tarda en calcular el criptograma. Esto es debido a que si el tamaño del mensaje es grande, para valores grandes del exponente se tarda mucho más tiempo que para exponentes pequeños. Por ello, siempre que en OpenSSL se utiliza la clave privada para firmar o descifrar un mensaje, antes es cegado para modificar el tiempo que tardaría en calcularse y despistar a posibles atacantes.

Si no lo hiciera, un atacante podría determinar aproximadamente el tamaño de la clave de cegado y reducir notablemente el número de claves posibles.

Por ello las primitivas RSABP1 y RSAUP1 están ya implementadas e incluso forman parte de la librería que gestiona los BIGNUM. Las funciones se llaman BN_BLINDING_convert_ex y BN_BLINDING_invert_ex y se pueden encontrar en el anexo C.3.

2.3. Esquemas de cegado y descegado

Finalmente, para implementar realmente firma ciega en OpenSSL, hace falta definir el esquema completo de cómo se realiza el cegado.

Para ello hay que utilizar un esquema de relleno. Esto es necesario, porque si no, para poder firmar, haría falta que el mensaje tuviera el mismo tamaño que el modulo de la clave de cegado. En concreto, utilizaremos el esquema de relleno estándar EME-PKCS1-v1_5, por lo que al esquema completo lo llamaremos RSABS-PKCS1-v1_5 (RSA blind scheme PKCS1-v1_5).

2.3.1. RSABS-PKCS1-v1_5-BLIND (K_b, M)

Entrada:	K_b	clave de cegado RSA. Su formato es (n,β,μ). El tamaño de n es k.
	M	cadena de caracteres del mensaje.
Salida:	M_b	cadena de caracteres del mensaje cegado. Su tamaño es k.
Error:	“representación de la firma cegada fuera de rango”.	
Presunciones:	La clave de cegado K_b es válida.	

Pasos:

1. Hay que aplicar el esquema de relleno EMSA-PKCS1-v1_5 (tal como indica el apartado 1.3.2.2).
   $EM =$ EMSA-PKCS1-v1_5-Encode(M, k)
2. Transforma la cadena de caracteres del mensaje en una representación numérica.
   $m = s2i(EM)$
3. Aplica la primitiva de cegado.
   $m_b = RSABP1(m)$
4. Vuelve a transformar el número en una cadena de caracteres.
   $M_b = i2s(m_b)$
5. Devuelve M_b.

La implementación de este algoritmo se encuentra en el anexo C.4.1.

2.3.2. RSABS-PKCS1-v1_5-UNBLIND (K_b, B_s)

Entrada:	K_b	clave de cegado RSA. Su formato es (n,β,μ). El tamaño de n es k.
	S_b	cadena de caracteres del la firma cegada.
Salida:	S	cadena de caracteres de la firma. Su tamaño es

k.

Error: "representación de la firma cegada fuera de rango".

Presunciones: La clave de cegado es válida.

Pasos:

1. Transforma la cadena de caracteres de la firma cegada en una representación numérica.
   $s_b = s2i(S_b)$
2. Aplica la primitiva de descegado.
   $s = RSAUP1(S_b)$
3. Vuelve a transformar el número en una cadena de caracteres.
   $S = i2s(s)$
4. Devuelve S.

La implementación de este otro algoritmo está en el anexo C.4.2.

2.4. Comandos OpenSSL

Para poder testear las funcionalidades implementadas ampliaremos los comandos de OpenSSl rsa y rsautl.

2.4.1. rsa

La ampliación de este comando consiste básicamente en permitir gestionar claves de cegado.

Por ejemplo, si quisiéramos generar una clave de cegado, con nuestras modificaciones, ahora podemos hacerlo a partir de una clave pública o privada utilizando el siguiente comando:

```
openssl rsa -in private.pem -blout
openssl rsa -in public.pem -pubin -blout
```

Para esta ejecución obtendríamos una clave de cegado como por ejemplo la que aparece en el anexo C.2.2.

También podríamos obtener una clave pública a partir de una clave de cegado.

```
openssl rsa -in blindkey.pem -blin -pubout
```

Podemos encontrar un resumen del código modificado a la herramienta rsa en el anexo C.5.1.

2.4.2. rsautl

Esta otra utilidad sirve para poder cegar y descegar mensajes con las claves generadas usando el comando anterior.

Primero de todo, para cegar utilizaríamos el siguiente comando:

```
openssl rsautl -blind -in digst.txt -inkey blindkey.pem \
    -blin -out blind.bin
```

Después, el firmante tendría que firmar usando el comando tradicional de firma pero especificando que no quiere añadir padding.

```
openssl rsautl -sign -in blind.bin -inkey private.pem \
    -raw -out blindsign.pem
```

Por último, hay que descegar el mensaje utilizando la misma clave de cegado.

```
openssl rsautl -unblind -in blindsign.pem -inkey \
    blindkey.pem -blin -out sign.pem
```

Para validar habría que ejecutar el siguiente comando y comprobar que es igual que el hash del mensaje inicial.

```
openssl rsautl -verify -in sign.pem -inkey public.pem \
    -pubin -out digst.txt
```

Las modificaciones de esta herramienta se encuentran en el anexo C.5.2.

CAPÍTULO 3. Un sistema de moneda electrónica

En este capítulo se explica un ejemplo un sistema que utiliza firma ciega para crear monedas electrónicas anónimas. El objetivo de este sistema no es ser un sistema real, ya que se escapa de los objetivos de este proyecto, sino una prueba de concepto para el uso de la firma ciega que hemos implementado.

3.1. Entidades

Una vez implementada la firma ciega en RSA podemos diseñar un sistema de moneda electrónica.

Lo primero que hay que hacer es un análisis de quiénes son los participantes en la comunicación, de qué necesitan para poder participar (servidores, móvil, bases de datos, etc), de qué responsabilidades tienen y finalmente de cuáles son sus funciones.

En la figura 3.1, que aparece a continuación, podemos ver quiénes son y qué necesitan.

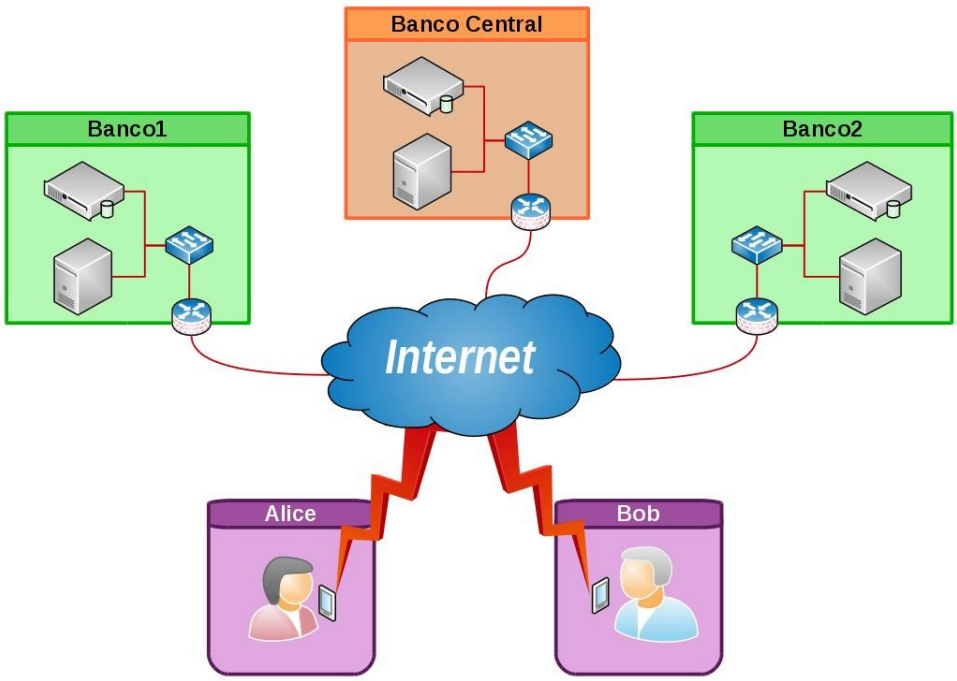

Fig 3.1 *Diagrama de entidades*

3.1.1. Banco Central

Para el sistema hará falta una entidad de confianza para los usuarios finales y para los bancos. Este banco (Banco Central) se encargará de impedir que los usuarios generen mensajes maliciosos y, por otro lado, ha de mantener en anonimato a los usuarios.

Sus funcionalidades serán las siguientes:

- Generar monedas sin valor y guardarlas para mantener un control de las monedas generadas y evitar duplicados.
- Cegar las monedas y firmar el valor cegado para garantizar que no es un mensaje malintencionado.
- Descegar las monedas una vez firmadas por el banco.
- Garantizar intercambios seguros entre varios usuarios finales.
- Eliminar del sistema las monedas que han sido cobradas.
- Distribuir los certificados a los demás bancos, es decir, será la autoridad de certificación (CA) de nuestro sistema.

Será responsable de:

- No dar identificadores de moneda duplicados.
- Que no haya dos usuarios con la misma moneda en su versión actual.
- Que los otros bancos no firmen mensajes inseguros.
- No desvelar información que exponga el anonimato de los usuarios.
- Como CA ha de mantener la lista de certificados revocados (CRL).

Para ello, contará con un servidor HTTP y una base de datos para almacenar las monedas y los intercambios en curso. Además es la única entidad que ha de tener acceso a las nuevas funcionalidades de firma ciega. Las peticiones se gestionarán mediante una API RESTful y sólo podrá haber un único Banco Central.

No podrá ser un servicio autenticado para mantener el anonimato de los usuarios finales.

3.1.2. Bancos

La segunda entidad que aparece en el diagrama son los bancos. Estos tendrán que firmar sin añadir relleno para no estropear el cegado generado por el Banco Central.

Cada uno de los bancos tendrá como funciones dar valor a las monedas añadiendo su firma y su única responsabilidad es mantener el saldo de dinero que tiene cada usuario.

Al igual que el Banco Central, para ofrecer su servicio, cada banco tendrá un servidor HTTP con una API RESTful y una base de datos donde guardar la cantidad de dinero que tiene cada usuario.

Este servicio tendrá que ser autenticado, ya que se necesita saber a qué usuario hay que descontar dinero cuando se generan monedas o a cuál añadir cuando se depositan.

3.1.3. Usuarios finales

La última entidad son los usuarios finales. Para poder gestionar las monedas, utilizarán una aplicación móvil, que guardará en sus teléfonos las monedas generadas, y que provee la implementación de las llamadas necesarias a las APIs para acceder a las funcionalidades de los bancos.

3.2. La moneda

La moneda se guarda en formato JSON, ya que es un formato muy utilizado, compacto y simple. No obstante, este formato tiene un problema a la hora de ser firmado y es que es un formato clave-valor en el que las claves pueden tener cualquier orden. A la hora de representarlo es importante tenerlo en cuenta, ya que dependiendo del orden de las claves, generaría valores diferentes de firma y al verificarlo daría problemas.

Por ello, utilizaremos una versión canónica de JSON en la cual las claves estarán ordenadas alfabéticamente.

3.2.1. Moneda básica

El objeto JSON que representa la moneda tendría el siguiente contenido:

```
{
  "bank": "Common Name del Banco",
  "name": "Tipo de moneda",
  "UUID": "Identificador único universal",
  "value": 1
}
```

“Bank” es el banco que ha firmado esa moneda, *“name”* es el tipo de la moneda, *“UUID”* es un identificador único universal que identifica esa moneda en el sistema y *“value”* es el valor de la moneda (para ese tipo).

3.2.2. Moneda cegada

La moneda anterior es cegada por el Banco Central y añade su firma. Después, el banco del usuario añade su firma directamente sobre el valor cegado. El JSON resultado tendrá la siguiente forma:

```
{
  "content": "Representación base64 de la moneda cegada",
  "sign": [{
    "algorithm": "sha256WithRSAEncryption",
    "signer": "Common Name del Banco Central",
    "value":
      "Representación base64 de la firma de la moneda cegada"
  }, {
    "algorithm": "sha256WithRSAEncryption",
    "signer": "Common Name de la moneda",
    "value":
      "Representación base64 de la firma de la moneda cegada"
  }]
}
```

Donde *"content"* contiene la moneda básica cegada y *"sign"* contiene la firma de ambos bancos.

La moneda cegada es enviada de vuelta al Banco Central para que la desciegue.

3.2.3. Moneda con versión y estado

Como una moneda puede pasar por varias personas, tiene que tener una versión que solo conoce el propietario actual de la moneda. Además necesitamos un estado para indicar si la moneda ya ha sido cobrada o esta actualmente en un intercambio. Estas funcionalidades son propias del Banco Central y por lo tanto tienen que estar fuera de la moneda básica porque sino el Banco Central no podría modificarlas.

En este nivel tendríamos el siguiente objeto:

```
{
  "content": { JSON de una moneda básica },
  "sign": [{
    "algorithm": "sha256WithRSAEncryption",
    "signer": "Common Name de la moneda",
    "value": "Representación base64 de la firma de la moneda"
  }],
  "state": "alive/deleted/trade/blinded",
  "version": "Texto que indica la versión"
}
```

Podemos ver el campo *"state",* que indica como se encuentra la moneda actualmente (cegada, vigente, eliminada o en un intercambio), el campo *"version",* que indica la versión de esta moneda, *"content",* que es el JSON de la moneda y *"sign",* que contiene la firma de la moneda.

3.2.4. Moneda completa

Por último se añade la firma del Banco Central para garantizar que la versión de la moneda y el estado son correctos.

Un ejemplo de moneda completa sería el siguiente:


```
{
  "content": {
    "content": {
      "bank": "Bank 1",
      "name": "euro",
      "UUID": "c56168ff-2635-4938-bd63-62708056c425",
      "value": 1
    },
    "sign": [{
      "algorithm": "sha256WithRSAEncryption",
      "signer": "bank1-euro-1",
      "value":
        "uXADPirs+FOolE098wx4w15bUDt74L0ZUW9QTC7y7xft3VM9rMFm
        RA5y9Wl4X3OMP4FzV4j0Y8pltRcz+QkvR0w3xojuTNXy822qHt+KC
        EaCAKuV9LdWgO8jv8nJ69A7Ww9wZx+99u2CjC9sEwVLQKV2j/xdc5
        4ldBUsNDKQ6Hd/CSgdsjsP4A6H6lnC9uXU/S8eeWBMUbi2+KdriHU
        xv3+n7m7Vs7XT2z2o7Bs01JGbv9janDV619GbvrCA750Zf8CPaKyb
        m/BwxgO4rj8xsE5CiacSJKS1FO5l7D76pXmejFHATjepwCpuPgIqL
        TbqIiDqUkBMla1oXHi/THX8qQ=="
    }],
    "state": "alive",
    "version": "j5744jXR2pR3oRmN"
  },
  "sign": [{
    "algorithm": "sha256WithRSAEncryption",
    "signer": "api.centralbank.com",
    "value":
      "rgNkFPJojIKtaDdTqURCLYKfqH+ai8EEM668RqN3ZiU1rYplghWPZh
      rUrsUMkJwu4eaOmBC0K6LZ+Gt7do5139uGM0XSYKDhBq/UL5WZrYQOJ
      27ZahTnVX+paebqJp782CAUR8EPctzaEXj7aE7lhPbbZ6TSpSvN41Ga
      gd0QJ3GItYfuEElzipfNzMiUNgk1hE57th5sc7mwBSDmZW2n6A0GHtl
      2NZpD7TgERnj3jjcUAsZKpWT2hnemDd77OFnL2PSCwhHZhZBTDSF4oN
      D69m5mrDjR/ff/29HjL+8EtxYzNcuFri/CFIrDlFpSAgP/QUOlU8Aay
      ETqFqouWXil8Q=="
  }]
}
```


3.3. Operaciones

En este apartado veremos los diagramas de comunicación que llevarán a cabo las entidades para poder realizar las operaciones de sacar y depositar dinero, y pagar/cobrar.

3.3.1. Sacar dinero

En la figura 3.2 podemos ver el protocolo que debería seguir Alice si quiere sacar dinero del Banco 1.

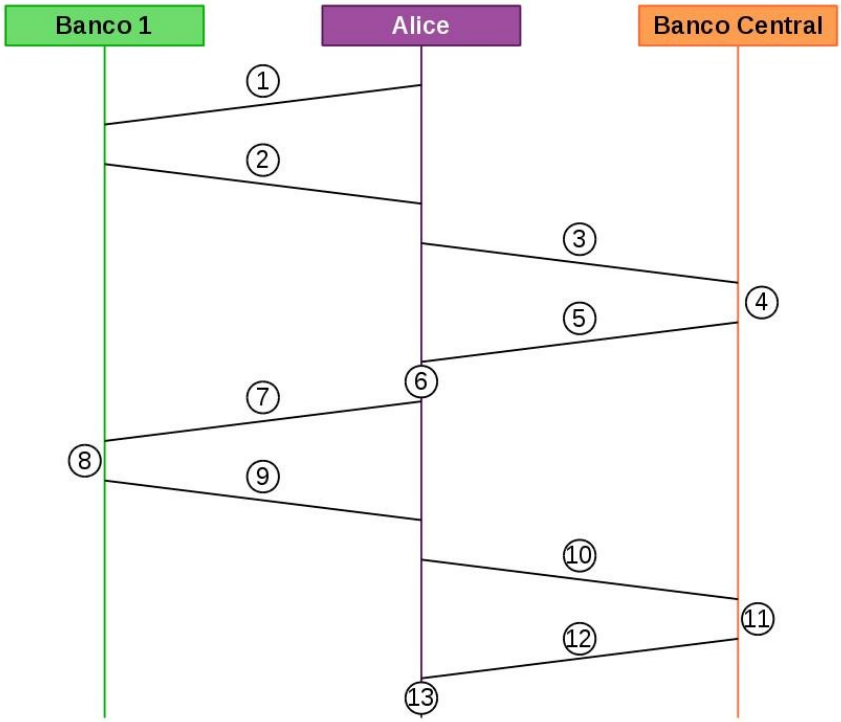

Fig 3.2 *Diagrama para sacar dinero*

1. Alice se autentica en el Banco 1.
2. El Banco 1 envía a Alice un token que la identifica.
3. Alice hace una petición al Banco Central pidiendo que genere una moneda del Banco 1 con un valor y tipo concreto.
4. El Banco Central genera una moneda básica (ver apartado 3.2.1), la ciega y finalmente la firma.
5. El Banco Central responde a Alice con la moneda, el valor cegado y firma (ver apartado 3.2.2).
6. Alice guarda la moneda sin cegar.
7. Alice envía al Banco 1 la moneda cegada con la firma del Banco Central. También indica el tipo y el valor de la moneda y el token que la identifica.
8. El Banco 1 verifica la firma, descuenta el dinero solicitado a Alice y añade la firma de la moneda solicitada (ver apartado 3.2.2).
9. El Banco 1 envía la moneda cegada con las dos firmas.
10. Alice envía la moneda cegada, el *"UUID"* de la moneda y las dos firmas al Banco Central.

11. El Banco Central verifica todas las firmas y desciega el mensaje, le cambia el estado a *"alive"* y añade una versión (ver apartado 3.2.3). A continuación, guarda en su base de datos esta moneda para mantener el estado y versión actual de la moneda. Finalmente añade su firma y ya tenemos la moneda completa (ver apartado 3.2.4).
12. El Banco Central envía la moneda completa a Alice
13. Alice guarda esta moneda para poder utilizarla posteriormente.

3.3.2. Depositar dinero

En la figura 3.3 Alice quiere depositar la moneda que ha sacado en el Banco 2.

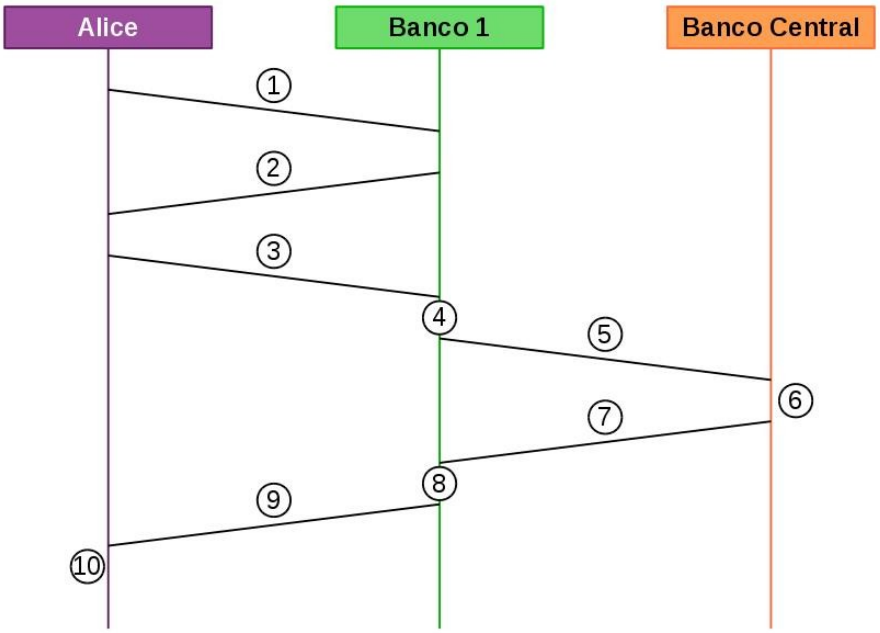

Fig 3.3 *Diagrama para depositar dinero*

1. Alice se autentica en el Banco 2.
2. El Banco 2 envía a Alice un *"token"* que la identifica.
3. Alice envía al Banco 2 la moneda completa (ver apartado 3.2.4) y el *"token"* de sesión.
4. El Banco 2 comprueba la firma del Banco Central (la del Banco 1 no hace falta ya que confía en el Banco Central, y su firma valida la del Banco 1).
5. El Banco 2 hace una petición al Banco Central para eliminar la moneda del sistema.

6. El Banco Central comprueba que la versión de la moneda sea la que tiene apuntada en su base de datos. Si es así, marca la moneda como eliminada.
7. El Banco Central responde al Banco 2 conforme la moneda ha sido eliminada.
8. El Banco 2 añade el dinero a la cuenta de Alice.
9. El Banco 2 responde conforme todo ha ido correctamente.
10. Alice elimina la moneda de su monedero.

Como la moneda ha sido eliminada por el Banco Central, si se repitiese el proceso, el Banco Central notificaría al Banco 2 que ya esta eliminada y no se podría completar la transacción.

3.3.3. Pagos/cobros

Finalmente, la ultima operación que puede hacer Alice es pagar a Bob. El procedimiento lo podemos ver en la figura 3.4.

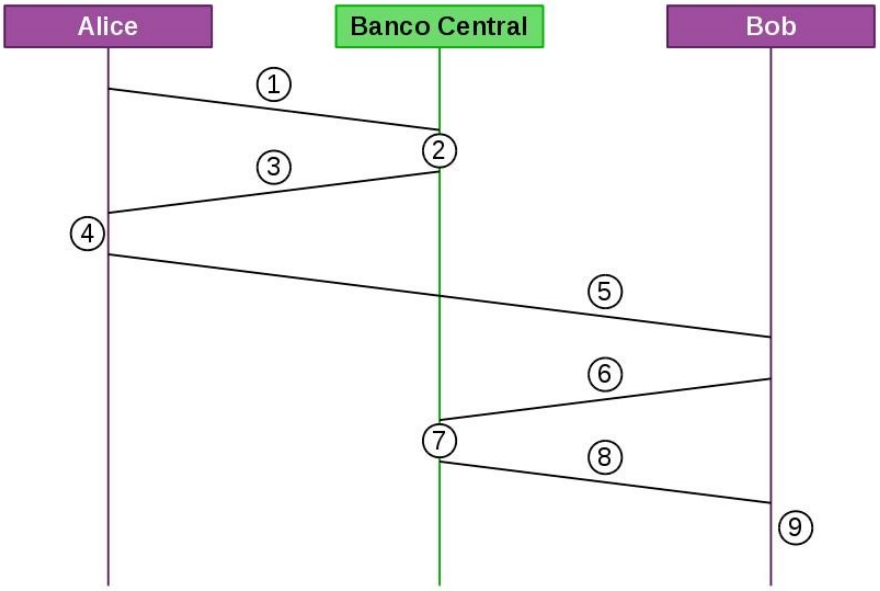

Fig 3.4 *Diagrama para pagos/cobros*

1. Alice envía un conjunto de monedas al Banco Central.
2. El Banco Central genera una nueva versión para cada una de las monedas, firma las monedas y cambia su estado a *"trade"*. Genera un *"UUID"* que identifica el intercambio y lo guarda en su base de datos.
3. El Banco Central envía a Alice el identificador de intercambio.
4. Alice elimina la moneda del monedero.

5. Alice envía a Bob el *"UUID"* del intercambio.
6. Bob envía el *"UUID"* de intercambio al Banco Central.
7. El Banco Central elimina el intercambio de la base de datos y vuelve a cambiar el estado de las monedas a *"alive"*.
8. El Banco Central envía las monedas con la nueva versión a Bob.
9. Bob guarda estas monedas en su monedero.

Si Alice utilizase el *"UUID"* de intercambio para cobrar la moneda antes que Bob sería ella quien recibiría la moneda. Cuando Bob intentase finalizar el intercambio, el Banco Central notificaría conforme no existe ningún intercambio vigente para ese *"UUID"* y no cobraría el dinero por lo que tendría que volver a ponerse en contacto con Alice para poder cobrar.

CAPÍTULO 4. Implementación del sistema de moneda electrónica

En este capítulo mostraremos la implementación del sistema de moneda electrónica definido en el capítulo anterior, detallando cómo funcionan las APIs de los bancos, el cliente para móviles y la configuración de un posible entorno de demostración.

4.1. Entorno de demostración

En este apartado veremos la configuración de un entorno en el cual se puede ver la aplicación en funcionamiento. Para ello veremos que servidores hay y como crear la autoridad de certificación para poder distribuir los certificados.

4.1.1. Servidores

El entorno de demostración consta de un servidor para el Banco Central y dos para otros bancos: Banco 1 y Banco 2. En nuestro caso todas las máquinas utilizarán Debian Jessie como sistema operativo.

En cada uno de los servidores instalaremos los paquetes de NodeJS y mongodb. Para no utilizar direcciones IP, se darán de alta los servidores en algún servidor DNS. Utilizaremos api.centralbank.com para el Banco Central y api.bank1.com y api.bank2.com para otros bancos.

Para hacer más sencillo el desarrollo de las APIs, se utilizará express. Express es un framework para NodeJS que simplifica el desarrollo de APIs RESTful gracias a que proporciona una delgada capa de características para aplicaciones web. Además, utilizaremos mongoose, otro framework que sirve para interactuar con bases de datos mongodb.

4.1.2. Autoridad de Certificación

El sistema de certificados lo gestiona el Banco Central. Para crear la autoridad de certificación hemos de seguir el tutorial que se encuentra en la web de jaimelinux.com [6]. La ruta donde la crearemos será en /srv/ca.

Ahora crearemos un link simbólico a la cadena de certificados de confianza.

```
cd /srv/com.centralbank.api/public/
ln -s /srv/ca/intermediate/certs/ca-chain.cert.pem \
    trusted
```

En el ejemplo se nos recomienda crear dos niveles. Esto sirve para exponer lo mínimo la clave de la autoridad de certificación raíz. Además, si en algún momento creemos que la clave privada del nivel intermedio ha sido obtenida por un usuario malintencionado, todos los certificados generados por esta clave deben ser anulados. Para ello tendremos que añadir el certificado de la autoridad de certificación intermedia a nuestra CRL.

4.1.2.1. Certificado del Banco Central

Una vez tenemos generadas las autoridades de certificación hay que generar el certificado que utilizará el Banco Central para la API. Para ello en el mismo servidor generamos una clave privada RSA.

```
cd /srv/com.centralbank.api/private/keys/
openssl genrsa -out api.centralbank.com 2048
chmod 400 api.centralbank.com
```

Posteriormente generamos la petición de certificado, indicando como Common Name: api.centralbank.com, y finalmente lo firmaremos. También añadiremos los certificados de confianza.

```
cd /srv/ca
openssl req -config intermediate/openssl.cnf \
    -key
/srv/com.centralbank.api/private/keys/api.centralbank.com \
    -new -sha256 \
    -out intermediate/csr/api.centralbank.com.csr.pem

openssl ca -config intermediate/openssl.cnf \
    -extensions server_cert -days 375 -notext -md sha256 \
    -in intermediate/csr/api.centralbank.com.csr.pem \
    -out intermediate/certs/api.centralbank.com.cert.pem

chmod 444 intermediate/certs/api.centralbank.com.cert.pem
cp intermediate/certs/api.centralbank.com.cert.pem \
  /srv/com.centralbank.api/private/certs/api.centralbank.com
cp intermediate/certs/ca-chain.cert.pem \
    /srv/com.centralbank.api/private/certs/trusted

cd /srv/com.centralbank.api/
cp private/certs/api.centralbank.com \
    public/
```

Una vez añadido el certificado reiniciaremos el servicio de la API.

```
service api-central-bank restart
```

4.1.2.2. Certificados de otros bancos

Primero de todo tendremos que descargar el certificado de confianza.

```
cd /srv/com.bank.api/private/certs
wget https://api.centralbank.com/trusted \
    --no-check-certificate
wget https://api.centralbank.com/api.centralbank.com \
    --ca-certificate=trusted
```

Ahora tenemos que generar la clave RSA del banco y de sus monedas. Para ello el banco utilizaremos el comando:

```
# Certificado para el banco
cd /srv/com.bank.api/bin/
./bankCertReq.sh BANK_COMMON_NAME

# Para cada moneda
./bankCertReq.sh BANK_NAME COIN_NAME
```

Después habrá que enviar las peticiones de certificado para que las firme el Banco Central.

4.1.3. Clientes

Nuestro cliente será una aplicación para móviles. Para la implementación se ha utilizado Appcelerator Titanium. Es una IDE (Integrated Development Environment) que permite el desarrollo de aplicaciones para móviles de forma rápida. Una gran ventaja es que las aplicaciones, una vez compiladas son multiplataforma. El lenguaje de programación que utiliza es Javascript que, como es el mismo que las APIs, hará más fácil el desarrollo.

En las últimas versiones han incorporado un framework llamado Alloy. Está basado en el modelo MVC (modelo-vista-controlador), un patrón que separa el modelo de negocio, la vista y el controlador de la vista.

4.2. Paquetes npm

Como ambas APIs tratan sobre moneda electrónica, hay una parte común en el código. Para poder reutilizarlo se han creado paquetes para npm (node package manager), que es un gestor de paquetes para NodeJS. Los paquetes se llaman: blind-node-rsa y blind-coinlib.

4.2.1. blind-node-rsa

Otra motivación era la implementar un sistema que utilizase un lenguaje de bajo nivel, como C, para hacer los cálculos computacionalmente más complejos, y uno de alto nivel, Javascript, para acceder a esas funciones. Precisamente para eso sirve precisamente este paquete.

Esta desarrollado en C++ y sirve para acceder a las nuevas funcionalidades que se han añadido a OpenSSL y poder generar firmas ciegas utilizando RSA.

4.2.1.1. Configuración

Para poder crear addons para node hace falta instalar un paquete llamado node-gyp. Es un comando de consola que permite compilar módulos nativos y añadirlos a NodeJS.

Lo primero que hay que configurar es node-gyp para que pueda generar su Makefile interno. Para ello hay que crear un fichero llamado binding.gyp. Este fichero contiene un JSON que describe el proyecto. En nuestro caso tendrá el siguiente contenido.

```
{
  "targets": [
    {
      "target_name": "blind-node-rsa",
      "include_dirs": ["lib/include"],
      "libraries": [
        "-Wl,-Bsymbolic,../lib/static/libcrypto.a"
      ],
      "sources": [
        "src/module.cc",
        "src/node_rsa.cc",
        "src/node_rsa_utl.cc"
      ]
    }
  ]
}
```

Lo más importante a destacar de esta configuración son los includes y las librerías. Dentro de la carpeta include tenemos una carpeta custom/openssl. Es importante no poner directamente la carpeta de openssl porque NodeJS está compilado utilizando OpenSSL y si queremos diferenciar qué headers utilizar, tenemos que diferenciar las rutas.

Además de añadir los headers modifiacdos de OpenSSL, tenemos que incorporar la librería estática libcrypto. Los flags -Wl,-Bsymbolic hacen que a la hora de compilar se añada la librería entera dentro del binario. Si utilizáramos las funciones normales de OpenSSL, no harían falta estos flags, pero al ser una versión modificada, es necesario que cuando intente acceder a alguna función de OpenSSL, lo haga de nuestra librería y no de las que node tiene guardadas en su ejecución.

Para hacer más sencilla la compilación y el cambio de modo release a debug, podemos crear un pequeño script. En nuestro caso, se ha creado un script que genera un fichero Makefile que permite compilar el proyecto, limpiar los ficheros compilados y generar la estructura de carpetas para poder publicar el módulo a npm.

Otro fichero de configuración importante es el fichero package.json. Este fichero sirve para indicar a npm los metadatos de nuestro proyecto como por ejemplo la versión, el fichero principal o una breve descripción.

Ya que este módulo es de los más importantes y críticos del proyecto, se ha utilizado git para controlar las versiones y así poder diferenciar los cambios más importantes realizados en el código.

4.2.1.2. Funcionalidades

Cuando este paquete se importa en Javascript utilizando la función *require()*, devuelve una clase llamada RSA. Esta clase se instancia utilizando un objeto que admite los siguientes campos:

- **key:** buffer que contiene la clave que se va a utilizar.
- **keyType:** tipo de clave que se va a utilizar. Admite clave pública, privada, de cegado o un certificado. Para llenar este campo tenemos una constante en la clase.
- **trusted:** en caso que keyType sea certificado, en este campo se especifican los certificados en los que se confía. Si no es un certificado confiable, la construcción del objeto devuelve una excepción.
- **password:** en caso de que la clave este cifrada, este campo contiene la contraseña para descifrar.

La clase también provee de las funciones criptográficas: *blind*, *sign*, *unblind*. Cada una de ellas tiene como parámetro un objeto con los siguientes campos:

- **in:** buffer que se quiere procesar.
- **padding:** tipo de padding que se va a utilizar.
- **digest:** funcion hash que se quiere utilizar.

También posee una función llamada "*verify"* que permite comprobar si una firma es correcta. Además de los campos anteriores esta función necesita "*sign"* que contiene la firma del "*buffer"* que se quiere verificar.

La ultima función que provee es la función "*key*" que transforma la clave que hemos introducido en otro formato, como por ejemplo, si queremos exportar una clave de cegado a partir de una pública, o una clave pública a partir de una privada.

4.2.1.3. Implementación

Lo primero que tenemos que hacer para crear un módulo para node es inicializarlo. Para ello hay que añadir el siguiente código:

```
using v8::Local;
using v8::Object;
```

```
void InitAll(Local<Object> exports) {
      NodeRSA::Init(exports);
}

NODE_MODULE(node_rsa, InitAll)
```

La macro *NODE_MODULE* es la que se encarga de añadir el modulo. Necesita dos parámetros, el nombre del módulo y una función para registrarlo. En nuestro caso la función de registro inicializa la clase *NodeRSA*.

El parámetro "*exports*" será lo que devuelva la función *require()* cuando se ejecute sobre nuestro modulo. La función *NodeRSA::Init()* hace de conector entre nuestra clase C++ y el objeto de Javascript. Si la queremos ver, la podemos ver en el anexo D.1.

La definición de la clase *NodeRSA* la podemos ver a continuación:

```
enum KeyRSA {
      NONE        = 0,
      PUBKEY      = 1 << 0,
      CERT        = 1 << 1 | PUBKEY,
      BLKEY       = 1 << 2 | PUBKEY,
      PRIVKEY     = 1 << 3 | PUBKEY,
};

class NodeRSA : public node::ObjectWrap {
public:
      static void Init(v8::Local<v8::Object> exports);

private:
      // Internal
      int _keyType;
      RSA *_rsa;

      explicit NodeRSA(v8::Local<v8::Object> input);
      ~NodeRSA();

      bool checkKey(int needle);
      KeyRSA getKeyType();
      bool setupBlind(v8::Isolate *isolate);

      // Node exports
      static v8::Persistent<v8::Function> constructor;
      static void New(const
                  v8::FunctionCallbackInfo<v8::Value>& args);

      static void Blind(const
                  v8::FunctionCallbackInfo<v8::Value>& args);
      static void Sign(const
                  v8::FunctionCallbackInfo<v8::Value>& args);
      static void Unblind(const
                  v8::FunctionCallbackInfo<v8::Value>& args);
      static void Verify(const
```

```
                v8::FunctionCallbackInfo<v8::Value>& args);

    static void Key(const
                v8::FunctionCallbackInfo<v8::Value>& args);
};
```

Para ver la implementación de las funciones criptográficas podemos ir al anexo D.2.

4.2.1.4. Test unitario

Se ha diseñado un test unitario para garantizar que el paquete funciona, ya que es necesario comprobarlo antes de añadirlo a otros proyectos. Con este test se evalúa la posibilidad de realizar firmas ciegas o convencionales y verificarlas utilizando certificados. También se verifica que es posible transformar las claves a otros formatos y verificar firmas.

4.2.2. blind-coinlib

Este otro paquete sirve para simplificar la gestión de claves, firmar objetos JSON, proveer una forma estándar de cargar ficheros de configuración y ofrece una lista de errores genéricos.

4.2.2.1. Configuraciones

Este paquete está directamente desarrollado en Javascript por lo que su configuración es muy sencilla. La configuración del paquete se encuentra en el package.json y contiene como dependencia el modulo blind-node-rsa.

4.2.2.2. Funcionalidades

Para la gestión de objetos y firmas de objetos JSON, encapsula las funciones del módulo blind-node-rsa en un objeto llamado *ccrypto* (coin crypto).

Por otro lado, para la configuración provee un objeto que guarda en memoria global las configuraciones del servidor. Este objeto se puede inicializar utilizando un JSON.

Por último, tiene una lista de errores genérica que puede retornar el servidor utilizando express. Estos errores son:

- **NotFound.** Sirve para devolver el código HTTP 404 - Not Found cuando un recurso no ha sido encontrado.
- **Forbidden.** Sirve para devolver el código HTTP 400 - Forbidden cuando el cliente esta intentando acceder a un recurso que no esta autorizado.

- **WrongSignature**. Sirve para devolver el código HTTP 403 - Bad Request cuando el usuario esta intentando enviar un JSON con una firma incorrecta.

4.3. APIs

En este apartado veremos como se han implementado las operaciones descritas en el apartado 3.3. Han sido implementas utilizando una arquitectura RESTful utilizando express.

Como base de datos utilizan mongodb, ya que las estructuras de datos que necesitamos guardar se pueden representar de forma más sencilla con documentos que con tablas, y es precisamente a lo que está orientada esta base de datos NoSQL.

4.3.1. Banco Central: centralbank-blcoin-api

Los recursos que gestiona la API del Banco Central son: *bank*, *coin* y *trade*.

4.3.1.1. Recurso bank

Este recurso contiene los bancos soportados por la plataforma. El recurso *"bank"* tiene la siguiente estructura:

```
{
    "name": "The Bank",
    "url": "https://api.thebank.com:443",
    "coins": [
        "euro-1",
        "euro-0.1"
    ]
}
```

Como podemos ver la información que contiene es el nombre del banco, la *"url"* del banco donde se encuentra la API y las monedas que soporta. En este caso vemos que su nombre es "The Bank", su *"url"* es https://api.thebank.com:443 y que soporta las monedas de 1 euro y 10 céntimos.

Si queremos obtener el listado de los bancos disponibles tenemos que hacer una llamada GET a la *"url"* /api/banks. También podemos obtener la información completa de un banco a partir de su nombre, haciendo la petición GET /api/banks/:bank.

4.3.1.2. Recurso coin

Este recurso es el encargado de generar monedas y descegarlas. Si queremos obtener una moneda cegada (ver apartado 3.2.2), hemos de hacer una petición GET /api/coins/:bank/:coin/:value. Para que el usuario que ha pedido la moneda pueda descegarla posteriormente, el Banco Central ofrece la moneda básica (ver apartado 3.2.1). En caso que el usuario envíe la moneda generada a otro banco, o que el importe no se ajuste al especificado, al descegar la firma, esta será invalida.

Para descegar la moneda, haremos una petición POST /api/coins/:UUID. Como contenido enviaremos la moneda con la firma de ambos bancos y obtendremos la moneda completa (ver apartado 3.2.4). En este momento, el Banco Central marcará el **"***UUID"* de la moneda como una moneda válida en el sistema.

Para eliminar una moneda del sistema se hará una petición DELETE /api/coins/:UUID. En caso de que la moneda no exista o ya este eliminada devolverá un error.

4.3.1.3. Recurso trade

Este recurso gestiona los intercambios de monedas electrónicas entre los usuarios.

Para crear un *trade* tenemos que hacer POST /api/trades/ con un vector de las monedas que queremos utilizar. La respuesta que obtendremos es el identificador del intercambio. Una vez creado el intercambio las monedas que tiene el usuario dejan de ser validas porque su ultima versión es diferente.

Para recuperar las monedas en la ultima versión hay que hacer una petición a GET /api/trades/:UUID con el identificador del intercambio generado.

4.3.1.4. Configuraciones

Primero instalaremos la API del Banco Central. Se ha desarrollado un paquete .deb para facilitar la instalación. Para instalarlo tan solo tendremos que utilizar el siguiente comando:

```
dpkg -i api-central-bank.deb
```

El paquete instala los servicios de la base de datos y de la API además de añadirlos en el arranque. Todo el contenido de la aplicación lo instala en la ruta /srv/com.centralbank.com/.

Si queremos modificar la configuración de la API tenemos que editar el fichero private/config.json. Los campos que podemos configurar son:

```
{
    "port": "443",
    "digest": "sha256",
    "common-name": "api.centralbank.com",
    "database_host": "localhost",
    "database_port": "27016",
    "database_name": "centralbank",
    "site-url": "https://api.centralbank.com:443"
}
```

Ahora hay que configurar la base de datos. Los ficheros de la base de datos se encuentra en mongodb/. Para modificar la configuración editaremos el fichero mongodb/mongodb.conf. Una vez esté configurada modificaremos el fichero mongodb/init-database.js para que inserte nuestros bancos y lo ejecutaremos:

```
cd /srv/com.centralbank.com/mongodb
nano init-database.js
cat init-database.js | mongo localhost:27016/centralbank
```

4.3.2. Otros bancos: bank-blcoin-api

Cada banco posee su propia API rest. A diferencia del Banco Central hay que mantener al usuario autenticado y para ello utilizaremos tokens JWT.

4.3.2.1. Llamadas a la API

Para ser autenticados en el servidor del banco, tenemos que realizar una petición GET /api/auth/:mail/:password. A partir de ahora, pondremos el token obtenido en el campo x-access-token de la cabecera http.

Otra petición que pueden realizar los bancos es la peticion GET /api/account/. Con esta petición obtendremos la información de nuestra cuenta y podremos saber del dinero que disponemos en ese banco.

Si queremos sacar dinero del banco tenemos que hacer la petición POST /api/take/:coin/:value con el contenido de la moneda cegada. De esta forma, obtendremos la firma del banco. Cuando queramos depositar una moneda tendremos que hacer una petición POST /api/deposit/ con la moneda completa (ver apartado 3.2.4).

Otra petición que tiene esta API sirve para obtener los certificados de las monedas. Esto es necesario para que el Banco Central pueda generar las monedas cegadas. Para ello hay que realizar la petición GET /api/coins/:coin/:value y obtendremos el certificado de la moneda en formato PEM.

4.3.2.2. Instalación del Banco 1 y 2

Para la instalación de los otros bancos tendremos instalar el paquete api-bank.deb. Para ello utilizaremos el comando:

```
dpkg -i api-bank.dpkg
```

Este paquete instala los servicios de base de datos y de la API y los añade al arranque. Los ficheros instalados se encuentran en /srv/com.bank.api/. Para modificar las configuracies tenemos que editar el fichero private/config.json. Los campos que podemos configurar son:

```
{
    "port": "443",
    "digest": "sha256",
    "secret": "youcantknowit",
    "session-time": 86400,
    "common-name": "api.bank[1/2].com",
    "database_host": "localhost",
    "database_port": "27016",
    "database_name": "bank[1/2]",
    "coins-url": "https://api.centralbank.com:443/api/coins"
    "coin-euro": true
}
```

Si queremos cambiar la configuración de la base de datos tendremos que editar el fichero mongodb/mongodb.conf. Para añadir las cuentas de los usuarios utilizaremos el fichero mongodb/init-database.js como plantilla. Para ello ejecutaremos los siguientes comandos:

```
cd /srv/com.bank.api/mongodb
nano init-database.js
cat init-database.js | mongo localhost:27016/bank[1/2]
```

4.4. Aplicación móvil

En este apartado veremos como funciona la aplicación móvil. La aplicación tiene tres pestañas, una para gestionar las cuentas, otra para gestionar el monedero y por último una para finalizar los intercambios.

4.4.1. Gestión de cuentas

En la primera vista podemos gestionar las cuentas. En esta pestaña podemos ver las cuentas que tenemos registradas. Si pulsamos el botón *Add Account* se nos abrirá una ventana donde seleccionaremos el banco en el que queremos autenticarnos. Para ello tenemos que rellenar el campo e-mail y contraseña y pulsar el botón *Add*. Esta vista se puede ver en la figura 4.1.

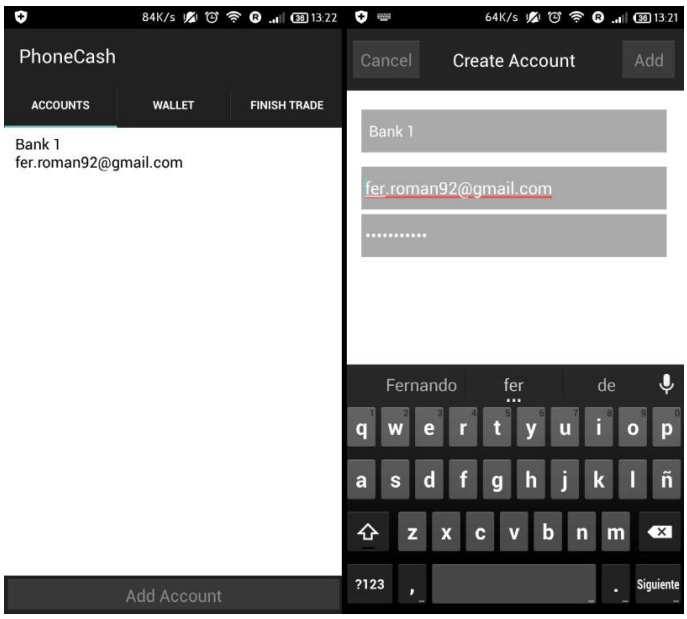

Fig 4.1 *Gestión de cuentas de la aplicación móvil.*

4.4.2. Gestión del monedero

La segunda pestaña es el monedero, y lo podemos ver en la figura 4.2.

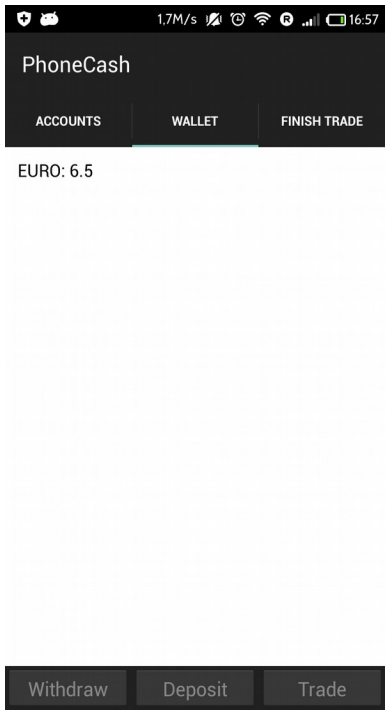

Fig 4.2 *Pestaña de monedero.*

En el monedero podemos consultar el dinero que tenemos actualmente en el móvil.

También podemos sacar dinero del banco, depositarlo o pagar a otro usuario/comerciante pulsando los botones que aparecen en la parte inferior. En la figura 4.3 aparecen las ventanas que se abrirían al pulsar dichos botones.

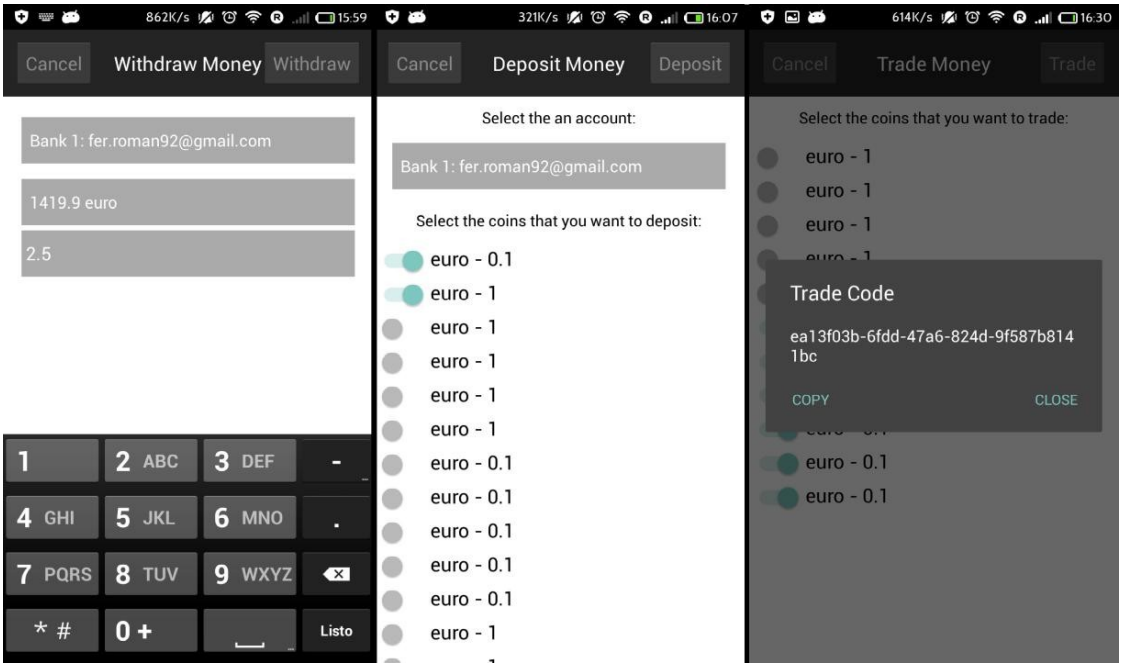

Fig 4.3 *Ventanas para sacar dinero, depositarlo y pagar.*

La primera vista, empezando por la derecha, es en la que los usuarios pueden sacar dinero. Para ello han de especificar de que cuenta y la cantidad de dinero que quieren sacar.

En la segunda vista vemos el listado de monedas que tiene el usuario. Si quiere depositarlas, tan solo ha de seleccionar la cuenta donde las quiere depositar, marcarlas y pulsar el botón *"Deposit"*.

La última vista sirve para pagar a comerciantes o a otros usuarios. Para ello, seleccionaremos las monedas con las que queremos pagar y pulsaremos el botón *"Trade"*. Posteriormente nos aparecerá un mensaje de alerta donde podremos copiar el identificador de intercambio para enviárselo a la persona a la que queremos pagar.

4.4.3. Cobrar

En la figura 4.4 tenemos la vista para cobrar.

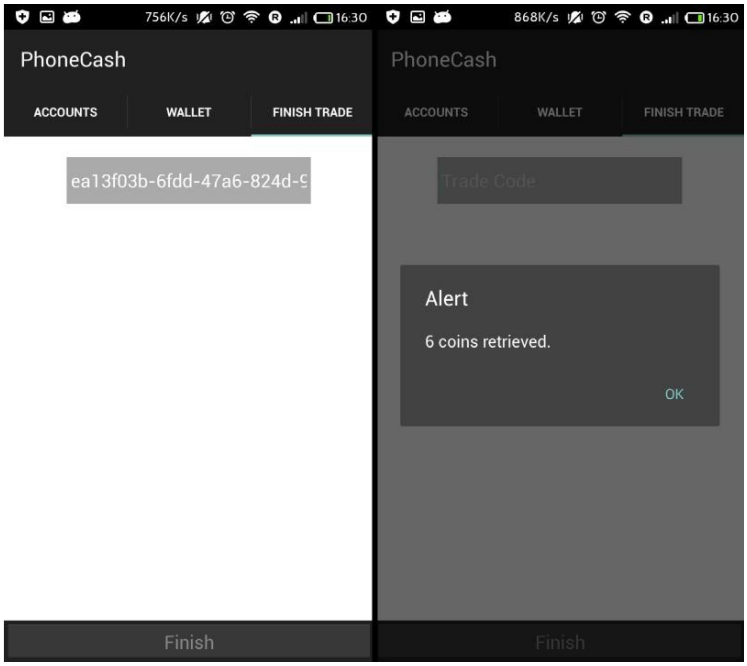

Fig 4.4 *Pestaña para cobrar.*

Para poder cobrar, tendremos que insertar el código del intercambio y pulsar el botón *"Finish"*. Una vez lo hagamos, tendremos ese dinero en nuestro móvil y podremos depositarlo o realizar otros pagos por esta vía.

CAPÍTULO 5. Conclusiones y líneas futuras

El objetivo principal del proyecto ha sido implementar firma ciega en OpenSSL y su consecución ha sido plenamente satisfactoria: las librerías implementan todas las funcionalidades requeridas y el código es perfectamente integrable en la suite OpenSSL.

Otro objetivo era presentar un sistema (prueba de concepto) que hiciese uso de las librerías de firma ciega para OpenSSL. A este respecto, se optó por implementar un sistema de moneda electrónica. Se trata de una implementación *naïve,* no válida para el uso real, lo que requeriría una casuística mucho más compleja en que se tendría que aplicar normativas legales y conceptos económicos. Sin embargo, el sistema desarrollado es plenamente funcional y hace un uso adecuado y potencialmente real de las librerías de firma ciega implementadas.

El ultimo objetivo era el de aprender cosas nuevas. En la elaboración de este proyecto he tenido que utilizar lenguajes de programación que desconocía al iniciar el proyecto, como por ejemplo, C y C++. También he aprendido mucho sobre la gestión de servidores linux, creando servicios y añadiéndolos a diferentes run levels para que se inicien cuando arranca el sistema operativo. Por último, he aprendido conceptos nuevos de telemática como ASN.1 y los formatos de codificación DER, BER y PEM.

La principal línea futura para este trabajo de fin de grado es la inclusión de las librerías desarrolladas en la suite OpenSSL oficial. Si bien, el código, a día de hoy, cumple los requisitos necesarios, OpenSSL sólo incluye código que implemente estándares o RFC. Es por tanto necesario un trabajo de escritura de la firma ciega RSA como un *internet draft*, que pueda posteriormente pasar a RFC e incluso a estándar.

Entre otras posibles líneas futuras estaría conseguir una forma más sencilla para el traspaso del identificador de monedas del intercambio, utilizando protocolos como por ejemplo NFC, que son muy famosos para este tipo de operaciones. También haría falta incorporar mecanismos de *proof-of-work* para dar valor a las monedas electrónicas, como por ejemplo, mediante la búsqueda de colisiones en algoritmos de hash. Por último, se podrían tener en cuenta intercambios bidireccionales, en los que el comerciante devuelva una cantidad de monedas al usuario.

En cuanto al impacto ambiental habría que tener en cuenta el consumo energético que generan los servidores, ya que haría falta una gran cantidad de ellos. Sobretodo teniendo en cuenta la refrigeración. Para ello seria conveniente, para el medio ambiente, situar los servidores en lugares fríos, donde el consumo energético para refrigerar las maquinas fuese considerablemente menor.

Bibliografía

Bibliografía

[1]: Wikipedia. Seguridad de la informacion. (2016). Recuperado de https://es.wikipedia.org/wiki/Seguridad_de_la_información

[2]: Wikipedia. ASN.1. (2014). Recuperado de https://es.wikipedia.org/wiki/ASN.1

[3]: Wikipedia. Abstract Syntax Notation One. (2016). Recuperado de https://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One#Example

[4]: Wikipedia. JSON. (2016). Recuperado de https://es.wikipedia.org/wiki/JSON

[5]: Bitcoin Wiki. Bitcoin. (2011). Recuperado de https://es.bitcoin.it/wiki/Bitcoin

[6]: Jamie Nguyen. OpenSSL Certificate Authority. (2015). Recuperado de https://jamielinux.com/docs/openssl-certificate-authority/

Anexo A. Primitivas RSASP1 y RSAVP1

En el documento PKCS #1 v2.2 se encuentran definidas ambas primitivas de la siguiente forma.

A.1. RSASP1 (K, m)

Entrada:
- K clave privada RSA, donde K tiene uno de los siguientes formatos:
  - Un par (n, d).
  - Una quintupla $(p, q, dP, dQ, qInv)$ y una secuencia de tripletes $(r_i, d_i, t_i), i = 3, \ldots, u$ (posiblemente vacía).
- m representación del mensaje, un número entero entre 0 y $n-1$.

Salida:
- s representación de la firma, un número entero entre 0 y $n-1$.

Error: "representación del mensaje fuera de rango".

Presunciones: La clave privada K es valida.

Pasos:

1. Si la representación del mensaje m no esta no esta entre 0 y $n-1$ devuelve, "representación del mensaje fuera de rango" y para.
2. La representación firma s se calcula de la siguiente forma.
   a) Si se utiliza la primera estructura (n, d) de K, $s \equiv m^d \bmod n$.
   b) Si se utiliza la segunda estructura $(p, q, dP, dQ, qInv)$ y (r_i, d_i, t_i). de K, hay que proceder de la siguiente manera:
      i. Evalúa $s_1 \equiv m^{dP} \bmod p$ y $s_2 = m^{dQ} \bmod q$.
      ii. Si $u > 2$, $s_i \equiv m^{d_i} \bmod r_i$, $i = 3, \ldots, u$.
      iii. Evalúa $h \equiv (s_1 - s_2) \cdot qInv \bmod p$.
      iv. Evalúa $s = s_2 + q \cdot h$.
      v. Si $u > 2$, $R = r_1$ y mientras $i = 3$ hasta u haz:
         1. Evalúa $R = R \cdot r_i - 1$.
         2. Evalúa $h = (s_i - s) \cdot t_i \bmod r_i$.
         3. Evalúa $s = s + R \cdot h$.
3. Devuelve s

A.2. RSAVP1 $((n, e), s)$

Entrada: (n, e) clave pública RSA.

	s	representación de la firma, un número entero entre 0 y $n-1$.
Salida:	m	representación del mensaje, un número entero entre 0 y $n-1$.
Error:	representación de la firma fuera de rango".	
Presunciones:	La clave pública (n,e) es valida.	

Pasos:

1. Si la representación de la firma s no esta no esta entre 0 y $n-1$ devuelve, "representación de la firma fuera de rango" y para.
1. Evalúa $m = s^e \bmod n$.
2. Devuelve m.

Anexo B. Representaciones ASN.1

Este anexo contiene representaciones ASN.1 definidas en estándares que es recomendable conocer para entender mejor el documento.

B.1. ASN.1 de las claves RSA

La clave pública tan solo contiene el exponente público y el modulo.

```
RSAPublicKey ::= SEQUENCE {
        modulus             INTEGER,    -- n
        publicExponent      INTEGER     -- e
}
```

Por otro lado, la clave privada contiene toda la información necesaria para representar la clave pública por lo que se puede derivar de forma trivial.

```
RSAPrivateKey ::= SEQUENCE {
        version             Version,
        modulus             INTEGER, -- n
        publicExponent      INTEGER, -- e
        privateExponent     INTEGER, -- d
        prime1              INTEGER, -- p
        prime2              INTEGER, -- q
        exponent1           INTEGER, -- d mod (p-1)
        exponent2           INTEGER, -- d mod (q-1)
        coefficient         INTEGER, -- (inverse of q) mod p
        otherPrimeInfos     OtherPrimeInfos OPTIONAL
}
```

Anexo C. Código OpenSSL

En este anexo aparecen las modificaciones más importantes que hemos hecho a OpenSSL para implementar firma ciega. También contiene otras partes de código necesarias para entender su funcionamiento.

C.1. BN_BLINDING_create_param

Esta función genera los valores necesarios para la clave de cegado. Como se puede ver, en vez de comprobar si se puede invertir, realizan 32 pruebas aleatorias confiando que si la clave esta bien generada. Si es así prácticamente imposible no encontrar un coprimo.

En su caso utilizan A como factor de cegado y Ai como factor de descegado.

```
BN_BLINDING *BN_BLINDING_create_param(BN_BLINDING *b,
            const BIGNUM *e, BIGNUM *m, BN_CTX *ctx,
            int (*bn_mod_exp) (BIGNUM *r,
                        const BIGNUM *a,
                        const BIGNUM *p,
                        const BIGNUM *m,
                        BN_CTX *ctx,
                        BN_MONT_CTX *m_ctx),
            BN_MONT_CTX *m_ctx)
{
      int retry_counter = 32;
      BN_BLINDING *ret = NULL;

      if (b == NULL)
            ret = BN_BLINDING_new(NULL, NULL, m);
      else
            ret = b;

      if (ret == NULL)
            goto err;

      if (ret->A == NULL && (ret->A = BN_new()) == NULL)
            goto err;
      if (ret->Ai == NULL && (ret->Ai = BN_new()) == NULL)
            goto err;

      if (e != NULL) {
            BN_free(ret->e);
            ret->e = BN_dup(e);
      }
      if (ret->e == NULL)
            goto err;

      if (bn_mod_exp != NULL)
            ret->bn_mod_exp = bn_mod_exp;
```

```
        if (m_ctx != NULL)
                ret->m_ctx = m_ctx;

        do {
                int rv;
                if (!BN_rand_range(ret->A, ret->mod))
                        goto err;
                if (!int_bn_mod_inverse(ret->Ai, ret->A,
                                ret->mod, ctx, &rv)) {
                        /*
                         * this should almost never happen for good RSA
                         * keys
                         */
                        if (rv) {
                                if (retry_counter-- == 0) {
                                        BNerr(BN_F_BN_BLINDING_CREATE_PARAM,
                                                BN_R_TOO_MANY_ITERATIONS);
                                        goto err;
                                }
                        } else
                                goto err;
                } else
                        break;
        } while (1);

        if (ret->bn_mod_exp != NULL && ret->m_ctx != NULL) {
                if (!ret->bn_mod_exp(ret->A, ret->A, ret->e,
                                ret->mod, ctx, ret->m_ctx))
                        goto err;
        } else {
                if (!BN_mod_exp(ret->A, ret->A, ret->e,
                                ret->mod, ctx))
                        goto err;
        }

        return ret;
 err:
        if (b == NULL) {
                BN_BLINDING_free(ret);
                ret = NULL;
        }

        return ret;
}
```

C.2. Clave de cegado en OpenSSL

A continuación veremos que fue necesario añadir a OpenSSL una estructura de datos ASN.1 para representar la clave de cegado y que hizo falta añadir para codificarla en formato PEM.

C.2.1. Representación ASN.1

Para añadir la representación ASN.1 de la clave de cegado a OpenSSL basta con añadir este código:

```
ASN1_SEQUENCE_cb(RSABlindKey, rsa_cb) = {
      ASN1_SIMPLE(RSA, b, BIGNUM),
      ASN1_SIMPLE(RSA, u, BIGNUM),
      ASN1_SIMPLE(RSA, e, BIGNUM),
      ASN1_SIMPLE(RSA, n, BIGNUM),
} ASN1_SEQUENCE_END_cb(RSA, RSABlindKey)

IMPLEMENT_ASN1_ENCODE_FUNCTIONS_const_fname(RSA,
            RSABlindKey, RSABlindKey)
```

Esto genera una estructura en memoria y las funciones necesarias para poder codificar utilizando ASN.1.

C.2.2. Codificación en formato PEM

Para poder codificar en formato PEM en necesario definir el texto que lo encapsulara y la definición de la función en el header de pem.h.

```
# define PEM_STRING_RSA_BLIND    "RSA BLIND KEY"
DECLARE_PEM_rw_const(RSABlindKey, RSA)
```

Para implementar la función hay que utilizar la siguiente macro:

```
IMPLEMENT_PEM_rw_const(RSABlindKey, RSA, PEM_STRING_RSA_BLIND,
           RSABlindKey)
```

El resultado de la codificación PEM de una clave de cegado podría ser como por ejemplo:

```
-----BEGIN RSA BLIND KEY-----
MIIDEgKCAQBCgN0ErE3l5EeTv4T18mm2X16fO72CH4zRQUmem+RHsEoqpLHyYtvX
C7eNIuSLo+Kf9dWi4HDjcTkKhsJT8yMlpvTW8VOykNZK7Rz1kXvwxdbz5OOWV0TM
Tich3BUJtlclhC2JwgjeQSSsgPjt9Yi1wOhSALhSeH/6SkyPj9V4CerN2YSCEvqz
/SUDdy7xJUytWqT3YAFOgtp3HBRkDlER+G0M4h5YeegrkjvOK0zEDo3UdMmHSh3j
cXrrrV9Z9R4Jmh8sZ5kUFxPL94STxTPSAgtxFsBYCf8wXK+pMlZARjJ4TlZ19SKm
Rokm48F2s5zByWax0DaSMo411QEaG3HyAoIBAHwATPZB8kxdyAXDQ3wttm6z7aU7
JuL9rXgNxPJMY5g6f5xvHhuzhF9AzZB7I4x7H/M+ItKIMVW4zu8U8bNFsPeGgnqo
o9a4IyU6smfBbetvIOsxvYHkmL7nJ8c+YMY7rJDSk9mtuVG3u8GguQvOGxgPJO6Z
D2rvPDlz8m0IGVojJmSsnSF6eiu+Asq9+pgw66c0TKaWpXA40NKTvLwUzGjPyUER
1esJfM9ftVBDntk5trqH2DQofhmpVbiW8FreUd3wfg7daq8GbemBctTO1Jk8o22V
agosciVx/Sav8Jqjqq3htNp9Y1C40ARpxfARisG7dL6lewqJ4EqP5TGC7TwCAwEA
AQKCAQEAsYSApz712Th7bnHvTTTqCQ6TRd0quYtjyfcmAiwWXFDmKw6eY+DGPCfg
z6ZHi0efzH+2ApnD3z3TkqKSczRXjgdXTKydP+RAKXJqUg0BN+7lICiVu+8pEDNT
zpBBn6PbxIxuITAZ3g9dDHwLtfId+hQZ90uF0DcNBkzAWVh5L7ouxlYtKZXuSxmL
Pt0eIuQQtKhnpGen6OnGqhXD85il/mY3/8sPt2xTIclUE6775du5XxCyllXzgw3G
aBdGdeW8xpa852I6DeATbixZGrGuIf+cBP5/H8fP37/e1ty6lHJQd0FsjV3mJ/uN
bTR8RATwo5ru2XWjmdXgzM0OTHXawQ==
-----END RSA BLIND KEY-----
```

C.3. BN_BLINDING_convert_ex/BN_BLINDING_invert_ex

En este apartado veremos las funciones que implementan las primitivas RSABP1 y RSAUP1. Son funciones que ya existían en OpenSSL y se llaman BN_BLINDING_convert_ex y BN_BLINDING_invert_ex.

```
int BN_BLINDING_convert_ex(BIGNUM *n, BIGNUM *r, BN_BLINDING *b,
            BN_CTX *ctx)
{
      int ret = 1;

      bn_check_top(n);

      if ((b->A == NULL) || (b->Ai == NULL)) {
            Bnerr(BN_F_BN_BLINDING_CONVERT_EX,
                        BN_R_NOT_INITIALIZED);
            return (0);
      }

      if (b->counter == -1)
            /* Fresh blinding, doesn't need updating. */
            b->counter = 0;
      else if (!BN_BLINDING_update(b, ctx))
            return (0);

      if (r != NULL) {
            if (!BN_copy(r, b->Ai))
                  ret = 0;
      }

      if (!BN_mod_mul(n, n, b->A, b->mod, ctx))
            ret = 0;

      return ret;
}

int BN_BLINDING_invert_ex(BIGNUM *n, const BIGNUM *r,
            BN_BLINDING *b, BN_CTX *ctx)
{
      int ret;

      bn_check_top(n);

      if (r != NULL)
            ret = BN_mod_mul(n, n, r, b->mod, ctx);
      else {
            if (b->Ai == NULL) {
                  Bnerr(BN_F_BN_BLINDING_INVERT_EX,
                              BN_R_NOT_INITIALIZED);
                  return (0);
            }
            ret = BN_mod_mul(n, n, b->Ai, b->mod, ctx);
      }

      bn_check_top(n);
```

```
        return (ret);
}
```

C.4. Esquemas de firma ciega

Este apartado contiene el código necesario para implementar los esquemas de firma ciega RSABS-PKCS1-v1_5-BLIND y RSABS-PKCS1-v1_5-UNBLIND.

C.4.1. RSABS-PKCS1-v1_5-BLIND

```
static int RSA_eay_public_blind_on(int flen,
            const unsigned char *from, unsigned char *to,
            RSA *rsa, int padding)
{
        BIGNUM *ret;
        int i, j, k, num = 0, r = -1;
        unsigned char *buf = NULL;
        BN_CTX *ctx = NULL;
        /*
         * Used only if the blinding structure is shared.
         * A non-NULL unblind instructs rsa_blinding_convert()
         * and rsa_blinding_invert() to store the unblinding
         * factor outside the blinding structure.
         */
        BN_BLINDING *blinding = NULL;

        if ((ctx = BN_CTX_new()) == NULL)
                goto err;
        BN_CTX_start(ctx);
        ret = BN_CTX_get(ctx);
        num = BN_num_bytes(rsa->n);
        buf = OPENSSL_malloc(num);
        if (!ret || !buf) {
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                                ERR_R_MALLOC_FAILURE);
                goto err;
        }

        switch (padding) {
        case RSA_PKCS1_PADDING:
                i = RSA_padding_add_PKCS1_type_1(buf, num, from,
                                flen);
                break;
        default:
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                                RSA_R_UNKNOWN_PADDING_TYPE);
                goto err;
        }
        if (i <= 0)
                goto err;
```

```
        if (BN_bin2bn(buf, num, ret) == NULL)
                goto err;

        if (BN_ucmp(ret, rsa->n) >= 0) {
                /* usually the padding functions would catch this */
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                           RSA_R_DATA_TOO_LARGE_FOR_MODULUS);
                goto err;
        }

        blinding = rsa_get_blinding_key(rsa, ctx);
        if (blinding == NULL) {
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                           ERR_R_INTERNAL_ERROR);
                goto err;
        }

        if (!BN_BLINDING_convert(ret, blinding, ctx))
                goto err;

        /*
         * put in leading 0 bytes if the number is less than the
         * length of the modulus
         */
        j = BN_num_bytes(ret);
        i = BN_bn2bin(ret, &(to[num - j]));
        for (k = 0; k < (num - i); k++)
                to[k] = 0;

        r = num;
 err:
        if (ctx != NULL)
                BN_CTX_end(ctx);
        BN_CTX_free(ctx);
        BN_BLINDING_free(blinding);
        OPENSSL_clear_free(buf, num);
        return (r);
}
```

C.4.2. RSABS-PKCS1-v1_5-UNBLIND

```
static int RSA_eay_public_blind_off(int flen,
                const unsigned char *from, unsigned char *to,
                RSA *rsa, int padding)
{
        BIGNUM *f, *ret;
        int i, j, k, num = 0, r = -1;
        BN_CTX *ctx = NULL;
        /*
         * Used only if the blinding structure is shared.
         * A non-NULL unblind instructs rsa_blinding_convert()
         * and rsa_blinding_invert() to store the unblinding
         * factor outside the blinding structure.
         */
```

```
        BN_BLINDING *blinding = NULL;

        if ((ctx = BN_CTX_new()) == NULL)
                goto err;
        BN_CTX_start(ctx);
        f = BN_CTX_get(ctx);
        ret = BN_CTX_get(ctx);
        num = BN_num_bytes(rsa->n);
        if (!f || !ret) {
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                                ERR_R_MALLOC_FAILURE);
                goto err;
        }

        if (BN_bin2bn(from, num, ret) == NULL)
                goto err;

        blinding = rsa_get_blinding_key(rsa, ctx);
        if (blinding == NULL) {
                RSAerr(RSA_F_RSA_EAY_PRIVATE_ENCRYPT,
                                ERR_R_INTERNAL_ERROR);
                goto err;
        }

        if (!BN_BLINDING_invert(ret, blinding, ctx))
                goto err;

        /*
         * put in leading 0 bytes if the number is less than the
         * length of the modulus
         */
        j = BN_num_bytes(ret);
        i = BN_bn2bin(ret, &(to[num - j]));
        for (k = 0; k < (num - i); k++)
                to[k] = 0;

        r = num;
 err:
        if (ctx != NULL)
                BN_CTX_end(ctx);
        BN_CTX_free(ctx);
        BN_BLINDING_free(blinding);
        return (r);
}
```

C.5. Comandos modificados a OpenSSL

En este apartado veremos un resumen del código modificado las herramientas de OpenSSl.

C.5.1. rsa

En primer lugar se ampliaron las opciones de la herramienta para que admitiera claves de cegado como entrada y salida.

```
{"blin", OPT_BLIN, '-', "Input a blind key"},
{"blout", OPT_BLOUT, '-', "Output a blind key"},
```

En segundo lugar se añadió la posibilidad de cargar claves de cegado:

```
{
      EVP_PKEY *pkey;

      if (pubin) {
            int tmpformat = -1;
            if (pubin == 2) {
                  if (informat == FORMAT_PEM)
                        tmpformat = FORMAT_PEMRSA;
                  else if (informat == FORMAT_ASN1)
                        tmpformat = FORMAT_ASN1RSA;
            } else
                  tmpformat = informat;

            pkey = load_pubkey(infile, tmpformat, 1, passin, e,
                        "Public Key");
      } else if(blin) {
            int tmpformat = -1;
            if(informat == FORMAT_PEM)
                  tmpformat = FORMAT_PEMRSA;
            else
                  tmpformat = informat;

            pkey = load_blkey(infile, tmpformat, 1, passin, e,
                        "Blind key");
      } else
            pkey = load_key(infile, informat, 1, passin, e,
                        "Private Key");

      if (pkey != NULL)
            rsa = EVP_PKEY_get1_RSA(pkey);
      EVP_PKEY_free(pkey);
}
```

A continuación se añadió la llamada a la función para generar las claves:

```
if(blout && !blin) {
      if(!RSA_setup_blkey(rsa, NULL))
            goto end;
}
```

Y finalmente se añadió la posibilidad de devolver la clave en formato PEM

```
if (outformat == FORMAT_PEM) {
      if (pubout || (pubin && !blout)) {
            if (pubout == 2)
                  i = PEM_write_bio_RSAPublicKey(out, rsa);
```

```
            else
                  i = PEM_write_bio_RSA_PUBKEY(out, rsa);
      } else if(blout || blin) {
            i = PEM_write_bio_RSABlindKey(out, rsa);
      } else {
            assert(private);
            i = PEM_write_bio_RSAPrivateKey(out, rsa,
                        enc, NULL, 0, NULL, passout);
      }
}
```

C.5.2. rsautl

Primero, se añadieron las opciones para hacer posible aceptar como entrada una clave de cegado y realizar las operaciones de cegado y descegado.

```
{"blin", OPT_BLIN, '-', "Input is an RSA blind"},
{"blind", OPT_BLIND_ON, '-', "Blind with a blind key"},
{"unblind", OPT_BLIND_OFF, '-', "Unblind with a blind key"},
```

Después, se añadió el código necesario para poder aceptar como entrada una clave de cegado.

```
case KEY_BLIND:
{
      int tmpformat = -1;
      if(keyformat == FORMAT_PEM)
            tmpformat = FORMAT_PEMRSA;
      else
            tmpformat = keyformat;

      pkey = load_blkey(keyfile, tmpformat, 0, NULL, e,
                  "Blind Key");
}
      break;
```

Y por último, se añadió la llamada a los esquemas de cegado y descegado.

```
case RSA_BLIND_ON:
      rsa_outlen = RSA_public_blinding_on(rsa_inlen, rsa_in,
                  rsa_out, rsa, pad);
      break;

case RSA_BLIND_OFF:
      rsa_outlen = RSA_public_blinding_off(rsa_inlen, rsa_in,
                  rsa_out, rsa, pad);
      break;
```

Anexo D. Código blind-coin-rsa

Este anexo contiene el código más importante que contienen los servidores. Primero veremos la implementación del módulo blind-coin-rsa, y posteriormente las APIs.

D.1. Inicialización

Para ver al detalle como exportar la clase de C++ a Javascript necesitamos implementar la función *NodeRSA::Init()*.

```
void NodeRSA::Init(Local<Object> exports)
{
      Isolate* isolate = exports->GetIsolate();

      /*
       * Add algorithms in order to decrypt keys or check
       */   certificates!
      OPENSSL_add_all_algorithms_noconf();

      // Prepare constructor template
      Local<FunctionTemplate> tpl =
                  FunctionTemplate::New(isolate, New);
      tpl->SetClassName(String::NewFromUtf8(isolate, "RSA"));
      tpl->InstanceTemplate()->SetInternalFieldCount(1);

      // Constants
      NODE_RSA_KEY_CONSTANT(tpl, isolate, PUBKEY);
      NODE_RSA_KEY_CONSTANT(tpl, isolate, BLKEY);
      NODE_RSA_KEY_CONSTANT(tpl, isolate, PRIVKEY);
      NODE_RSA_KEY_CONSTANT(tpl, isolate, CERT);

      NODE_KEY_CONSTANT(tpl, isolate, PADDING_PKCS1,
                  RSA_PKCS1_PADDING);
      NODE_KEY_CONSTANT(tpl, isolate, PADDING_NONE,
                   RSA_NO_PADDING);

      // Prototype
      NODE_SET_PROTOTYPE_METHOD(tpl, "blind", Blind);
      NODE_SET_PROTOTYPE_METHOD(tpl, "sign", Sign);
      NODE_SET_PROTOTYPE_METHOD(tpl, "unblind", Unblind);
      NODE_SET_PROTOTYPE_METHOD(tpl, "verify", Verify);
      NODE_SET_PROTOTYPE_METHOD(tpl, "key", Key);

      constructor.Reset(isolate, tpl->GetFunction());
      exports->Set(String::NewFromUtf8(isolate, "RSA"),
                  tpl->GetFunction());
}
```

D.2. Funciones criptográficas

Otra parte importante de la clase *NodeRSA* es la implementación de las funciones criptográficas.

```
void NodeRSA::Blind(const FunctionCallbackInfo<Value>& args)
{
      Isolate* isolate = args.GetIsolate();
      NodeRSA *self = NodeRSA::Unwrap<NodeRSA>(args.Holder());
      unsigned char md[SHA256_DIGEST_LENGTH];
      OP_ARGS *op_args = NULL;
      unsigned char *buff_in, *buff_out;
      int olen = 0, ilen = 0;

      if(!(op_args = OP_ARGS_new(args))) {
            goto err;
      }

      if(!self->setupBlind(isolate)) {
            goto err;
      }

      if(op_args->md && (EVP_Digest(op_args->buffer,
                  op_args->length, md, NULL,
            op_args->md, NULL))) {
            buff_in = md;
            ilen = SHA256_DIGEST_LENGTH;
      } else {
            buff_in = (unsigned char *)op_args->buffer;
            ilen = op_args->length;
      }

      buff_out =(unsigned char *)OPENSSL_malloc(
                  RSA_size(self->_rsa));
      if(!(olen = RSA_public_blinding_on(ilen, buff_in,
                  buff_out,
            self->_rsa, op_args->padding))) {
            NODE_THROW_EXCEPTION_err(isolate,
                        _E_M_SOMETINK_WRONG);
      }

      args.GetReturnValue().Set(Buffer::New(isolate,(char *)
                  buff_out, olen).ToLocalChecked());

err:
      OP_RSA_free(op_args);
}

void NodeRSA::Sign(const FunctionCallbackInfo<Value>& args)
{
      Isolate* isolate = args.GetIsolate();
      NodeRSA *self = NodeRSA::Unwrap<NodeRSA>(args.Holder());
      unsigned char md[SHA256_DIGEST_LENGTH];
      OP_ARGS *op_args = NULL;
      unsigned char *buff_in, *buff_out;
```

```
      int olen = 0, ilen = 0;

      if(!(op_args = OP_ARGS_new(args))) {
            goto err;
      }

      if(!self->checkKey(KeyRSA::PRIVKEY)) {
            NODE_THROW_EXCEPTION_ret(isolate,
                        _E_M_KEY_NOT_PRIVATE);
      }

      if(op_args->md && (EVP_Digest(op_args->buffer,
                  op_args->length, md, NULL, op_args->md, NULL)))
      {
            buff_in = md;
            ilen = SHA256_DIGEST_LENGTH;
      } else {
            buff_in = (unsigned char *)op_args->buffer;
            ilen = op_args->length;
      }

      buff_out = (unsigned char *)OPENSSL_malloc(
                  RSA_size(self->_rsa));
      olen = RSA_private_encrypt(ilen, buff_in, buff_out,
                  self->_rsa, op_args->padding);
      if(olen < 0) {
            NODE_THROW_EXCEPTION_ret(isolate,
                        _E_M_SOMETINK_WRONG);
      }

      args.GetReturnValue().Set(Buffer::New(isolate, (char *)
                  buff_out, olen).ToLocalChecked());

err:
      OP_RSA_free(op_args);
}

void NodeRSA::Unblind(const FunctionCallbackInfo<Value>& args)
{
      Isolate* isolate = args.GetIsolate();
      NodeRSA *self = NodeRSA::Unwrap<NodeRSA>(args.Holder());
      OP_ARGS *op_args = NULL;
      unsigned char *buff_in, *buff_out;
      int olen = 0, ilen = 0;

      if(!(op_args = OP_ARGS_new(args))) {
            goto err;
      }

      if(!self->setupBlind(isolate)) {
            goto err;
      }

      buff_in = (unsigned char *)op_args->buffer;
      ilen = op_args->length;
      buff_out = (unsigned char *)OPENSSL_malloc(
```

```
                RSA_size(self->_rsa));
        if(!(olen = RSA_public_blinding_off(ilen, buff_in,
                buff_out, self->_rsa, op_args->padding))) {
            NODE_THROW_EXCEPTION_err(isolate,
                    _E_M_SOMETINK_WRONG);
        }

        args.GetReturnValue().Set(Buffer::New(isolate, (char *)
                buff_out, olen).ToLocalChecked());

err:
        OP_RSA_free(op_args);
}
void NodeRSA::Verify(const FunctionCallbackInfo<Value>& args)
{
        Isolate* isolate = args.GetIsolate();
        OP_ARGS *op_args = NULL;
        NodeRSA *self = NodeRSA::Unwrap<NodeRSA>(args.Holder());
        unsigned char md[SHA256_DIGEST_LENGTH];
        Local<Value> val;
        Local<Object> input, sign;
        unsigned char *s = NULL, *m = NULL, *v = NULL;
        int vlen = 0, mlen = 0;
        size_t slen = 0;
        bool ret = false;

        if(!(op_args = OP_ARGS_new(args))) {
            goto err;
        }
        input = args[0]->ToObject();

        // Check in parameter
        val = input->Get(String::NewFromUtf8(isolate, "sign"));
        if(val->IsUndefined() || !Buffer::HasInstance(val)) {
            NODE_THROW_EXCEPTION_err(isolate,
                    _E_M_INPUT_BUFFER_NOT_SPECIFIED);
        }
        sign = val->ToObject();

        s = (unsigned char *)Buffer::Data(sign);
        slen = Buffer::Length(sign);
        v = (unsigned char *)OPENSSL_malloc(RSA_size(self->_rsa));
        if(!(vlen = RSA_public_decrypt(slen, s, v, self->_rsa,
                op_args->padding))) {
            NODE_THROW_EXCEPTION_err(isolate,
                    _E_M_SOMETINK_WRONG);
        }

        if(op_args->md && (EVP_Digest(op_args→buffer,op_args→
                op_args->length, md, NULL,op_args->md, NULL))) {
            m = md;
            mlen = SHA256_DIGEST_LENGTH;
        } else {
            m = (unsigned char *)op_args->buffer;
            mlen = op_args->length;
        }
```

```
	if(mlen != vlen)
		goto err;

	for(int i = 0; i < mlen; i++)
		if(m[i] != v[i])
			goto err;

	ret = true;

err:
	OP_RSA_free(op_args);
	args.GetReturnValue().Set(Boolean::New(isolate, ret));
}
```