UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ELIMINACIÓN DE PUNTOS CRÍTICOS DE FALLA EN EL SISTEMA THRESHOLD CRYPTOGRAPHY HSM

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FRANCISCO JOSÉ CIFUENTES QUIJADA

PROFESOR GUÍA 1:
JAVIER BUSTOS JIMÉNEZ
PROFESOR GUÍA 2:
ALEJANDRO HEVIA ANGULO

MIEMBROS DE LA COMISIÓN:
PATRICIO POBLETE OLIVARES
JOCELYN SIMMONDS WAGEMANN
FEDERICO MEZA MONTOYA

ESTE TRABAJO HA SIDO PARCIALMENTE FINANCIADO POR NIC CHILE RESEARCH LABS

SANTIAGO DE CHILE
ENERO 2016

Resumen

En un sistema distribuido, los puntos críticos de falla son aquellos nodos tales que si sólo uno de ellos falla, el sistema completo deja de funcionar. La no existencia de estos permite tener sistemas robustos y tolerantes a fallas.

Threshold Cryptography Backend para DNSSEC es un sistema desarrollado previamente a este trabajo, el cual permite firmar electrónicamente una zona DNS de manera distribuida con el uso criptografía umbral. El autor de esta tesis, en un trabajo previo, desarrolló una interfaz para que el Threshold Cryptography Backend sea utilizado por otras herramientas. La interfaz consistía en dos elementos: una biblioteca que implementa la API PKCS#11, estándar para soluciones criptográficas, y un nodo capaz de escuchar a esta biblioteca y transformar las solicitudes al Threshold Cryptography Backend. La interfaz permite que el sistema sea utilizado como una virtualización de un módulo de seguridad por hardware (HSM por sus siglas en inglés), por parte de cualquier aplicación diseñada para que use estos dispositivos. Por lo cual, el nuevo sistema, que incluye tanto la nueva interfaz como el Threshold Cryptography Backend, adquiere el nombre de Threshold Cryptography HSM. Este nuevo sistema consta de varios nodos: el nodo pkcs11, el nodo maestro, y de los nodos firmadores, además de los nodos que manejan la comunicación entre cada uno de estos, por medio del software de mensajería RabbitMQ. Sin embargo, dada las características del sistema original y los requerimientos de la API, dicho sistema presenta puntos críticos de falla, lo cual deja vulnerable al sistema en su conjunto.

La presente tesis, describe el trabajo realizado para rediseñar totalmente el sistema Treshold Cryptography HSM, con la finalidad de eliminar los puntos críticos de fallas. Esto se logró trasladando funcionalidad entre nodos. El primer traslado de funcionalidad consistió en los mecanismos de comunicación: se cambiaron los nodos de mensajería RabbitMQ por un protocolo de mensajería implementado por la biblioteca ZeroMQ. El segundo traslado de funcionalidad, consistió en mover varios algoritmos de criptografía umbral que corrían en el nodo maestro, hacia el nodo pkcs11. Originalmente, los algoritmos de criptografía umbral estaban implementados en el lenguaje de programación Java, sin embargo la API PKCS#11 está diseñada para ser usada por el lenguaje de programación C. Por lo tanto, el traslado de la funcionalidad criptográfica requirió reescribir los algoritmos en el lenguaje de programación C. Para enfrentar este desafío, se desarrolló una biblioteca en C que implementaba la funcionalidad criptográfica necesaria para eliminar el punto crítico de falla. De este modo, el nodo pkcs11 utiliza esta biblioteca, incorporando así parte de la funcionalidad del nodo maestro.

Este desarrollo permitió, entre otras cosas, una mejora en desempeño y una simplificación del código. Adicionalmente, quedó como subproducto la implementación inédita de un protocolo criptográfico, el algoritmo de firma criptografica umbral RSA. Dicha implementación quedó disponible bajos los términos una licencia de software libre tipo MIT. Este protocolo de criptografía umbral, cumple la propiedad de no falsificabilidad aún cuando nodos no comprometidos reciben una solicitud de firmar un documento falsificado. Esta propiedad de seguridad permiten asegurar que el sistema es robusto. Como toda funcionalidad criptográfica, requiere un cuidado especial en el desarrollo y testing, por tanto se diseñó un mecanismo para realizar pruebas del sistema basado en la equivalencia entre sistemas distribuidos y sistemas concurrentes. A partir de esto, se procedió a utilizar model checking como mecanismo de verificación formal para estudiar la correctitud del sistema.

A Macarena Palacios y nuestra Miel.

Agradecimientos

- A mi familia por todo el apoyo brindado.
- A mi mamá, por enseñarme a soñar.
- A Macarena Palacios por ayudarme a soñar y por el amor que me ha dado en cada momento en que he trabajado en esta tesis.
- A NIC Chile Research Labs por todo el conocimiento otorgado y el excelente ambiente.
- A Nicolás Aguilera, Sebastián Blasco, Camilo Gómez y Boris Romero, por compartir nuestro camino a la graduación.
- A mis profesores guía Javier Bustos y Alejandro Hevia, por su invaluable ayuda durante la investigación.
- A la profesora Jocelyn Simmonds por introducirme en el tema de métodos formales.

Tabla de Contenido

Capítulo 1

Introducción

No es extraño encontrarse con organizaciones en donde las decisiones deban tomarse entre varias personas. Un ejemplo típico de esto es un directorio corporativo: cuando el directorio toma una determinación y desea hacerla pública, se escribe un documento, luego cada uno de los directores toma su pluma lo firma. Cuando todos los directores ya firmaron el documento, las firmas certifican que éste es válido y oficial.

¿Cómo se haría este procedimiento cuando los documentos son digitales?

Las firmas digitales existen hace ya bastante tiempo. Estas herramientas, ayudan a simplificar el proceso de certificación y validación cuando los documentos son de naturaleza digital. Sin embargo, en la actualidad, la única manera de utilizarlas de manera práctica es confiando en el ente centralizado que administra el procedimiento de certificación. Si un directorio desea firmar digitalmente un documento, debe recurrir a una implementación centralizada.

Si bien este enfoque centralizado es seguro y útil en variados casos, no siempre se puede confiar en el procedimiento a ciegas.

Se puede imaginar un sistema en el que cada director firma parcialmente el documento digital con su *smartphone*, y le envía el resultado al resto de los directores. Cuando todos los directores han firmado el documento, se podría generar el certificado. Mejor aún, podemos pensar en un sistema que sea tolerante al compromiso o falla de el smartphone de algún director –o incluso la corrupción de este–. En él, se exige que la generación del certificado se realice con algún número k, menor a la totalidad de directores, permitiendo tolerancia a fallas sin que el sistema quede expuesto (mientras más de k directores sean confiables).

¿Existen implementaciones, tales que permitan simular de manera realista el procedimiento de firmar parcialmente un documento, de manera realmente distribuida? ¿Qué tan flexibles pueden ser estas implementaciones?

En esta tesis se busca dar una respuesta a esta pregunta.

1.1. DNS y DNSSEC

Internet es una red de redes, descentralizado desde su concepción. Toda la infraestructura de Internet está diseñada para que la comunicación no dependa de ningún ente centralizado.

El sistema de nombres de dominio (DNS por sus siglas en inglés) sigue este diseño. DNS es el sistema responsable de asociar *domain names*, nombres designados por un administrador de sistemas que identifica a una máquina dentro de una red de computadores, con sus direcciones IP respectivas. DNS es en gran parte responsable de la masiva adopción de Internet en el mundo, debido a que simplifica el proceso de identificación de máquinas en una red, permitiendo que los nombres identificadores sean fáciles de memorizar.

El conjunto de todos los nombres representables por el sistema corresponde al espacio de nombres de DNS. El sistema puede representarse como un árbol, cuya jerarquía representa la responsabilidad que tiene cada nodo con sus hijos. La raíz (`root`) del árbol almacena la información de los *top level domain* (TLD), que son los primeros dominios en tener etiquetas identificables, como "`com.`", "`org.`" o "`cl.`". A su vez, cada TLD almacena información de los dominios que están bajo su jerarquía dentro del árbol de nombres de dominio. Por ejemplo, el TLD "`cl.`" almacena la información para llegar a la dirección IP de "`uchile.cl.`". En la figura 1.1 se muestra una representación del árbol que determina la jerarquía del espacio de nombres. La jerarquía comienza en la raíz de DNS.

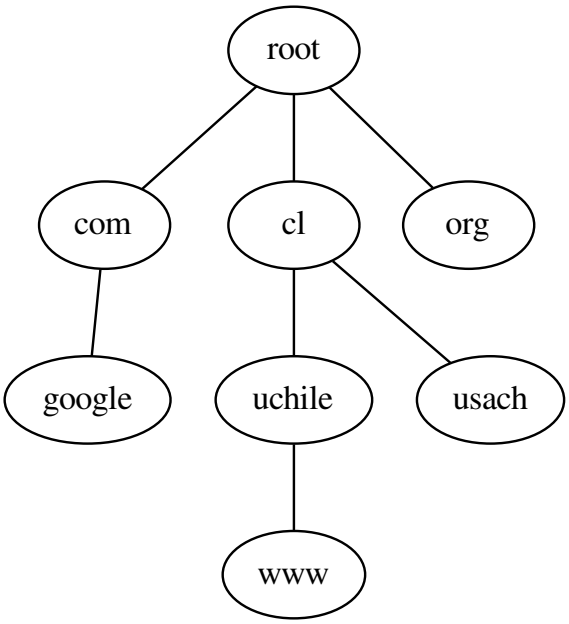


Figura 1.1: Árbol de nombres de DNS. La responsabilidad de "`www.uchile.cl.`", recae sobre los nodos "`www`", "`uchile`", "`cl`" y `root`

En DNS existen dos componentes de software principales: servidores autoritativos de un

dominio y *resolvers*.

Los servidores autoritativos almacenan un conjunto de registros –llamado zona– asociados a un dominio en particular bajo su jerarquía. Es decir, los servidores autoritativos son los responsables de responder ante una consulta sobre un dominio que esté bajo su jerarquía. Los servidores autoritativos pueden delegar la responsabilidad de responder acerca de un dominio bajo su jerarquía a otro servidor autoritativo más cercano a las hojas dentro del árbol.

Los *resolvers* son la capa de software que está encargada de realizar consultas DNS a los servidores autoritativos para poder determinar la dirección IP de un *hostname* en particular [27].

En la figura 1.2 se puede observar como un *resolver* traduce un hostname a una dirección IP. Para realizar esto, el resolver le va preguntando por el hostname a cada servidor autoritativo del árbol del espacio de nombres, hasta llegar al servidor autoritativo para ese dominio.

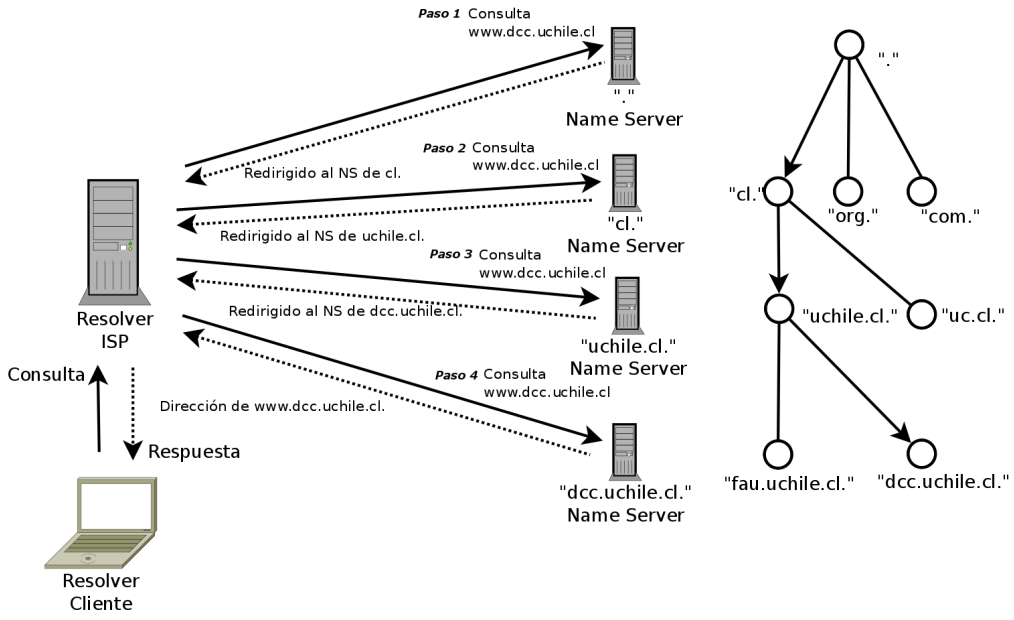


Figura 1.2: Esquema de una consulta al sistema DNS. Un cliente le solicita al resolver que traduzca el hostname "`www.dcc.uchile.cl.`".

A pesar de ser un sistema con extenso uso en Internet, tiene vulnerabilidades conocidas que son especificas para su protocolo. Un ejemplo de ello son los ataques del tipo *hombre en el medio* (*man in the middle* en inglés). Este ataque consiste en la intercepción de una consulta DNS por parte de un adversario, respondiendo éste con una dirección IP de un computador distinto al que realmente pertenece el *hostname* consultado. Este tipo de ataques puede utilizarse para falsificar sitios web disimuladamente, con la finalidad de robar contraseñas [3]. En la figura 1.3 se muestra un esquema que representa un ataque de hombre en el medio. En el diagrama, el nodo A le desea consultar al nodo B la dirección IP correspondiente a "`www.uchile.cl.`". En el ataque, C se interpone en la consulta y le responde con una IP que puede llevarlo a un sitio web malicioso.

Para enfrentar las vulnerabilidades, se diseñaron las extensiones de seguridad de DNS,

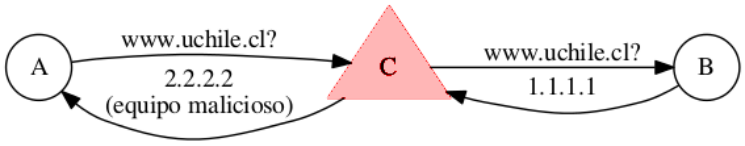


Figura 1.3: Esquema que representa un ataque de tipo *hombre en el medio*.

DNSSEC [2]. Estas extensiones proveen autenticación de origen y verificación de integridad de los datos de DNS utilizando infraestructura de llave pública [44]; así como también mecanismos para distribuir las llaves públicas que se usan para la autenticación. Gracias a la autenticación, un ataque del tipo *hombre en el medio* no se puede realizar, ya que se tiene la certeza de quien responde no es malicioso.

Sin embargo, la utilización de estas extensiones está mermada por la dificultad que presenta la administración de las llaves criptográficas que se utilizan para firmar los registros. La adopción de DNSSEC requiere de decisiones técnicas y operacionales que impactan en la administración de un servidor autoritativo de DNS. Por esta razón, el uso de DNSSEC no está totalmente extendido entre los TLD[1]. En el contexto chileno, el TLD local (NIC Chile) desplegó DNSSEC en el año 2011[2], permitiendo que quienes tienen nombres de dominio bajo la jerarquía de NIC Chile puedan desplegarlo también. Sin embargo, el uso de DNSSEC en nombres de dominios chilenos, sigue siendo muy bajo. Según fuentes de NIC Chile, al 22 de enero de 2016, "`.cl.`" tiene en su jerarquía 48 dominios con DNSSEC, de un total de 493596 dominios registrados.

Existen dos escenarios típicos que enfrentan los administradores de DNS: (i) Se tiene un servidor autoritativo principalmente estático, donde los registros tienen una tasa de cambio baja, o (ii) se tiene un servidor autoritativo de alta rotación de registros, donde los registros son creados, modificados o borrados frecuentemente. Este último puede ser el caso de los TLD. En ambos escenarios, DNSSEC introduce complejidades. En (i), los registros tendrán que ser actualizados con mayor frecuencia que cuando no se utilizaba DNSSEC, ya que las firmas tienen fecha de expiración por motivos de seguridad. En (ii), cualquier cambio en los registros requiere una actualización en las firmas de ellos, por lo que el proceso debe ser automatizado de alguna manera.

Además, las herramientas criptográficas que normalmente se utilizan para la implantación de DNSSEC –firmas RSA estándar– requieren de un ente centralizado encargado del procedimiento de firmado.

Es por esto que en DNSSEC, por su naturaleza distribuida, se puede desear la utilización de firmas parciales para la generación de certificados. Para evitar la centralización de DNSSEC, se podrían enviar a firmar parcialmente los registros de la raíz a cada una de los TLD, dejando de depender en la solución centralizada que existe en la actualidad.

[1]Para mayor información ver `http://www.internetsociety.org/deploy360/dnssec/maps/`
[2]`https://www.nic.cl/tecnologia/dnssec.html`

1.1.1. Administración de llaves en DNSSEC

En una zona de DNS se almacenan dos llaves criptográficas: ZSK (*Zone Signing Key*, llave utilizada para firmar una zona) y KSK (*Key Signing Key*, llave utilizada para firmar una ZSK). Para tener una política de seguridad adecuada, los registros del servidor DNS deben dejar de ser estáticos. La ZSK tiene una validez limitada, lo que implica que la zona necesita ser firmada nuevamente antes que las llaves expiren. Además tanto la KSK como la ZSK deben ser renovadas periódicamente, ya que corren el riesgo de ser descifradas. La criptografía moderna se basa en que usando fuerza bruta sólo se necesita suficiente tiempo y poder computacional para lograr desencriptar un texto cifrado. Esto implica que a medida que pasa el tiempo se vuelve más probable que las llaves sean descubiertas [22].

Para enfrentar la complejidad del manejo de llaves y firmas en DNSSEC se han desarrollado múltiples herramientas que ayudan a manejar la implantación de DNSSEC. Existen herramientas tanto libres (OpenDNSSEC[3], BIND[4]) como privativas (Infoblox[5], Bluecat DNS[6]). Estas soluciones se encargan de realizar automáticamente el firmado de registros y la renovación de llaves, entre otras operaciones que facilitan la vida a los administradores de sistemas.

Para el almacenamiento seguro de datos sensibles, como llaves privadas u otros, la industria ha desarrollado hardware especializado. Estos dispositivos son llamados Hardware Security Modules o HSMs, los cuales tienen mecanismos físicos especializados para evitar intromisión y robo de datos sensibles [30]. Para acceder a estos módulos, se estandarizó la API PKCS #11 [18]. Esta API esta definida como una serie de funciones para ser llamadas desde el lenguaje de programación C.

Cuando se habla de seguridad en DNSSEC, el almacenamiento cuidadoso de llaves es crítico. OpenDNSSEC –la principal herramienta libre para el manejo de llaves sobre DNSSEC– delega el almacenamiento de llaves a algún HSM al que se tenga acceso. De esta manera, las organizaciones que administren un servidor DNS pueden elegir cuánto gastar o qué nivel de seguridad quieren tener para almacenar sus llaves.

Además del hardware especializado, existen implementaciones virtuales de un HSM. Por ejemplo, los desarrolladores de OpenDNSSEC crearon SoftHSM, el cual implementa la API PKCS #11. Mediante este software, se puede utilizar la herramienta sin ningún costo adicional. Sin embargo, el nivel de seguridad y confiabilidad de SoftHSM recae en la configuración de un computador no diseñado para estos fines. Si alguien logra tener acceso físico a ese computador, fácilmente puede robar las llaves. En otras palabras, SoftHSM no posee ningún tipo de certificación de seguridad.

[3] `https://www.opendnssec.org/`
[4] `https://www.isc.org/downloads/bind/`
[5] `https://www.infoblox.com/`
[6] `https://www.bluecatnetworks.com/products/dns/`

1.1.2. Problemática derivada de la utilización de HSMs.

Hay dos conceptos que sirven para describir una problemática de la solución actual de almacenamiento de llaves para DNSSEC: la robustez y la tolerancia a fallas. Para los efectos de esta tesis, la robustez se define como la capacidad del sistema de mantener la seguridad aún cuando parte de él es controlado por un usuario malicioso. A su vez, la tolerancia a fallas se define como la capacidad del sistema de seguir funcionando en caso de que parte de él falle.

Al utilizar un HSM para almacenar las llaves de DNSSEC se está buscando seguridad, la cual se consigue especialmente mediante métodos físicos como la limpieza de memoria ante intrusión. Sin embargo, al tener un punto único de almacenamiento de llaves se pierde la tolerancia a fallas que un sistema como DNS necesita. Además, si un usuario malicioso logra tener acceso a las llaves almacenadas en el HSM, va a lograr obtener el control total del sistema.

Por otro lado, el costo de las HSM tiene una notoria correlación positiva con el nivel de seguridad. Las soluciones de bajo costo carecen de los mecanismos que impiden a intrusos obtener datos privados. Estos mecanismos pueden ser el sellado especial de la máquina, el borrado inmediato de claves ante ataques físicos, etc.

1.2. TCHSM, Threshold Cryptography HSM

Durante el año 2011, Barros et al. [5] propusieron utilizar criptografía umbral en el contexto de DNSSEC, por medio de la creación de un software llamado Threshold Cryptographic Backend. Este software sería una herramienta de automatización en el manejo de llaves, reemplazando software existente (por ejemplo, OpenDNSSEC). Threshold Cryptographic Backend utilizaría criptografía umbral (threshold crytography en inglés) para el almacenamiento seguro de llaves.

A grandes rasgos, la criptografía umbral tiene como propósito la distribución de funcionalidad criptográfica entre varios nodos, y en donde el éxito de una operación depende de un cierto parámetro de umbral. Tal parámetro define la cantidad mínima de resultados parciales válidos, recibidos desde los nodos, para generar un resultado total válido.

Parte del desarrollo de Threshold Cryptographic Backend fue entregado a NIC Chile Research Labs. Este desarrollo que incluye un sistema de criptografía umbral distribuido básico, que sirve de base o inspiración para cualquier trabajo posterior.

En la propuesta de Threshold Cryptographic Backend se planteó que el sistema era:

- Distribuido: El sistema se implementa utilizando n nodos.
- Tolerante a falla: Un subconjunto de nodos puede fallar sin causar problemas a la totalidad del sistema.
- Robusto: Las fallas se pueden mitigar utilizando diferentes implementaciones.

- Seguro: Ningún nodo guarda la llave completa. Más de k nodos deben estar comprometidos para romper el sistema.

Dichas propiedades son sumamente importantes para el éxito del sistema. Por otro lado, el trabajo mencionado plantea la creación de un nuevo robot firmador que se encargue del mantenimiento de los registros DNS.

Algunos de los sistemas existentes que automatizan el firmado de registros DNS, como OpenDNSSEC o BIND, utilizan HSMs para el almacenamiento seguro de llaves. En particular, para la comunicación con estos dispositivos, OpenDNSSEC utiliza la API PKCS #11 [18].

El autor de esta tesis, en un trabajo previo [10], desarrolló una interfaz para que las aplicaciones que utilizan la API PKCS #11 pudieran utilizar parte de Threshold Cryptographic Backend. La idea de ese proyecto era tener un mecanismo mediante el cual el sistema de criptografía umbral sea interoperable con OpenDNSSEC, proporcionándole las funcionalidades criptográficas que necesite. Para realizar esto, se desarrolló una implementación de la API PKCS #11, la cuál se comunica con el sistema de criptografía umbral. Tal mecanismo consiste en un sistema de RPC, Remote Procedure Call, el cual sirve para la invocación de funciones de manera remota. El mecanismo de RPC desarrollado (figura 1.4), comunica una librería C que implemente la API PKCS #11 con la implementación del sistema criptográfico de Barros et al [5, 10].

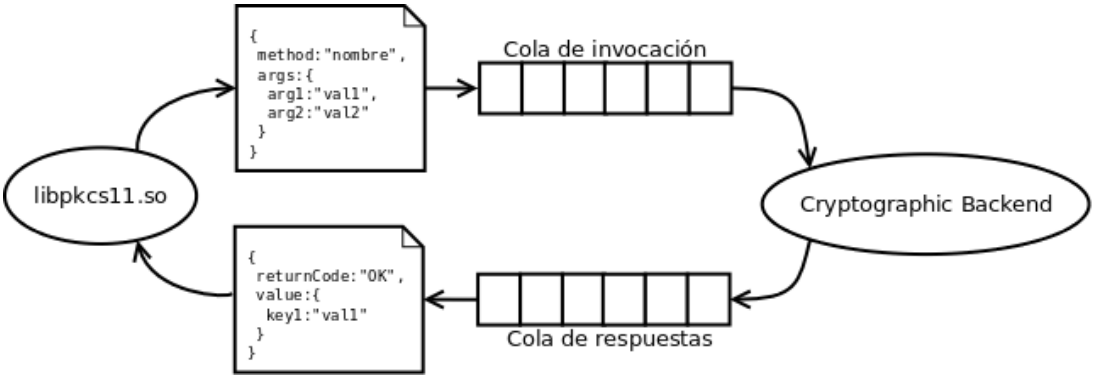


Figura 1.4: Idea general del mecanismo de RPC. La cola de respuesta es única por cada invocación de un método. La cola de invocación es la misma para toda invocación. Esto permite un funcionamiento asíncrono y persistente.

Una ventaja no esperada del sistema es que permite el uso de otras aplicaciones distintas a OpenDNSSEC y en ámbitos muy distintos. Todo gracias a la implementación de una API estándar como lo es PKCS#11. Por este motivo dicho sistema deja de ser exclusivo a DNSSEC y pasa a llamarse de manera genérica Threshold Cryptography HSM (TCHSM).

1.3. Descripción general del problema a estudiar

En un sistema distribuido, los puntos críticos de falla son aquellos nodos tales que si sólo uno de ellos falla, el sistema completo deja de funcionar. La no existencia de estos permite tener sistemas robustos y tolerantes a fallas.

La interfaz desarrollada por el autor de esta tesis [10], incluye puntos de falla críticos, quitando en gran parte las ventajas que tiene la utilización de un sistema de criptografía umbral: la tolerancia a fallas. Dichos puntos de falla críticos serán descritos en las secciones que vienen.

La tolerancia a fallas es sumamente importante dado que TCHSM puede reemplazar a los HSMs dentro de sistemas que requieren confiabilidad, disponibilidad y seguridad. Por lo tanto es un desafío técnico importante diseñar una solución a este problema.

Este trabajo busca estudiar las posibilidades de rediseño que tiene el sistema distribuido TCHSM, la implementación de una biblioteca necesaria para ese rediseño y un acercamiento a la verificación de las propiedades que este sistema dice tener.

Una posibilidad para la eliminación de estos puntos de falla críticos, requiere la implementación de una biblioteca de criptografía umbral que contenga uno de los protocolos descritos en [37] desarrollado en el lenguaje de programación C para ser utilizada por el sistema original.

Además, el protocolo criptográfico originalmente implementado en TCHSM, el primer protocolo del artículo de Shoup [37], no posee la demostración matemática de la seguridad del sistema en las condiciones que este funciona. Esta demostración sí existe para otro protocolo, el segundo protocolo del artículo de Shoup [37]. Una implementación del segundo protocolo criptográfico ayudaría a motivar la adopción del sistema. De acuerdo a lo conocido por el autor, no existe una implementación pública de este protocolo. Una descripción de alto nivel de ambos protocolos de Shoup [37] es descrita más adelante, en la sección de antecedentes.

Esto trae la oportunidad de implementar una biblioteca criptográfica que provea las primitivas de este protocolo, lo que conlleva todos los desafíos que bibliotecas de este tipo tienen: seguridad, eficiencia y confiabilidad.

Por otro lado, el haber encontrado los puntos de falla críticos dentro del sistema distribuido trae consigo el cuestionamiento a la verificabilidad de las propiedades que dicen tener tales sistemas. Dichas propiedades pueden ser estudiadas de manera rigurosa mediante la utilización de herramientas del ámbito de métodos formales.

1.4. Estructura de la tesis

El desarrollo de esta tesis se divide en cinco capítulos: El primero aborda los antecedentes de criptrafía umbral necesarios para entender las siguientes secciones, además de mostrar el trabajo relacionado.

El segundo muestras las distintas alternativas de diseño que se tienen para la eliminación de los puntos críticos de falla en el sistema TCHSM. En el se describe el rediseño con el que queda el sistema. El aporte esperado de este capítulo es la descripción de la ingeniería en sistemas distribuidos necesario para un estudio detallado de este tipo de sistemas.

En el tercero capítulo estudia una implementación de una biblioteca criptográfica necesaria para la eliminación de los puntos críticos de falla, los detalles de diseño que implica la ingeniería de criptografía, consideraciones generales, el detalle de los algoritmos y la manera en que se realiza testing en esta recién creada biblioteca. El aporte esperado de este capítulo es la descripción precisa de los algoritmos necesarios para la implementación del protocolo descrito en [37].

En el cuarto capítulo se explora la posibilidad de realizar verificación formal del sistema distribuido y se propone una nueva manera de realizarlo por medio de la utilización de herramientas de métodos formales de sistemas concurrentes. El aporte esperado es la propuesta de una novel manera de realizar verificación formal de sistemas distribuidos.

En el quinto capítulo se entregan finalmente las conclusiones del presente trabajo.

Capítulo 2

Antecedentes y trabajo relacionado

En esta sección se muestran tanto los antecedentes de criptografía umbral necesarios para el entendimiento del resto del trabajo, así como también trabajo relacionado a esta tesis.

2.1. Antecedentes de criptografía umbral

Un esquema de criptografía umbral k de n es un protocolo que permite a cualquier subconjunto de k nodos del conjunto total de n nodos para generar una firma, pero no permite la generación de una firma válida si menos de k participantes trabajan.

En este tipo de esquemas, una llave privada es dividida en n key shares (KS), es decir una llave por nodo. Los participantes utilizan cada key share para firmar el documento, generando cada uno su respectivo signature share. Cada signature share puede verificarse independientemente. La firma del documento se construye en un proceso que necesita k signature shares.

En la figura 2.1 se muestra un esquema de un proceso de firmado bajo criptografía umbral 3 de 5. El icono de la esquina inferior derecha representa a un usuario entregando un documento al signature dealer (nodo administrador del sistema distribuido), el cual envía a cada uno de los nodos. Cada nodo que recibe el documento lo firma parcialmente con su key share. Finalmente, el signature dealer une las firmas parciales –siempre que sean 3 o más–, y entrega la firma del documento al usuario. En la figura, los nodos N1, N2 y N4 representan nodos activos, es decir que son capaces de responder las solicitudes del signature dealer. En cambio, los nodos N3 y N5 representan nodos inactivos, es decir aquellos que por cualquier motivo, no son capaces de responder las solicitudes del signature dealer.

En un esquema tradicional de llave pública se tiene un par de llaves (PK, SK) –llaves pública y privada respectivamente–. En un esquema de criptografía umbral se define un par (PK, SS), en donde $SS = \{KS_{\mathrm{i}}\}_{\mathrm{i}=1}^{n}$ es el conjunto de *key shares* válidos.

En este tipo de esquemas, las firmas se verifican utilizando la llave pública PK, de la

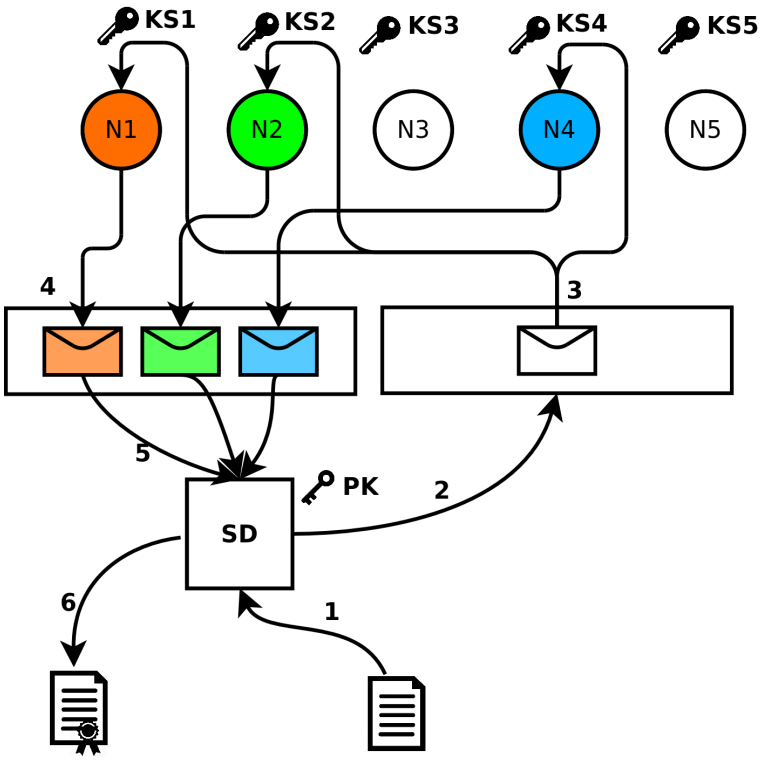


Figura 2.1: Esquema de un proceso de firmado bajo criptografía umbral k de n.

misma manera que en un esquema tradicional de llave pública. Gracias a esto, este esquema es completamente transparente para los usuarios finales [34, 37].

2.1.1. Firmas RSA utilizando criptografía umbral

Un caso particular de criptografía umbral es el descrito en el artículo Practical Threshold Signatures de Shoup [37]. En ese artículo se describen dos protocolos para generación de firmas RSA estándar mediante la utilización de criptografía umbral. De este modo, se puede dividir la responsabilidad del almacenamiento de la llave privada en varios *Key Shares*. Estos *Key Shares* son utilizados en los protocolos para la generación de *Signature Shares*, los cuales mediante un algorítmo público se pueden utilizar para generar una firma RSA estándar.

Una propiedad importante que deben cumplir los protocolos de firmado es la propiedad de no falsificabilidad. Esta propiedad se cumple cuando no es posible generar una firma válida para un mensaje por alguien que no tiene la llave secreta.

En un protocolo de firmado utilizando criptografía umbral k de n, esta propiedad se cumple cuando:

- Cualquier subconjunto con al menos k nodos no corruptos permite generar una firma válida.
- Cualquier subconjunto con k o más nodos corruptos no puede generar una firma válida.

Los dos protocolos presentados en el artículo son similares. Sin embargo, el análisis de seguridad del primer protocolo demuestra que el sistema es seguro cuando $k = t+1$, donde k es el número necesario de *Signature Shares* para generar una firma, y t es la cantidad de nodos participantes del protocolo que están corruptos. Esta limitante significa que la propiedad de no falsificabilidad se cumple sólo si ningún nodo no corrupto participa del proceso de firmado. La demostración de seguridad se hace utilizando el supuesto RSA [35].

En el caso del segundo protocolo, el análisis de seguridad demuestra que el sistema es seguro cuando $k \geq t + 1$. Esto significa que la propiedad de no falsificabilidad se cumple aún cuando nodos corruptos en conjunto con nodos no corruptos participan del proceso de firmado. En este caso, la demostración de seguridad se hace utilizando tanto el supuesto RSA [35] como el supuesto DDH [8], es decir, se hace bajo un supuesto más fuerte que el primer protocolo. Aún así, si sólo se utiliza el supuesto RSA, el segundo protocolo sigue siendo seguro para el caso $k = t + 1$.

Entonces, ¿por qué Shoup describe los dos protocolos?

La respuesta es más bien sencilla. Para la demostración del teorema de seguridad del segundo protocolo se necesita utilizar como lema el teorema de seguridad del primer protocolo.

2.2. Trabajo relacionado

La criptografía umbral ha sido utilizada en protocolos de red para ayudar en la seguridad de varios sistemas distribuidos sobre redes inalámbricas ad-hoc (red inalámbrica descentralizada, donde cada nodo participante realiza enrutamiento de paquetes), y ha sido propuesto para su utilización en la administración de llaves para DNSSEC. En la siguiente sección se discute de diversos tópicos en donde se utiliza este tipo de esquemas.

2.2.1. Redes inalámbricas y móviles ad-hoc

Crépeau y Davis [13] proponen un esquema de revocación de certificados utilizando distintos protocolos criptográficos. En el artículo afirman que en redes ad-hoc utilizar llaves simétricas no brinda la seguridad necesaria para tener una comunicación secreta: existe una mayor probabilidad de que la llave compartida sea comprometida y si solo un nodo es comprometido la red entera es comprometida. Además, utilizar llaves simétricas tiene problemas de escalabilidad, dado que todos los nodos tienen que compartir el la misma clave. Para solucionar estos problemas proponen un sistema que utiliza criptografía umbral, lo que provee seguridad y flexibilidad a costa de rendimiento. Como estos esquemas son computacionalmente intensivos (realizan una mayor cantidad de exponenciaciones modulares que otros protocolos), su rendimiento es pobre en equipos de bajas prestaciones. Esto es un punto a tener en cuenta para efectos de esta tesis, si es que se enfoca demasiado en el bajo costo de la solución. Por otro lado, requiere de cooperación no-egoísta, es decir, los nodos deben estar disponibles cuando el sistema lo requiera, lo que contradice las políticas de conservación de

energía de los aparatos que suelen utilizar redes ad-hoc, pero no dificulta la utilización de esta tecnología en esta tesis.

En el artículo de Yi y Kravets [48], se utiliza criptografía umbral para crear una autoridad certificadora (CA) distribuida, utilizando nodos ad-hoc. Una autoridad certificadora provee 4 servicios principalmente: emisión de certificados, renovación de certificados, revocación de certificados y un servicio de directorio de certificados, en donde un certificado está directamente relacionado con una llave pública. En este trabajo se desarrollan los conceptos de tolerancia a fallos, vulnerabilidad y disponibilidad. Para que el sistema propuesto sea tolerante a fallas, tenga bajo nivel de vulnerabilidad y alta disponibilidad, los autores utilizan criptografía umbral. Un contrapunto interesante del artículo es el análisis que realizan a un sistema en el cual se le asigna el rol de CA a un nodo particular. Ahí se afirma que un sistema con esta configuración no es tolerante a fallas y es altamente vulnerable.

En el artículo de Wu et al. [46] también proponen un sistema para crear una CA segura y eficiente en redes móviles ad-hoc. Para ello se vale de un esquema de criptografía umbral, una serie de protocolos de actualización de *key shares*, de inclusión de nuevos servidores y generación de certificados. En el artículo se evalúa la eficiencia del sistema mediante una simulación.

En el trabajo de Basile et al. [6] utilizan criptografía umbral para garantizar la integridad de mensajes enviados entre nodos de una red inalámbrica ad-hoc. Para ello desarrollan el concepto de consistencia del círculo interno, es decir, si se tiene un nodo x, todos los nodos locales al nodo x –aquellos que están a 1 salto de distancia– verifican la posibilidad de que el nodo x esté comprometido. Esta es una interesante apuesta de este trabajo, ya que para no afectar el rendimiento de los nodos componentes de la red, los algoritmos más demandantes sólo se ejecutan en la proximidad de el nodo a verificar. Este trabajo cubre un área dejada de lado por los trabajos previos, que es la protección ante ataques internos en redes ad-hoc.

En el trabajo de Luo et al. [24] se desarrolla un sistema para controlar el acceso a una red inalámbrica ad-hoc. En él, ningún nodo monopoliza el control de acceso o es completamente confiable, y cada nodo verifica que sus vecinos tengan un comportamiento adecuado. Para esto, se utiliza un esquema de criptografía umbral que se adapta a la necesidad de un numero cambiante de nodos. En este esquema, cada nodo que quiera participar de la red requiere que su solicitud haya sido certificada por sus nodos vecinos, lo que permite establecer una conexión. Estas conexiones tienen un tiempo de expiración que, cuando se cumple, se le solicita a los nodos vecinos que colaborativamente certifiquen una nueva conexión del nodo a la red, mediante el análisis el comportamiento previo del nodo. En este caso, para probar el rendimiento realizan simulaciones utilizando el software *ns-2*.

Los últimos dos trabajos pueden servir de guía sobre cómo actuar frente a la entrada y salida dinámica de nodos.

2.2.2. Administración de llaves de DNS

Kozic et al. [22] analizaron la manera en que se administran las llaves privadas para DNSSEC, evaluando disponibilidad, seguridad, prevención en la pérdida de datos e interoperabilidad. Ellos estudiaron en este caso 3 herramientas: una libre, OpenDNSSEC, y dos comerciales, Infoblox y Bluecat. De ellas concluyeron que la seguridad que proveen las herramientas comerciales es cuestionable por almacenar la llave privada de manera insegura en el servidor DNS. OpenDNSSEC, por utilizar la API PKCS #11, puede almacenar sus secretos en dispositivos certificados de manera segura.

Yang et al. [47], declaran que DNSSEC es quizás el primer intento de aplicar criptografía a un sistema a la escala de todo Internet. Es por esto que varias lecciones han tenido que ser aprendidas a la fuerza: un sistema de estas características debe ser diseñado con la escalabilidad en mente, pensando a la heterogeneidad de Internet. Por esos motivos, la adopción de DNSSEC requiere de la utilización de herramientas de administración de firmas de registros DNS que cumplan esas características.

Wang et al. [43] escribieron un artículo en el se propone por primera vez un esquema verdaderamente distribuido para la administración de llaves en DNSSEC. En él describen que un principio de seguridad primordial para DNSSEC es la separación de roles. Los roles de administración de las zonas de DNS y de administración del servidor de nombres están diferenciados. Un servidor DNS es solamente un lugar para almacenar registros DNS previamente firmados y es el administrador de la zona quien debe encargarse de firmar los registros. Cuando una zona requiere una actualización, el administrador de la zona deberá firmar los registros de manera *off-line*. La idea de mantener la llave privada de la zona *off-line* se basa en el supuesto de que las zonas DNS son relativamente estáticas. Sin embargo, se ha establecido que la actualización dinámica de zonas DNS puede ser útil en varias circunstancias. Por ejemplo, puede permitir a instituciones cambiar el equipo que responde a un cierto nombre sin esperar por un lento procesamiento manual.

Para poder utilizar DNSSEC sobre un esquema de actualización dinámica de registros DNS, se hace necesario tener acceso *on-line* a la llave, lo que en cierta manera se contradice con la separación de roles. Además, esto hace que la máquina que almacene esta llave se transforma en un único punto de falla, contradiciendo el diseño original de DNS. En el artículo se propone un sistema que goza de la propiedad de separación de roles y actualización dinámica de zona. Para ello se valen de la utilización de un conjunto de servidores de seguridad de la zona, los cuales utilizan criptografía umbral para el manejo seguro e independiente del firmado de registros DNS. Sin embargo, este sistema tiene una configuración compleja y es distinto a las soluciones actuales como las que proveen BIND o OpenDNSSEC. Esto último sirve como motivación para que un sistema como el propuesto gane relevancia.

Capítulo 3

Puntos críticos de falla en el sistema y su eliminación

En este capítulo se estudia la implementación del sistema Threshold Cryptography HSM: se analiza la existencia de puntos críticos en este sistema, algunas alternativas de diseño para quitar la presencia de estos y un nuevo diseño del sistema que quita los puntos críticos sin perder funcionalidad.

3.1. Puntos críticos de falla

La arquitectura de TCHSM incluye los siguientes nodos:

- Dos instancias de RabbitMQ[1] (RMQ), software libre encargado de proveer un sistema de mensajería entre procesos (ya sean remotos o locales). Estas instancias se encargan de concentrar la comunicación entre cada uno de los nodos.
- El nodo pkcs11, el cual incluye tanto la aplicación (APP) que quiere utilizar el sistema, como la biblioteca Libpkcs11, desarrollada en el lenguaje de programación C, que implementa la API PKCS #11. Esta biblioteca se encarga de generar las peticiones que hace la aplicación para enviarlas al Trusted Responser.
- Un nodo Trusted Responser, que se encarga de recibir y traducir las peticiones para entregárselas al Trusted Dealer.
- Un nodo Trusted Dealer que realiza la coordinación de los nodos firmadores. Este nodo originalmente está escrito en el lenguaje de programación Java. La funcionalidad de este nodo corresponde a la generación de key shares, al envío de estas, al envío de solicitudes de firmado a los nodos con el respectivo documento a firmar, y la unión de los signature shares generados por cada uno de los nodos.
- Múltiples nodos firmadores, que realizan el trabajo de generación de firmas digitales.

Dentro de esta arquitectura se define como subsistema de criptografía umbral a el nodo

[1] `https://www.rabbitmq.com`

Trusted Dealer, cada uno de los firmadores, y el nodo RabbitMQ que sirve para la comunicación entre los anteriores.

En esta arquitectura se encuentran cuatro puntos críticos de falla: las dos instancias de RabbitMQ, el nodo Trusted Responser y el nodo Trusted Dealer. En la figura 3.1 se muestra cada uno de estos puntos: en la caja de bordes rojos se encuentran cada uno de los puntos críticos de falla. Cada uno de estos implementa funcionalidad crítica del sistema distribuido. Las flechas indican comunicación remota. La linea entre Trusted Responser y el Trusted Dealer indica comunicación directa y local entre ambos nodos.

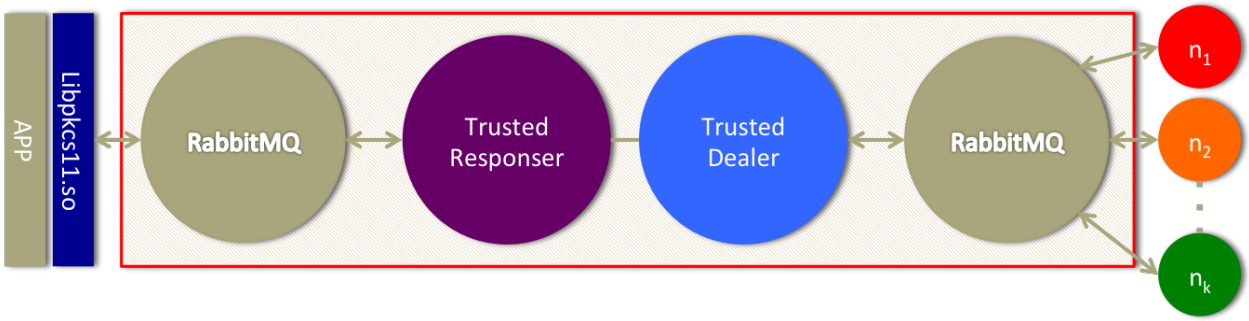


Figura 3.1: Diseño original del sistema, incluyendo la interfaz para ser utilizado por cualquier aplicación (APP).

La primera instancia de RabbitMQ se encarga de la distribución de los mensajes que la biblioteca Libpkcs11 envía a Trusted Responser. Trusted Responser recibe estas solicitudes y maneja el estado de ellas. Cuando se requiere, se envía una solicitud al subsistema de criptografía umbral. En el caso en que se envía una solicitud al subsistema de criptografía umbral, el módulo Trusted Dealer recibe la solicitud y crea un nuevo mensaje que esta vez será enviado a los nodos a través de segunda instancia de RabbitMQ. Cabe destacar que la funcionalidad necesaria que provee RabbitMQ puede ser cubierta por una sola instancia de este, pero en términos lógicos proveen funcionalidad distinta desde el punto de vista del sistema.

Si se desea aumentar la disponibilidad del sistema, usualmente se piensa en multiplexar cada uno de los nodos, es decir, tener varios nodos capaces de responder la misma solicitud de manera transparente al solicitante.

Los puntos de falla crítica descritos previamente son difíciles de multiplexar: requiere de configuración avanzada –la cual será descrita posteriormente– en el caso de las instancias de RabbitMQ y de nueva funcionalidad en los casos del módulo Trusted Responser y del módulo Trusted Dealer.

Todo lo anterior deja al sistema con múltiples puntos de falla críticos, quitando una de las propiedades que promete cualquier sistema que utilice criptografía umbral para firmar: la tolerancia a fallos.

3.2. Posibilidades de acción

Para enfrentar esta situación se consideran dos tipos de nodos, las instancias de RabbitMQ y los módulos específicos del sistema (es decir, Trusted Dealer y Trusted Responser). Los dos tipos de nodos requieren un enfoque distinto para que dejen de ser un punto de falla crítico.

3.2.1. Sistema de mensajería

En el diseño original se tiene a RabbitMQ como concentrador de mensajería. A este tipo de software se les conoce como Message-Oriented Middleware[2] (MOM) y se encargan de proveer una solución modular al intercambio confiable de mensajes entre nodos en un sistema distribuido. Utilizar este tipo de software agrega una componente arquitectural al sistema, componente cuya labor es la distribución de mensajes. A esta componente se le conoce como Message Broker. Ejemplos de MOMs son el ya mencionado RabbitMQ o Apache QPID[3].

RabbitMQ es un MOM que permite utilizar el Message Broker en modo *clustering*[4]. En este modo se permite distribuir el trabajo y la responsabilidad de entregar un mensaje de manera confiable entre varios Message Broker, distribuyendo así la carga del sistema, aumentando la disponibilidad y mejorando la tolerancia a fallas. En la figura 3.2 se muestra esta posibilidad: las flechas con lineas continuas simbolizan comunicación activa, y las flechas con lineas discontinuas simbolizan comunicación en espera. Esta opción permite mitigar el problema de tener un punto de falla crítico a costa de dificultar la configuración de un sistema cuyo desarrollo no es parte de este trabajo.

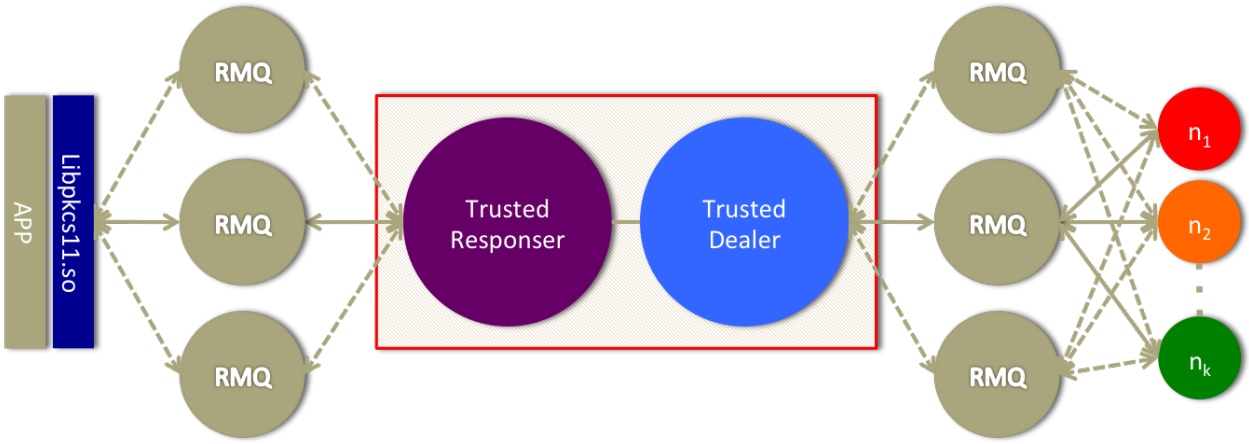


Figura 3.2: Posible diseño en el cual se muestran dos clusters de RabbitMQ, cada uno con tres nodos.

Por otro lado existen soluciones de intercambio confiable de mensajería que no requieren la utilización de un Message Broker, tan sólo la utilización de una biblioteca que implementa un protocolo de intercambio confiable de mensajes de largo arbitrario. A este tipo de soluciones

[2] https://en.wikipedia.org/wiki/Message-oriented_middleware
[3] https://qpid.apache.org/
[4] https://www.rabbitmq.com/clustering.htm

se les conoce como *Brokerless Message Queue Libraries*. Ejemplos de estas bibliotecas son ZeroMQ[5] o nanomsg[6], de las cuales ZeroMQ tiene un mayor respaldo de la industria, mayor desarrollo científico asociado y una comunidad más grande. Utilizar una biblioteca de este estilo permite mitigar la ocurrencia de puntos críticos de falla por medio del traslado de la funcionalidad de un nodo Message Broker hacia los nodos adyacentes a este, tal como se muestra en la figura 3.3, eliminando así la necesidad de una componente arquitectural extra para el intercambio de mensajes. Esta manera de eliminar puntos críticos es un enfoque clave usado en otras partes de este trabajo.

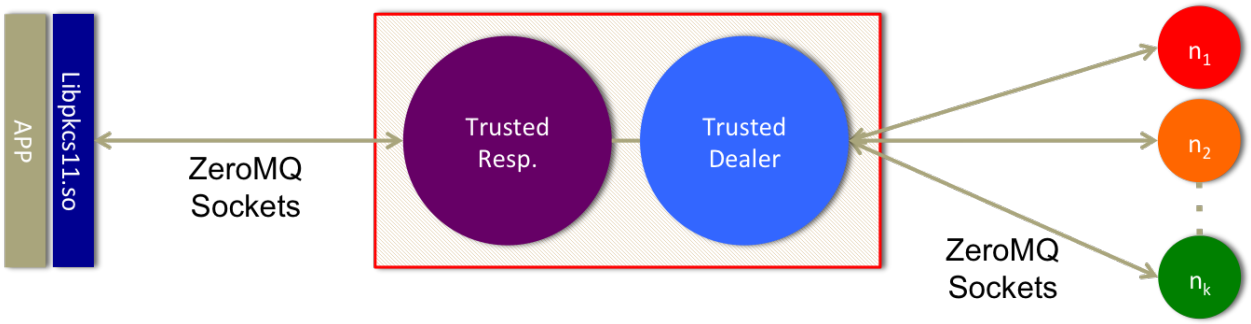


Figura 3.3: Posible diseño en el cual se cambia la funcionalidad de RabbitMQ por la biblioteca ZeroMQ.

Una tercera alternativa es utilizar el protocolo HTTP (*Hypertext Transfer Protocol*) por medio de crear una API REST (*Representational state transfer*) en cada uno de los nodos que requieren de comunicación remota. En el diseño original del sistema Threshold Cryptography Backend, se propuso utilizar una una API de ese estilo para la comunicación entre las componentes, pero se utilizaría RabbitMQ para la comunicación interna de cada componente. Desarrollar una API REST tiene la particularidad de ser un estándar de la industria, con conocida escalabilidad y seguridad mediante la utilización de TLS (*Transport Layer Security*). Sin embargo, obliga a utilizar los verbos del protocolo HTTP (PUT, GET, CREATE, DELETE, etc.) y a seguir una serie de restricciones para modelar la comunicación entre nodos, así como tener una comunicación asimétrica entre los nodos. Además, para realizar este enfoque y obtener los beneficios asociados se requiere una capa extra de configuración en cada uno de los nodos: un servidor HTTP.

3.2.2. Módulos específicos del sistema: Trusted Dealer y Trusted Responser

Tanto el Trusted Dealer, como el Trusted Responser son puntos de falla críticos. Para estudiar estos puntos de falla de manera más sencilla, se pueden unir en uno solo. La comunicación entre ambos nodos es posible realizarla directamente como llamada a los métodos Java que correspondan, por lo que no implica un trabajo mayor. A este nodo unificado se le llamará de aquí en adelante simplemente nodo maestro (NM).

[5] `http://zeromq.org`
[6] `http://nanomsg.org`

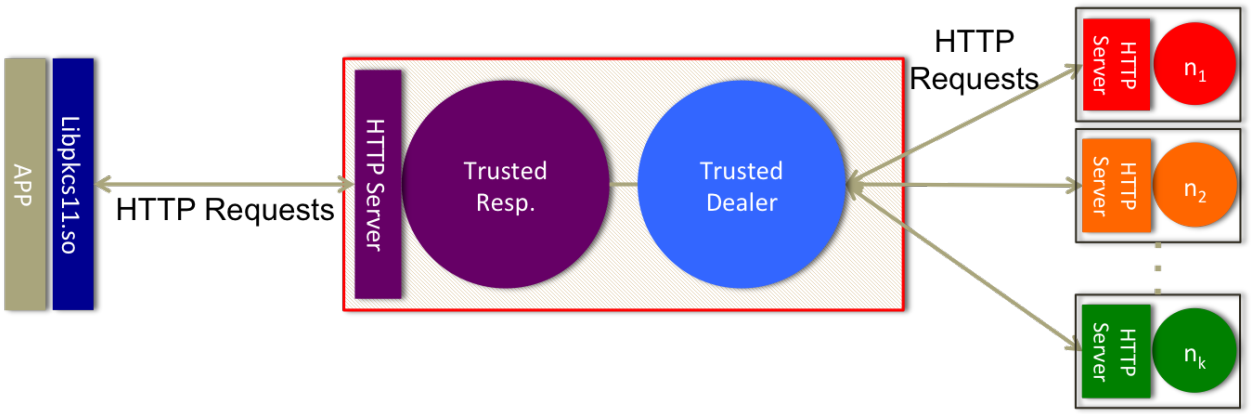


Figura 3.4: Posible diseño en el cual se cambia la funcionalidad de RabbitMQ por una API REST, con la ayuda de servidores y clientes HTTP.

Además consideremos lo siguiente: el nodo maestro originalmente está escrito en el lenguaje de programación Java, incluye funcionalidad criptográfica avanzada y mantiene estados dentro de su ejecución. Además, la biblioteca Libpkcs11 está programada en el lenguaje de programación C. Por otro lado, la solución no debe agregarle costos financieros extras a quien despliegue este sistema.

A continuación se presentan tres posibilidades para mitigar los problemas relacionados a un único punto de falla, utilizando técnicas conocidas de sistemas distribuidos.

La primera posibilidad consiste en implementar replicación de varios nodos maestro (Figura 3.5). En este esquema existe un nodo activo y varios nodos que esperan a que el nodo activo falle. En caso de que haya una falla, se realiza una migración de nodo activo de manera transparente, de forma que la comunicación ahora se realiza con alguno de los nodos replicados.

Según Tanenbaum y Van Steen [39] hay dos razones principales para la utilización de replicación de datos: confiabilidad y rendimiento. Confiabilidad porque es posible continuar trabajando, en caso de la falla de un nodo activo. En cuanto al rendimiento, la replicación es importante en sistemas distribuidos que necesitan escalar a áreas geográficas cada vez más grandes.

Por otro lado la replicación trae problemas. Su principal costo es la dificultad de mantener la consistencia de datos entre las instancias, lo que implica una complejidad extra en el sistema que requiere una evaluación crítica. Por el solo hecho de mantener el estado distribuido se agrega latencia extra. Además, en el momento de una migración, el tiempo de respuesta del sistema crece considerablemente.

Una segunda alternativa es que los nodos maestro no almacenen estado, traspasando esta lógica a algún sistema de bases de datos y utilizar conocidos mecanismos de tolerancia a fallas que estos sistemas tienen (Figura 3.6). En este caso también se debe realizar una migración al momento que el nodo activo falle, pero la migración es mucho más sencilla al no guardar estado.

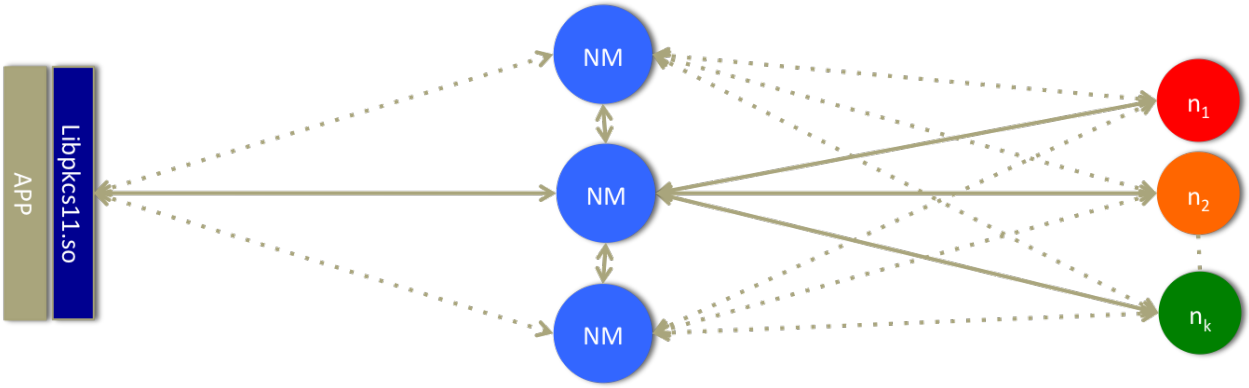


Figura 3.5: Posible diseño en donde se utilizan replicación y migración entre varios nodos maestro.

Sin embargo, se está confiando en los mecanismos que provee el sistema de bases de datos, los cuales por lo general implican replicación y migración del estado de la base de datos. Esto nos lleva a que se tienen las mismas ventajas y desventajas de la alternativa anterior, con las ventajas de delegar la implementación de esta funcionalidad a una herramienta externas a costa de una mayor complejidad en la configuración.

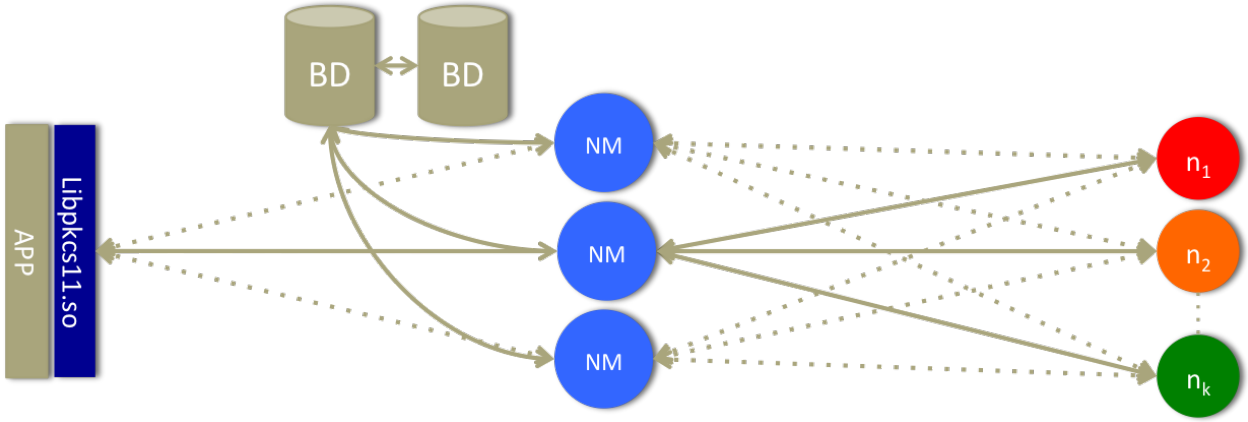


Figura 3.6: Posible diseño en donde se utilizan bases de datos para guardar el estado del programa.

Por último, la tercera alternativa consiste en trasladar la funcionalidad del nodo maestro a los nodos adyacentes, es decir tanto a la biblioteca Libpkcs11 como a los nodos según corresponda (figura 3.7).

La funcionalidad del Trusted Dealer tiene oportunidad de ser trasladada al espacio de Libpkcs11, ya que es realmente ahí donde se necesita almacenar el estado. Esta solución requiere implementar la funcionalidad criptográfica en el lenguaje de programación C. Esta alternativa conlleva la posibilidad perder efectivamente los puntos críticos de falla, en contraposición de las otras alternativas que son mitigaciones de esto.

Implementar esta solución requiere reimplementar todos los mecanismos originales que posee el nodo maestro, por lo que este costo debe ser considerado al diseñar el nuevo sistema. Sin embargo, deben considerarse también efectos secundarios importantes:

1. Al quitar un nodo dentro del sistema la latencia general del sistema disminuye.
2. Al traspasar funcionalidad de Java a C, si se realiza con cuidado, permite utilizar mecanismos de hardware para el manejo de funcionalidad criptográfica.

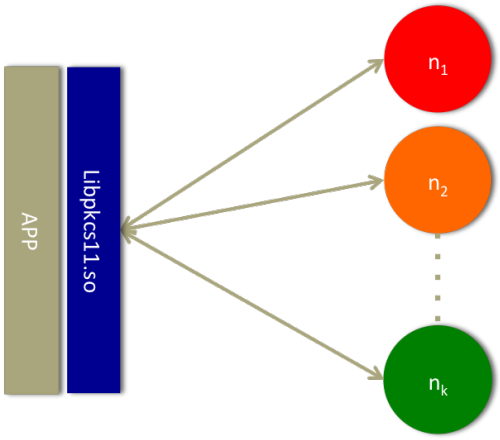


Figura 3.7: Posible diseño trasladando toda la funcionalidad del nodo maestro a los nodos extremos.

3.3. Nuevo Diseño

El análisis previo deja tres alternativas para el sistema de mensajería y tres alternativas para el nodo maestro, lo que nos deja con nueve alternativas totales. Considerando cada uno de los aspectos, tanto favorables como desfavorables, se llega a la conclusión que una alternativa que utilice ZeroMQ como mecanismo de intercambio de mensajes y el traslado de funcionalidad del nodo maestro al espacio de Libpkcs11 permite tener la propiedad de tolerancia a fallas, además de mejorar aspectos como la latencia, el desempeño general y una notable simplificación del sistema.

La configuración e instalación de los nodos, incluyendo que nodos son confiables se debe realizar en la configuración de la biblioteca Libpkcs11, mediante la utilización de mecanismos de autenticación y encriptación utilizando criptografía de llave pública.

Al traspasar toda la funcionalidad del nodo maestro a Libpkcs11, las operaciones de generación y distribución de llaves –críticas en materia de seguridad– quedan directamente en manos de la aplicación y su usuario, siguiendo exactamente el mismo modelo de seguridad que se tienen los HSM. Más aún: si cada nodo firmador está en un dispositivo físico distinto, al distribuir las llaves, estás quedan almacenadas en dispositivos físicos distintos. De este modo se elimina el punto de falla crítico.

Para llevar a cabo esta alternativa, una posibilidad es implementar una biblioteca criptográfica en el lenguaje de programación C que implemente modularmente la funcionalidad requerida por Libpkcs11. La descripción de como se implementa esta biblioteca, como se realiza testing y verificación formal tanto de la biblioteca como de un modelo del sistema distribuido, serán descritos en el siguiente capítulo.

El nuevo diseño permite cambiar la configuración de nodos por parte de las aplicaciones, e incluso permite a varias aplicaciones utilizar distintos conjuntos de nodos en donde se comparten algunos de éstos. Como ejemplo, en la figura 3.8 se tienen dos aplicaciones utilizando el mismo conjunto de nodos. En la figura 3.9 se tienen 3 aplicaciones, las cuales utilizan 3 conjuntos distintos de nodos. De ellas, 2 aplicaciones utilizan 3 nodos, la otra utiliza 4 nodos. Cada uno de estos conjuntos comparten algunos nodos con otra configuración. Ambas configuraciones son transparentes para las aplicaciones.

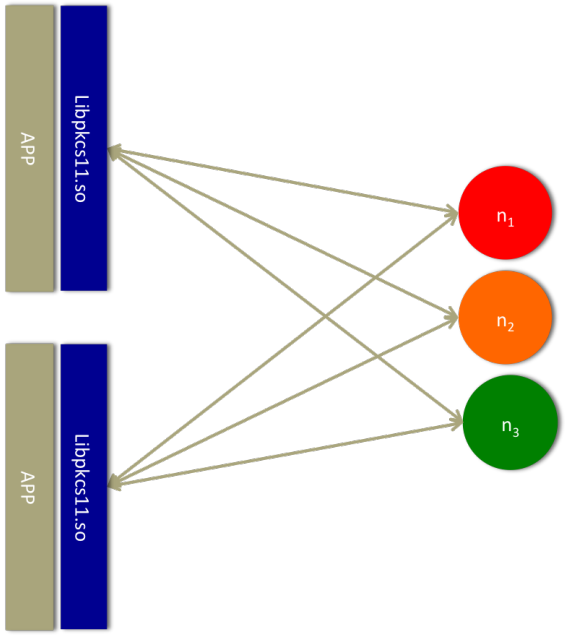


Figura 3.8: Dos aplicaciones con el mismo conjunto de nodos.

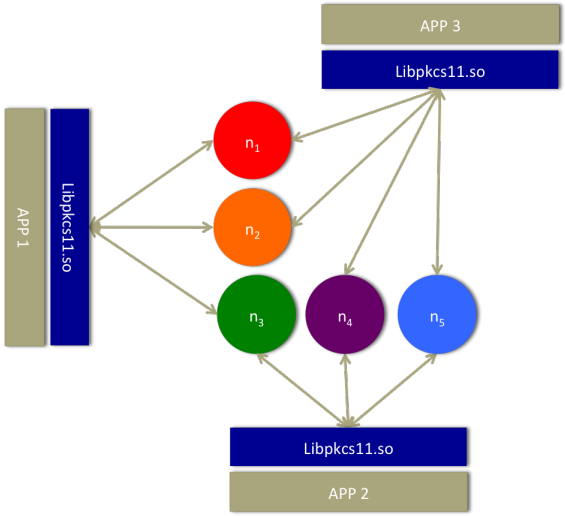


Figura 3.9: Tres aplicaciones con distintos conjuntos de nodos.

Capítulo 4

Implementación de la biblioteca de criptografía umbral

En este capítulo se muestra el estudio y análisis de la implementación de una biblioteca que provee primitivas criptográficas para ser usada en el esquema de firmado umbral RSA (del inglés *RSA Threshold Signature Scheme*) descrito por Shoup en [37]. En ese artículo se da una descripción general de 2 protocolos de criptografía umbral. El primero de estos, era el que estaba implementado originalmente en Java. De acuerdo a lo conocido por el autor, el segundo no tiene implementaciones públicas disponibles

Las particulares características del lenguaje de programación C obligan a tener consideraciones especiales al programar este tipo de software: el manejo manual de la memoria, la biblioteca estándar minimal, el uso de bibliotecas desarrolladas por la comunidad para resolver algunas necesidades, etc.

Por otro lado, la implementación de primitivas criptográficas siempre conlleva un estudio distinto al análisis teórico de las primitivas. A este estudio se le conoce como Ingeniería de Criptografía (del inglés *Cryptography Engineering*), y tiene relación con la elección de parámetros, el estudio de los resultados entregados, las definiciones estrictas de la memoria entregada como resultado, entre otras consideraciones de seguridad.

4.1. Discusión sobre la seguridad de los protocolos de criptografía umbral

Para el análisis de seguridad de un sistema de criptografía umbral se necesitan 3 parámetros clave: l, k, y t. El parámetro l es la cantidad total de nodos en el sistema. El parámetro k es la cantidad de nodos necesarios para generar una firma, es decir, el parámetro del umbral de firmado. Finalmente, el parámetro t es la cantidad de nodos corruptos manejados por un adversario que quiere romper el sistema.

La robustez del sistema se muestra en la capacidad de generar una firma RSA válida si se tienen al menos k nodos válidos. La no falsificabilidad del sistema es la incapacidad de que un adversario pueda generar una firma de un documento a través de nodos corruptos, si es que no fue enviado como solicitud de firmado a al menos $k - t$ nodos no corruptos.

Como fue mencionado anteriormente, en el artículo de Shoup se da una descripción de dos protocolos para la generación de firmas por umbral. El primero de ellos posee un análisis de seguridad que demuestra que es robusto y no falsificable cuando $k = t + 1$, asumiendo que el problema RSA (factorizar un número entero grande en sus componentes primas) es computacionalmente difícil. El segundo tiene un análisis de seguridad que demuestra que el sistema es robusto y no falsificable cuando $k >= t + 1$, asumiendo tanto que el problema RSA como que el problema DDH (problema decisional de Diffie-Hellman, problema con un supuesto más fuerte que el problema RSA) son computacionalmente difíciles, y que además, sigue cumpliendo estas propiedades cuando $k = t + 1$ tan sólo asumiendo la dificultad del problema RSA.

¿Qué significa esto? Significa que en el primer protocolo, la no falsificabilidad se cumple sólo si ningún nodo no corrupto recibe la solicitud de firmar un documento falso, mientras que en el segundo protocolo la no falsificabilidad se cumple siempre y cuando hayan más nodos confiables que corruptos.

Estas características hacen deseable para el sistema de HSM distribuido el segundo protocolo, y se decide desarrollarlo a pesar de no contar con alguna implementación previa.

4.2. Consideraciones generales

En esta sección se describen los terminos generales sobres los cuales se establece el diseño de la solución. Esto es, los aspectos legales de distribución, el uso de bibliotecas existentes y las decisiones generales de diseño.

4.2.1. Licencia de distribución

Para ayudar a la adopción de la biblioteca, se decide que ésta sea software libre. Dentro de las opciones, se busca un licenciamiento que permita que tanto software libre como privativo pueda utiliza la biblioteca con total libertad, sin ser necesaria la públicación de modificaciones que se le realicen, siempre y cuando se mencione la autoría del código original. Por estas razones, se escoge una licencia de tipo MIT. De este modo se permite su uso en aplicaciones tanto privativas como libres, así como mantiene el código original disponible para su estudio. Se debe tener en cuenta este licenciamiento para la elección de bibliotecas auxiliares.

4.2.2. Bibliotecas auxiliares

En criptografía se evita a como de lugar reimplementar código que implemente alguna funcionalidad, de modo de evitar cualquier tipo de vulnerabilidad y enfocar los estudios de estas en tan sólo unas pocas bibliotecas. En [36], Mike Shema advierte que hay que reutilizar el código, no reimplementarlo, ya que "la criptografía es el mejor ejemplo del peligro que significa reimplementar un algoritmo desde cero".

Sin embargo, de acuerdo a lo conocido por el autor, no existe una implementación de la funcionalidad criptográfica que se necesita tal que cumple los requisitos descritos.

Las primitivas criptográficas descritas en el artículo de Shoup necesita de el manejo de números enteros de largo arbitrario, así como también se necesita utilizar una función de hash también arbitraria. Dado que ni el lenguaje ni su libreria estándar ofrecen esta funcionalidad se debe buscar la manera de obtenerla. Como fue argumentado previamente, es mejor buscar la funcionalidad en alguna biblioteca existente que entregue las características que se necesitan. Las bibliotecas escogidas deben ser pequeñas y autocontenidas, esperando además que estén enfocadas en realizar sólo la tarea buscada y no ser una biblioteca criptográfica en general. Así se espera evitar el overhead y la poca libertad de acción que implica utilizar una de estas.

Al buscar bibliotecas de manejo de números enteros que provean una API para C, se encuentran varias alternativas. Dentro estas destacan GMP [15], MPIR [16], OpenSSL [42] y LibTomMath [41] como las más utilizadas y mantenidas. Abusharekh y Gaj [1] realizaron un análisis de las distintas alternativas de bibliotecas de números enteros de largo arbitrario, enfocados en que serán utilizadas en la implementación de algoritmos criptográficos. Esto significa que realizaron pruebas de rendimiento y seguridad para las operaciones necesarias en criptografía de llave públicas, las mismas que serán ocupadas para la biblioteca estudiada en este capitulo. Al analizar 8 bibliotecas distintas, concluyeron:

> "... For a developer targeting operations on large integers, GMP would be the best choice in terms of performance. The trade off however, is the amount of time and effort needed for implementation, and portability. ..."

Lo cual se interpreta como que GMP puede ser la mejor opción en términos de performance, teniendo en cuenta que su dificultad radica en su uso. Por otro lado, todas las bibliotecas analizadas se distribuyen bajo licencias compatibles con la licencia MIT bajo las condiciones que serán usadas en el proyecto de esta tesis.

En cuanto a bibliotecas que provean funciones de hash se buscan aquellas que sean pequeñas, autocontenidas, estables y con un licenciamento compatible. Dentro de las alternativas están MHash [26], LibTomCrypt [40] y OpenSSL [42]. De estas alternativas, MHash es la única cuyo único objetivo es proveer funciones de hash, dejando las posibilidad de no agregar el overhead de bibliotecas criptográficas. Por lo demás, MHash también se distribuye bajo la licencia LGPL, siendo compatible con el software a desarrollar. Esto lleva a escoger esta para utilizar las funciones de hash ahí implementadas.

4.2.3. Decisiones de diseño

Para desarrollar una biblioteca criptográfica en el lenguaje de programación C, deben existir lineamientos estrictos de diseño a ser respetados, de modo de evitar por completo los errores de programación. Según los desarrolladores de OpenBSD [31], sistema operativo conocido por su énfasis en seguridad y correctitud:

> "... We are not so much looking for security holes, as we are looking for basic software bugs, and if years later someone discovers the problem used to be a security issue, and we fixed it because it was just a bug, well, all the better. ..."

De lo cual se puede entender que el énfasis tiene que ser puesto en evitar errores básicos de programación, más que en agujeros de seguridad en sí.

Abstracción de funcionalidad de la biblioteca

El lineamiento general del desarrollo del código de la biblioteca de criptografía umbral tiene como principal característica la abstracción completa de las estructuras utilizadas por el sistema. En el lenguaje de programación C esto se realiza mediante la utilización de *punteros opacos* [33]. Este mecanismo consiste en la declaración de las estructuras por separado de la definición y utilizar punteros a estas estructuras en las funciones. De este modo se crea una capa de abstracción entre el cliente de la biblioteca y el desarrollador de la biblioteca, permitiendo un uso transparente por parte del cliente ante posibles cambios por parte del desarrollador.

La principal ventaja característica es que ordena el desarrollo y obliga a seguir estrictamente las firmas declaradas en la API pública.

Uso estricto del modificador *const*

En el lenguaje de programación C, desde el estándar C99 en adelante existe el modificador de tipo de variables *const*. Este modificador previene la mutación del contenido en memoria de variables y punteros. La utilización de esta herramienta permite al compilador chequear de manera precisa los accesos a memoria. Además, con este tipo de variables el desarrollador puede definir la intención de cada variable en términos de entrada o salida.

4.3. Descripción detallada de algoritmos

En esta sección se muestra una descripción detallada de las estructuras utilizadas y de los algoritmos descritos de manera general en el artículo de Shoup [37]. Los algoritmos aquí mostrados son descripciones precisas de los algoritmos realmente implementados. Mayor detalle en la descripción y justificación de los algoritmos puede ser visto en [37].

4.3.1. Estructuras

A continuación se muestra la definición de las distintas estructuras usadas en los algoritmos. En ellas se usan *Integers* y *uint16_t* como tipos básicos. Integers son números enteros de largo arbitrario, que están representados en bajo nivel por una cadena de bytes. *uint16_t* son enteros sin signo de largo fijo de 16 bits.

En el listado 4.1 se muestra la definición de la estructura de llave pública (Public Key). Ahí, el campo n es el módulo de la llave pública, el campo m es el módulo verificador y el campo e es el exponente público.

Listado 4.1: Public Key

```
1 struct public_key {
2     Integer * n;
3     Integer * m;
4     Integer * e;
5 };
```

En el listado 4.2 se muestra la definición de la estructura que contiene información asociada a cada conjunto de Key Shares (Key Metainfo). En ella, el campo public_key contiene la llave pública asociada, el campo k contiene el parámetro umbral del conjunto de Key Shares, el campo l contiene el número total de Key Shares, el campo vk_v contiene un número verificador grupal y el campo vk_i contiene un arreglo de números verificadores. Cada uno de estos números verificadores sirve para verificar que los Signature Shares recibidos hayan sido generados por Key Shares válidos.

Listado 4.2: Key Metainfo

```
1 struct key_metainfo {
2     struct public_key * public_key;
3     uint16_t k;
4     uint16_t l;
5     Integer * vk_v;
6     Integer * vk_u;
7     Integer[] * vk_i;
8 };
```

En el listado 4.3 se muestra la definición de la estructura que almacena un Key Share. En ella, el campo s_i contiene la llave parcial en si (el valor s_i de [37]), el campo n contiene el módulo de la llave pública y el campo id contiene el índice de la llave parcial (el valor i en s_i).

Listado 4.3: Key Share

```
1 struct key_share {
2     Integer * s_i;
3     Integer * n;
4     uint16_t id;
5 };
```

Finalmente, en el listado 4.4 se muestra la definición de la estructura que almacena el resultado del proceso de firmado parcial: Signature Share. En ella el campo x_i contiene la

firma parcial en si (el valor x_i de [37]), los campos c y z contienen números para ser usados posteriormente en el proceso de verificación, y en el campo id se almacena el índice de la llave parcial que fue usada para generar x_i (el valor i en x_i).

Listado 4.4: Signature Share

```
1 struct signature_share {
2     Integer *x_i;
3     Integer *c;
4     Integer *z;
5     uint16_t id;
6 };
```

4.3.2. Algoritmos

En esta subsección se describen de manera detallada los algoritmos de generación de Key Shares, de generación de un Signature Share, de verificación de un Signature Share y de unión de Signature Shares.

Para generar los Key Shares, se escogen dos números primos al azar de igual largo –largo definido por el usuario, principal parámetro de seguridad del protocolo– tales que son *safe primes*. Un número *safe prime* es aquel número que tiene la forma $p = 2p' + 1$, donde p' es también un número primo. Para generar un *Safe Prime*, el algoritmo descrito en el listado 4.5 busca un número primo al azar y prueba si $2p + 1$ o $\frac{p-1}{2}$ son números primos también. En caso de que alguno de los dos lo sea, se devuelve el *safe prime* correspondiente. Este enfoque es el mostrado por Naccache en [29], y si bien hay otros algoritmos (como [45]) que prometen ser más eficientes, estos requieren modificar la manera de buscar número primos cualquiera y por tanto es más difícil de implementar, dejándose como trabajo futuro.

El algoritmo mostrado en el listado 4.6 se encarga de generar 2 estructuras de datos: un arreglo de Key Shares y un Key Metainfo. La estructura Key Share almacena un entero s que es el key share en si, un entero id que es el identificador asociado a el nodo que recibe el Key Share, y un entero n módulo RSA. La estructura Key Metainfo almacena un entero vk_v verificador del conjunto de llaves, un entero n módulo RSA, un entero e exponente público RSA. Para llevar a cabo la operación se ocupan tres funciones auxiliares especiales: Jacobi, RandomPoly y ExtendedGcd. Jacobi calcula el símbolo de Jacobi[1]. $RandomPoly(c, g, m)$ genera un polinomio al azar de grado g, con c cómo término independiente y con m como módulo para las operaciones de arítmetica modular. Finalmente, Extended Gcd calcula el divisor común máximo (Gcd) entre dos números, digamos x e y, así como los coeficientes a y b, tales que $a \cdot x + b \cdot y = Gcd(x, y)$.

Además se ocupa la constante $F4 = 2^k16 + 1$, el cuarto número primo de fermat. Este número se caracteriza por la rapidez con que se realizan las operaciones modulares en los sistemas modernos.

El algoritmo del listado 4.7 corresponde a la generación de un Signature Share, es decir a una firma parcial tal que si se juntan un número mínimo de ellas se pueden unir para forma

[1] `http://mathworld.wolfram.com/JacobiSymbol.html`

Listado 4.5: Generación de un número *Safe Prime*

```
1  procedure GenerateSafePrime(bitLen)
2  require 0 < bitLen
3  ensure un entero safe prime
4
5      repeat
6          p := RandomPrime(bitLen)
7          q := (p−1)/2
8          r := 2p + 1
9      until ProbablePrime(q) || ProbablePrime(r)
10
11     if ProbablePrime(q) then
12         return p
13     else
14         return r
15     endif
16
17 end procedure
```

Listado 4.6: Generación de l Key Shares con un umbral de k nodos

```
1  procedure GenerateKeyShares(bitSize, k, l)
2  require 0 < bitSize && 0 < k < l
3  ensure un arreglo con l KeyShares y un KeyMetainfo
4      primeSize := bitSize/2
5      p := GenerateSafePrime(primeSize)
6      q := GenerateSafePrime(primeSize)
7      p' := (p−1)/2
8      q' := (q−1)/2
9      n := p · q
10     m := p' · q'
11
12     # F4 es el cuarto número primo de Fermat.
13     F4 := 65537
14     if l < F4 then
15         e := F4
16     else
17         e := RandomBits(sizeof(l)+1)
18     endif
19
20     d := e^−1 mod m
21     do
22         r := RandomBits(sizeof(n))
23     while gcd(r, d) != 1
24
25     do
26         mi.vk_u := RandomBits(sizeof(n))
27     while Jacobi(mi.vk_u, n) != 1
28
29     Δ := Factorial(l)
30     P := RandomPoly(d, k-1, m)
31
32     for i := 1 to l do
33         ks_i.s_i := P(i) * Δ^−1 mod m
34         mi.vk_i := mi.vk_v^s mod n
35         ks_i.id := i
36         ks_i.n := n
37     endfor
38
39     mi.public_key.n := n
40     mi.public_key.e := e
41
42     return (ks, mi)
43 end procedure
```

una firma RSA. Este algoritmo se ejecuta en cada uno de los nodos firmantes. Para la operación se necesita de un Key Share, de un documento a firmar y de el Key Metainfo asociado al Key Share. La operación consiste en generar tanto una firma parcial del documento, como de una "prueba de correctitud" (del inglés "*proof of correctness*"). Esta "prueba de correctitud" es un número calculado a partir del Key Share y de las llaves de verificación $VK = v$ y $VK_\mathrm{i} = v_\mathrm{i}$ que permite demostrar que esta firma fue generada por un nodo válido.

Listado 4.7: Generación de un *Signature Share*

```
1  procedure GenerateSignatureShare(kshare, x, mi)
2  require x, un documento hasheado; H, una funcion de hash cuyo output tenga largo HASH_LEN
3  ensure un SignatureShare con la firma parcial
4      if Jacobi(x, n) == -1 then
5          x := x · mi.vk_u^e mod n
6      endif
7
8      sshare.id := kshare.id
9      Δ := Factorial(mi.l)
10     sshare.x_i := x^(2kshare.s) mod n
11
12     r := RandomBits(sizeof(n) + 2HASH_LEN)
13     v' := v^r mod n
14     x̃ := x^4 mod n
15     x' := x̃^r mod n
16
17     sshare.c := H(mi.vk_v | mi.vk_u | x̃ | mi.vk_i | x_i^2 | v' | x') mod n
18     sshare.z := c \cdot sshare.x_i + r
19
20     return sshare
21
22 end procedure
```

El algoritmo mostrado en el 4.8 sirve para verificar que un Signature Share fue generado por un Key Share válido. Este algoritmo se ejecuta en el mismo nodo de la aplicación, en el contexto de la biblioteca Libpkcs11. La operación consiste en una verificación matemática de que la firma parcial fue generada por un nodo con una llave valida.

Listado 4.8: Verificación de un *Signature Share*

```
1  procedure VerifySignatureShare(ss, x, mi)
2  require ss, un SignatureShare; x, el documento usado para generar sshare
3  ensure la correctitud de ss
4
5      if Jacobi(x, n) == -1 then
6          x := x · mi.vk_u^e mod n
7      endif
8
9      x̃ := x^4 mod n
10     v' := v^(ss.z) · v_i^(-ss.c)
11     x' := x̃^z · ss.x_i^(-2·ss.c)
12
13     h := H(mi.vk_v | mi.vk_u | x̃ | mi.vk_(ss.id) | x_i^2 mod n | v' | x'} mod n
14
15     return h == c
16
17 end procedure
```

El algoritmo descrito en el listado 4.10 sirve para unir k *Signature Shares* y así generar una firma RSA estándar válida. Para realizar esta operación se requiere utilizar el algoritmo 4.9 que sirve para calcular los multiplicadores de Lagrange, números clave que permite realizar la unión de k firmas parciales entre l posibles firmas totales, con $k < l$.

Listado 4.9: Interpolación de Lagrange adaptada a las necesidades de la criptografía umbral.

```
1  procedure LagrangeInterpolation(id, S, Δ)
2  require id, identificador de un Signature Share en particular;
3          S, un arreglo con k Signature Shares y
4          Δ, con el factorial del total de nodos
5  ensure λ^S_{0,id}, multiplicadores de Lagrange
6
7      num := 1
8      dem := 1
9      for i := 0 to k-1 do
10         if S_i.id != id then
11         num := num · S_i.id
12         den := den · (S_i.id - id)
13         endif
14     endfor
15
16     return Δ · num/den
17
18 end procedure
```

Listado 4.10: Unión de *k Signature Shares*

```
1  procedure JoinSignatureShares(S, x, mi)
2  require S, un arreglo de k \emph{Signature Shares}; x, un documento a firmar y
3          mi un Key Metainfo
4  ensure y, bytes con una firma RSA
5
6      jacobied := false
7      if Jacobi(x, n) == -1 then
8          x := x · mi.vk_u^e mod n
9          jacobied := true
10     endif
11
12     e' := 4
13     n := mi.public_key.n
14
15     w := 1
16     for i := 0 to k-1 do
17         x_i := S_i.id
18         λ_k := LagrangeInterpolation(S_i.id, S, Δ)
19         w := w · x_i^{2λ_k} mod n
20     endfor
21
22     (_, a, b) := ExtendedGcd(e', e)
23     y := w^a x^b mod n
24
25     if jacobied then
26         y := y · u^{-1} mod n
27     endif
28
29     return y
30 end procedure
```

4.4. Testing

En el desarrollo implementaciones de protocolos criptográficos se debe tener un especial cuidado en probar de manera exhaustiva el código. Esto se debe a que la seguridad de un sistema que utiliza criptografía se basa en gran parte en la confianza de que la implementación no posee errores o *bugs* graves. Dentro de la Ingeniería de Software, una de las maneras tradicionales para buscar esto es hacer testing. Sin embargo, según Myers et al. [28], "Realizar testing es el proceso de ejecutar un programa con la intención de buscar errores.", lo que coincide con una conocida frase de E. Dijkstra al respecto, que dijo ya en 1969: "Program testing can be used to show the presence of bugs, but never to show their absence!", lo que se interpreta como que realizar testing sirve para buscar la presencia de errores, nunca su ausencia. Esta descripción ayuda a comprender el real alcance de realizar pruebas de este tipo. Esto no quiere decir que realizar testing no sea importante, si no más bien a como enfocar los tests escritos.

Para el desarrollo de la biblioteca se realizaron dos niveles de testing: tests unitarios (o de módulo), test de sistema (en donde se prueba la biblioteca en sí).

4.4.1. Tests unitarios

Realizar testing unitario se refiere a la operación de realizar testing sobre las funciones que componen un programa. Más que realizar testing de un programa, es realizar testing sobre las componentes del programa. En el caso de la biblioteca desarrollada para este trabajo, a cada una de las funciones que la componen tanto públicamente (como parte de la API), como privadamente (funciones utilitarias) se les realizaron pruebas. Estas pruebas se diseñaron pensando en la definición estricta de las funciones, tanto en sus posibles entradas como en sus posibles salidas.

4.4.2. Test de sistema

Realizar *testing de sistema* tiene el objetivo de buscar por errores en la implementación de un sistema, al ser comparado con los objetivos de este. En el caso de la biblioteca de criptografía umbral, la idea es buscar errores en el procedimiento de generar una firma RSA válida al utilizar los mecanismos de criptografía umbral.

Este test difiere de los tests unitarios en que es un sólo programa (con su estado asociado) que se encarga de utilizar varias funciones del sistema.

Una versión simplificada del test se muestra en el listado 4.11.

Listado 4.11: Test de sistema

```
1  procedure test_system
2      (ks, info) := GenerateKeyShares(1024, 3, 5)
3      message := H(''Hola mundo'')
4
5      for i := 0 to info.l do
6          ss_i := GenerateSignatureShare(ks_i, message, info)
7          assert VerifySignatureShare(ss_i, message, info)
8      endfor
9
10     signature := JoinSignatureShare(ss, message, info)
11
12     assert RSAVerify(signature, doc, info.public_key, H)
13 end procedure
```

Capítulo 5

Verificación formal

Verificación formal significa explorar rigurosamente la correctitud de un sistema utilizando modelos matemáticos, normalmente mediante la ayuda de computadores.

En éste capítulo se introduce el uso de verificación formal en sistemas distribuidos, una técnica de modelamiento de un sistema distribuido en particular, y cómo se utilizó ésta técnica, a modo exploratorio, para encontrar un error de programación en TCHSM.

5.1. Verificación formal del sistema distribuido

El cómo modelar un sistema distribuido para expresar su comportamiento ha sido ampliamente estudiado desde hace más de 35 años. Por ejemplo, en 1976 Keller presentó dos modelos formales para computación paralela: un modelo conceptual abstracto y un modelo de programa paralelo. El primero no distingue entre estados de control y de datos, y el segundo incluye la capacidad de representar un infinito conjunto de estados de control [21].

Los enfoques tradicionales para el modelamiento de sistemas distribuidos es utilizar alguna herramienta teórica existente, tales como redes de Petri [32]. Con las redes de Petri algunas propiedades pueden ser probadas, tales como *reachability* (si un nodo es alcanzable o no), *liveness* (cuando no existe estado que no permite la continuación del trabajo) y *boundedness* (cuando el número de estados alcanzables es acotado). Otra interesante herramienta para verificar programas concurrentes es la lógica temporal [25]. La lógica temporal es utilizada como una herramienta para razonar acerca de secuencias de estados. Esta lógica es usada para describir propiedades de programas concurrentes. Algunas de estas son exclusión mutua, ausencia de *deadlock*, terminación, precedencia, entre otras [25].

El principal problema con los enfoques previamente mencionados es el modelo en si. El modelo sigue siendo una representación de alto nivel del sistema, e incluso con herramientas de validación existentes, la mayoría de los modelos son versiones simplificadas del sistema. Estas versiones pueden simplificar aspectos que posiblemente afectan el resultado de la verificación. Esto se debe principalmente a las restricciones presentes en las herramientas de modelamiento.

Por otro lado, los programas concurrentes pueden ser verificados directamente en el lenguaje de programación en el que fueron escritos utilizando verificadores especializados. En el caso del lenguaje de programación C existen VCC [12], Verifast [20], CBMC [11] y Spin [17], entre otros. Estos verificadores suelen evaluar propiedades como ausencia de *deadlock* o correcto uso de la memoria, sin la necesidad instrumentar manualmente el código. Además proveen maneras para escribir nuevas propiedades, utilizando aserciones o pre y post condiciones.

En esta sección se propone una manera para realizar verificación formal de un sistema distribuido modelando la comunicación del sistema como un programa concurrente. Esto puede ser realizado directamente en el lenguaje en que está escrito el sistema, utilizando las mismas primitivas que se utilizarán cuando se pase a producción. De este modo, los desarrolladores pueden verificar formalmente sistemas distribuidos utilizando las herramientas a las que están acostumbrados. De acuerdo a lo conocido por el autor, este es el primer acercamiento para utilizar esta técnica en la verificación formal de sistemas distribuidos.

5.2. Modelamiento de la comunicación de un sistema distribuido como un sistema concurrente

La propuesta consiste en la utilización de herramientas de verificación de programas concurrentes que existen hoy, sobre la implementación de un sistema distribuido escrito en C. Para realizar la verificación el sistema tiene que ser modelado utilizando un único proceso, de manera de poder utilizar los verificadores existentes. Este enfoque puede verse en la figura 5.1. La idea principal es instanciar el sistema distribuido utilizando threads y colas atómicas para ser validados con herramientas comunes. Los rectángulos de bordes redondeados (M, M1, M2, M3 y M4) representan procesos, los rectangulos (P1, P2, P3, P4 y P5) representan threads. Las flechas de color azul representan comunicación entre procesos y las flechas de color negro unidireccionales representan comunicación vía memoria compartida.

El principal problema teórico de modelar un sistema distribuido como un programa concurrente es la demostración de la equivalencia entre la comunicación en red y la memoria compartida dentro de un proceso. Este tema será discutido posteriormente.

La principal técnica es la utilización de dos herramientas: threads (procesos livianos) y colas con operaciones atómicas [9]. Lo primero es necesario para modelar cada nodo como un nodo de ejecución independiente. La segunda es necesaria para simular de manera efectiva la comunicación en red en un ambiente que utiliza la memoria compartida como la única primitiva de comunicación.

Normalmente, las herramientas que realizan verificación formal en programas concurrentes tienen un buen soporte de los mecanismos de sincronización que proveen los lenguajes de programación, sin embargo, se debe tener un especial cuidado en la especificación del lenguaje que soporta el verificador. Por ejemplo, la mayor parte de los verificadores formales no tiene soporte para los tipos atómicos de la especificación C11 [19], y por lo tanto no se pueden utilizar como mecanismos de sincronización.

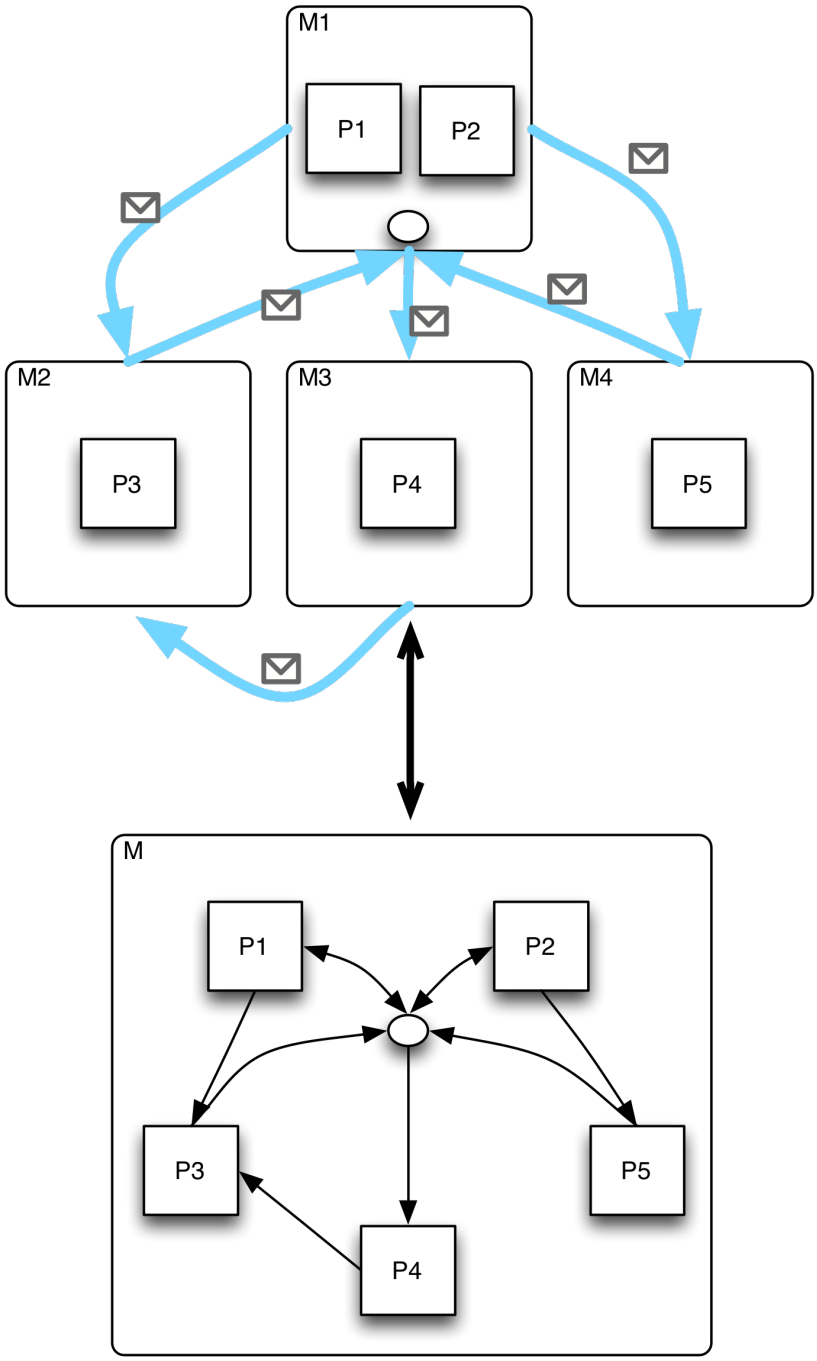


Figura 5.1: Diagrama que describe un modelo de equivalencia entre memoria compartida y mensajería asíncrona

Dependiendo del enfoque utilizado en la verificación, el modelo y los *headers* de las bibliotecas utilizadas deben ser instrumentadas en maneras ortogonales. Con verificadores basados en *ejecución simbólica*, el código debe ser instrumentado con pre y post condiciones, así como también con invariantes de bucles. De esta manera se pueden escribir propiedades complejas pero se debe realizar bastante esfuerzo en aprender el *lenguaje de instrumentación*.

Por otro lado, con *model checkers*, el código se instrumenta con aserciones y suposiciones. Esta manera es más natural para los desarrolladores, ya que parte del flujo normal de trabajo de los desarrolladores incluye la instrumentación con aserciones. Sin embargo, las propiedades que este enfoque permite modelar suelen ser más simples que las del enfoque previo.

Escoger uno u otro enfoque depende exclusivamente del esfuerzo que el equipo de desarrollo está dispuesto a realizar.

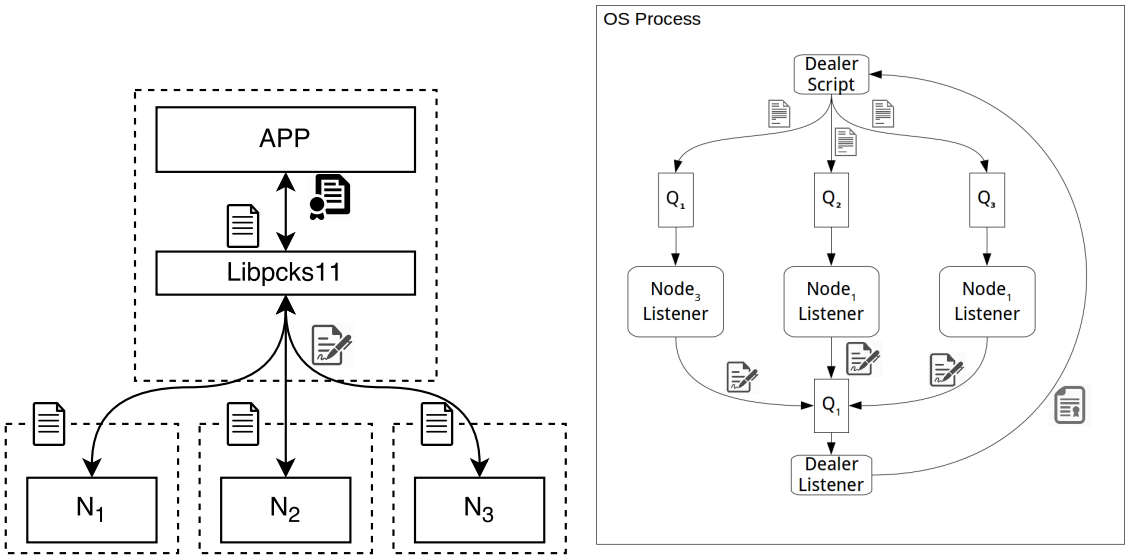
La equivalencia entre la comunicación por red y la memoria compartida es un tema ampliamente discutido en trabajo previo. Sin embargo, se ha estudiado principalmente como simular la memoria compartida a través de la comunicación por red [4,14,38]. Lo que se quiere realizar en este caso es justo al revés, es decir, simular la comunicación por red a través de los mecanismos provistos por la memoria compartida. Existen procedimientos conocidos para proveer esto último. En particular, el procedimiento propuesto aquí, utilizando variables condicionales para proveer una primitiva de señalización a las colas de entrada sigue el procedimiento de emulación de Kshemkalyani y Singhal de su libro "Distributed Computing: Principles, Algorithms and Systems" [23]. En él, ellos dicen:

> "... The shared space can be partitioned into disjoint parts, one part being assigned to each processor. 'Send' and 'receive' operations can be implemented by writing and reading from the destination/sender address space ... The write and read operations need to be controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received. ..."

Lo que se puede intepretar como que el espacio de memoria se puede dividir en partes disjuntas, cada una para cada procesador. Además, las operaciones de comunicación se pueden simular mediante una correcta utilización de primitivas de sincronización.

5.3. Verificación del protocolo de TCHSM

La actualización realizada a TCHSM para esta tesis utiliza un nodo maestro y múltiples nodos firmadores para realizar las operaciones para las cuales fue diseñado. En la figura 5.2(a) se muestra un diagrama que simboliza la generación de una firma para un documento dado utilizando el sistema distribuido: el ícono de un documento simboliza el documento entregado por alguna aplicación, el ícono de un lapiz firmando un documento simboliza una firma parcial entregada por un nodo, y el ícono de un documento con un listón simboliza un firma RSA válida. Cada uno de los rectángulos con lineas discontinua simboliza un proceso distinto, los cuales pueden estar corriendo tanto en la misma máquina, como en máquinas distintas. Las flechas simbolizan comunicación entre los nodos.



(a) Distributed TCHSM (b) Multithreaded TCHSM

Figura 5.2: En (a) cada rectángulo es un proceso del sistema operativo distinto, que puede incluso estar en otra máquina física, con comunicación a través de la red. En (b) cada rectángulo redondeado es un thread y cada rectángulo es una cola atómica. El paso de mensaje distribuido es simulado mediante el encolamiento atómico.

Para el modelado del protocolo de criptografía umbral como un único programa concurrente, el cual utiliza threads y colas atómicas para simular la comunicación por red, se separó el sistema en 3 tipos de *threads*: el dealer script, el dealer listener y los node listeners. En la figura 5.2(b) se muestra un diagrama que simboliza la generación de una firma para un documento dado, utilizando el sistema concurrente. En ésta figura, cada uno de los íconos simboliza lo mismo que en la figura anterior.

Los nodos N_{i} de la figura 5.2(a) son equivalentes a los Node_{i} Listeners de la figura 5.2(b). La operación de firmado ocurre como se describe a continuación. En el sistema distribuido, la aplicación entrega un mensaje para ser firmado a la biblioteca Libpkcs11. Luego, esta biblioteca reenvía el mensaje a cada uno de los nodos. Los nodos realizan la operación criptográfica y los resultados se envían a Libpkcs11. Finalmente, Libpkcs11 se encarga de unir las firmas de cada nodo y entregar el resultado a la aplicación. En el sistema concurrente, el dealer script entrega un mensaje para ser firmado a cada uno de los node listener. Luego, los node listeners realizan la operación criptográfica y los resultados se envían al dealer listener. Finalmente, el dealer listener se encarga de unir las firmas de los node listeners y entrega el resultado al dealer script. Lo anterior es una muestra de como se simulan las operaciones de Libpkcs11 con el eso de los 3 tipos de threads.

El dealer script se muestra en el listado 5.1. Ahí se puede ver que cada *thread* tiene su propio espacio de memoria disjunto de la forma node_info. En el listado 5.2 se puede observar la abstracción de comunicaciones por red simula la comunicación por medio de la utilización de una cola con operaciones atómicas. En el listado 5.3 se muestra el listener de los nodos. Ahí, los nodos esperan por nuevos mensajes con los cuales realizar operaciones. Finalmente, en el listado 5.4 se muestra el dealer listenet. Ahí se espera por las respuestas de los nodos.

Listado 5.1: Dealer script

```
1  void master_job(int node_cnt) {
2      /* Procedimientos de inicialización de recursos */
3      ...
4      for(int i=0; i<node_cnt; i++) {
5          node_info_init(info + i, i, master_get_info());
6          if (pthread_create(nodes+i, 0, node_job, info+i) != 0){
7                  perror("Create node_job");
8                  abort();
9          }
10     }
11     /* Script de pruebas */
12     ...
13     /* Abstracciones de red */
14     char **uuids = broadcast_key_shares(info, key_shares, meta_info);
15     ...
16     signature_share_t **signature_shares = broadcast_sign(info, doc_pkcs1,
17 uuids, meta_info);
18     ...
19     for (int i=0; i < meta_info->l; i++) {
20         signal_delete_key_share(info + i, uuids[i]);
21         signal_stop(info + i);
22         if(pthread_join(nodes[i], 0) != 0) {
23                 perror("pthread_join");
24         }
25     }
26     ...
27     /* Procedimientos de limpieza de recursos */
28 }
```

Listado 5.2: Comunicación entre threads

```
1  broadcast_sign(struct node_info *node, bytes_t *in_doc, char **uuid, const key_meta_info_t *mi) {
2      ...
3      for (int i=0; i<mi->l; i++) {
4          /* Prepara el mensaje y lo envía a l nodos. */
5          ...
6          queue_enqueue(node[i].queue, node_cmd);
7      }
8      /* Espera a que el dealer listener reciba como mínimo k mensajes de respuesta, donde k < l */
9      pthread_mutex_lock(&internal_mutex);
10     while(sign_count_k > 0) {
11         pthread_cond_wait(&store_cond, &internal_mutex);
12     }
13     pthread_mutex_unlock(&internal_mutex);
14     ...
15 }
16 signal_delete_key_share(struct node_info * node, const char * uuid) {
17     /* Prepara el mensaje */
18     ...
19     queue_enqueue(node->queue, node_cmd);
20     /* Espera a que el dealer listener reciba la respuesta */
21     pthread_mutex_lock(&internal_mutex);
22     while (delete_count > 0) {
23         pthread_cond_wait(&delete_cond, &internal_mutex);
24     }
25     pthread_mutex_unlock(&internal_mutex);
26 }
```

Listado 5.3: Node Listener

```
1  node_job(struct node_info * info) {
2      node_cmd_t * node_cmd;
3      master_cmd_t * master_cmd;
4      while(working) {
5          node_cmd = queue_dequeue(info->queue);
6          master_cmd = create_master_cmd();
7          master_cmd->node_id = info->id;
8          master_cmd->cmd = node_cmd->cmd;
9          /* Realiza las operaciones criptográficas y da una respuesta al dealer listener */
10         ...
11         queue_enqueue(info->master_queue, master_cmd);
12         node_cmd_dispose(node_cmd);
13     }
14 }
```

Listado 5.4: Dealer listener

```
1  master_listener() {
2      while(running) {
3          master_cmd_t * master_cmd = queue_dequeue(master_info.queue);
4          /* Recibe la respuesta y se la entrega al dealer script */
5          ...
6          master_cmd_dispose(master_cmd);
7      }
8  }
```

El modo en que el modelo fue realizado permite fácilmente reescribir las abstracciones de comunicación por red, y utilizar el mismo código para escribir cada módulo ejecutable. Esto significa que al momento de escribir cada thread como un único ejecutable, teniendo la certeza de que las abstracciones están implementadas correctamente, el mismo código ya está verificado.

5.3.1. Verificación del sistema

CBMC es un *Bounded Model Checker* [7, 11], lo que permite limitar la exploración de los estados de un sistema, haciendo así que el sistema sea factible de verificar. Como TCHSM tiene limites claros en sus parámetros, dependientes de la cantidad de nodos, para utilizar CBMC basta configurarlo para que utilice los parámetros definidos por el sistema. Al utilizar CBMC se verifican automáticamente varias propiedades tan sólo con definir aserciones en cada función y suposiciones en la función *main*. Por otro lado, se debe tener especial cuidado en las suposiciones realizadas, ya que estas pueden afectar los resultados de la verificación.

CBMC busca probar de manera automática ausencia *memory leaks*, ausencia de *indices fuera de límites en un arreglo* y ausencia *data races*, entre otras propiedades. Sin embargo, su lógica no es lo suficientemente extensible para verificar propiedades como conformidad de las operaciones criptográficas. En el caso de TCHSM, servirá para probar las propiedades del sistema distribuido y no de las operaciones criptográficas. Esto es, probar que el diseño de la comunicación, conforme a los parámetros del sistema, no tiene *deadlocks*, uso inválido de punteros o estructuras no inicializadas.

5.3.2. Prueba de concepto: análisis a una función

Como prueba de concepto, se instrumentó el código de modo de intentar verificar una función: la generación de un conjunto de Key Shares (listado 4.6). Estas estructuras contienen a id (listado 4.3), un número que identifica al Key Share en particular dentro de un conjunto de Key Shares. Si éste conjunto contiene l Key Shares, id debe cumplir que $1 \leq \text{id} \leq l$, de acuerdo al procedimiento matemático utilizado por el protocolo.

El algoritmo implementado, entrega como resultado un arreglo con l Key Shares, el cual representa al conjunto de Key Shares. Este arreglo, cómo todos los arreglos en el lenguaje C, tiene índices que van desde el 0 hasta $l - 1$, y los números identificadores de los Key Shares están correlacionados de manera que el Key Share i-ésimo del arreglo tiene número identificador $\text{i} + 1$.

Para verificar que los Key Shares estén inicializados correctamente, se instrumentó el código con la post-condición descrita en la aserción mostrada en el listado 5.5.

Listado 5.5: Aserción de verificación de inicialización

```
1 for (int i = 0; i < l; i++) {
2     assert(ks[i] != NULL);
3     assert(ks[i]->id == i+1);
4 }
```

Para que la ejecución de CBMC sea factible, se deben realizar suposiciones que permiten disminuir el número de estados posibles. En particular, para la verificación de la funcion descrita en esta sección se realizó la suposición inicial de que el parámetro k (parámetro que describe la cantidad de nodos necesarios para generar una firma) es igual a 2, y el parámetro l (parámetro que describe la cantidad total de nodos) es igual a 3. Esta suposición es la configuración del sistema factible con la menor cantidad de nodos, y la idea es liberar esta suposición a medida que se avanzaba en la verificación del sistema. Sin embargo, bastó con esta suposición para encontrar un error de programación.

Al ejecutar CBMC sobre esta función, lanza el mensaje mostrado en el listado 5.6. Este mensaje describe que hay una propiedad que no se cumple para algún estado posible del sistema. Posteriormente a mostrar ese mensaje, se muestra un contraejemplo y la propiedad que no se cumple. Dado que el contraejemplo tiene una salida muy verbosa, se muestra una versión reducida en el listado 5.7.

Luego del análisis de este contraejemplo, se encontró que el bug era la desigualdad mostradada en el listado 5.8, la cual no debía ser estricta de acuerdo a los índices utilizados.

Si bien este error puede ser encontrado por otros medios, utilizar las herramientas de verificación formal permite tener una herramienta potente a la disposición del desarrollo. Además, realizar esta prueba de concepto permite tener una mirada más acabada de las ventajas y desventajas de utilizar métodos formales para comprobar la correctitud del sistema.

Listado 5.6: Salida de CBMC

```
Generated 19 VCC(s), 14 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.0 with simplifier
116683 variables, 227727 clauses
SAT checker: negated claim is SATISFIABLE, i.e., does not hold
```

Listado 5.7: Contraejemplo y propiedad no cumplida

```
Building error trace
Counterexample:

State 19 file algorithms_generate_keys.c line 42 thread 0
------------------------------------------------------
  out=((struct key_metainfo **)NULL) + 4096 (00000000000000000000000000000000000000000000000000001000000000000)

State 20 file algorithms_generate_keys.c line 42 thread 0
------------------------------------------------------
  bit_size=4096 (00000000000000000000000000000000000000000000000000001000000000000)

...

State 1210 file algorithms_generate_keys.c line 145 function tc_generate_keys thread 0
------------------------------------------------------
  i=2 (00000000000000000000000000000010)

Violated property:
  file algorithms_generate_keys.c line 147 function tc_generate_keys
  assertion ks[i]->id == i+1
  (signed int)ks[(signed long int)i]->id == i + 1

VERIFICATION FAILED
```

Listado 5.8: Bug encontrado

```
    // Calcular Key Shares
    for(int i=1; i < l; i++) { // Esta desigualdad no debería ser estricta
        key_share_t * key_share = ks[TC_ID_TO_INDEX(i)];
        key_share->id = i;

        ...
    }
```

Capítulo 6

Conclusiones

6.1. Resultados y contribuciones

Esta tesis tiene como objetivo el estudio, mejora y propuestas para un sistema distribuido: TCHSM. Dicho sistema tiene como objetivo el reemplazo de un *Hardware Security Module* a través de una solución de software. El principal desafío de dicho sistema es proveer el mismo nivel de seguridad que proveen los HSM, utilizando un mecanismo criptográfico avanzado para la generación de firmas digitales. Dicho mecanismo es conocido como firmas RSA por umbral, y tiene la particularidad de que su resultados es compatible con firmas RSA estándar. El sistema original, sin las propuestas de cambio desarrolladas en este trabajo, es una interfaz de un sistema de criptografía umbral. El diseño original de esta interfaz incluye varios puntos críticos de falla, los cuales dejan al sistema sin una propiedad clave para la adopción del sistema: la tolerancia a fallas. Este problema es la motivación principal de este trabajo, donde las técnicas estudiadas y analizadas se puede utilizar además para otros sistemas distribuidos que se enfrenten a una situación similar.

6.1.1. Traslado de funcionalidad a nodos adyacentes

Al revisar las alternativas para quitar los variados puntos criticos de falla se encuentra un patrón particularmente importante: el traslado de la funcionalidad de un nodo crítico a los nodos adyacentes.

Ocurre en el caso de quitar los nodos RabbitMQ y reemplazarlos por sockets ZeroMQ, en donde un nodo RabbitMQ se encarga de realizar ruteo entre el nodo maestro y los nodos firmadores. Al utilizar los sockets ZeroMQ se traslada esa inteligencia a la definición de un protocolo confiable de mensajería asíncrona entre nodos, quitando definitivamente la necesidad de un nodo que realice routing. Cabe destacar que dada la existencia de infraestructura de bajo nivel que se encarga de realizar ruteo de manera eficiente y confiable, utilizar algo sobre eso que no lo aproveche realmente puede agregar overhead de manera innecesaria. Esto no quiere decir que el Middleware Orientado a Mensajes sea inútil en todos los casos, pero si

merece un estudio de las alternativas previo a su uso, especialmente en temas de escalabilidad y tolerancia a fallas.

Otro caso donde el patrón ocurre es en el caso de trasladar la funcionalidad criptográfica crítica desde el Trusted Dealer a una biblioteca en el lenguaje de programación C, la cual se espera sea utilizada directamente por la aplicación que utilizaría el HSM, todo por medio de la implementación de la API PKCS #11. Realizar esto nos deja sin la necesidad de confiar en un nodo central. El nodo en donde se encuentran las operaciones de generación y distribución de llaves es directamente incumbente en la seguridad de las operaciones. Esto quiere decir que se alcanza el mismo nivel de confiabilidad que se tiene un HSM: si nadie tiene acceso a este, nadie puede realizar operación alguna.

6.1.2. Implementación de un protocolo criptográfico

Para trasladar la funcionalidad desde el nodo Trusted Dealer a el nodo Aplicación es necesaria la implementación de la biblioteca criptográfica del segundo protocolo de *Practical Threshold Signatures* de Shoup [37] en el lenguaje de programación C. En el sistema TCHSM original existe una implementación del primer protocolo –cuyas propiedades de seguridad necesarias no están demostradas– desarrollada en Java. En esta tesis se implementa el segundo protocolo y se da una descripción detallada en pseudocódigo de cada uno de los algoritmos necesarios, para ser utilizada por cualquier implementador en el futuro. De este modo, se intenta acercar el desarrollo teórico criptográfico a los implementadores que lo requieran en el futuro.

Por otro lado, queda como un producto en si mismo y con la promesa de disponibilidad de manera pública, como software libre, de la biblioteca en el repositorio Github de NIC Chile Research Labs[1].

6.1.3. Técnica propuesta para la verificación formal de un sistema distribuido

El encontrar puntos críticos de fallas en el diseño del sistema distribuido lleva a la siguiente pregunta: ¿Es posible demostrar de manera automatizada que un sistema satisface ciertas propiedades? Esta pregunta tiene su base en el area de métodos formales, existiendo herramientas específicas para sistemas distribuidos. Estas herramientas necesitan un modelo del sistema escrito en lenguajes especificos para este fin. En esta tesis se propone la utilización de herramientas de verificación de sistemas concurrentes. ¿De qué manera? Se propone modelar el sistema distribuido como un sistema concurrente utilizando threads y colas con operaciones atómicas. De este modo se puede programar un modelo en el lenguaje de programación en que el sistema se desarrolla, y además, este modelo puede servir directamente como un test de sistema. En el trabajo se realiza una prueba de concepto utilizando el *Model Checker* CBMC.

[1] `https://github.com/niclabs/tclib`

6.2. Trabajo futuro

Para desarrollar en el futuro, se propone la siguiente lista de tareas pendientes:

- Realizar una comparativa de rendimiento entre el sistema distribuido original y el propuesto en esta tesis.
- Realizar una comparativa entre los dos protocolos de Shoup.
- Implementar algún algoritmo de generación distribuida de llaves.
- Realizar un completo análisis formal del sistema distribuido, para poder asegurar que cumple las propiedades del protocolo criptográfico.
- Utilizar la biblioteca criptográfica implementada en esta tesis para otros casos de usos, los que pueden ser:
  - Seguridad de nodos en redes ad-hoc.
  - Algún ingenioso protocolo de *evoting*. Por ejemplo, se puede realizar un mecanismo de firmado que permita que un directoria pueda firmar un documento digital de manera realmente distribuida, segura, robusta y tolerante a fallas. Exactamente como fue planteado en la introducción de este documento.

Bibliografía

[1] Ashraf Abusharekh and Kris Gaj. Comparative analysis of software libraries for public key cryptography. *Software Performance Enhancement for Encryption and Decryption, SPEED*, pages 11–12, 2007.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, IETF, March 2005.

[3] D. Atkins. Threat analysis of the domain name system (dns). RFC 2833, IETF, August 2004.

[4] Amotz Bar-Noy and Danny Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Mathematical Systems Theory*, 26(1):21–39, 1993.

[5] T Barros, A Cansado, and V Ramiro. Threshold Cryptographic Backend. `http://svsf40.icann.org/meetings/siliconvalley2011/presentation-cryptographic-backend-cansado-13mar11-en.pdf`, 2011. Presentation.

[6] Claudio Basile, Zbigniew Kalbarczyk, and Ravi K Iyer. Neutralization of errors and attacks in wireless ad hoc networks. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 518–527. IEEE, 2005.

[7] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[8] Dan Boneh. The decision diffie-hellman problem. In *Algorithmic number theory*, pages 48–63. Springer, 1998.

[9] Eliseu M Chaves, Prakash Ch Das, Thomas J LeBlanc, Brian D Marsh, and Michael L Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, 1993.

[10] Francisco Cifuentes, Alejandro Hevia, Francisco Montoto, Victor Ramiro, Tomás Barros, and Javier Bustos-Jiménez. Poor men's hardware security module (pmhsm): A threshold cryptographic backend for dnssec. In *Proceedings of the Chilean Workshop on Distributed Systems and Parallelism*, pages 1–6, 2014.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[12] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[13] Claude Crépeau and Carlton R Davis. A certificate revocation scheme for wireless ad hoc networks. In *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 54–61. ACM, 2003.

[14] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical Report IC/2003/77, EPFL, Laussane, December 2003.

[15] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.0 edition, 2015. `http://gmplib.org/`.

[16] William Hart and the MPIR Team. *MPIR: The Multiple Precision Integers and Rationals Library*, 2.7.2 edition, 2015. `http://www.mpir.org`.

[17] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, (5):279–295, 1997.

[18] RSA Security Inc. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40. OASIS Committee, 2015.

[19] ISO/IEC. Information technology – Programming languages – C. ISO 9899:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[20] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.

[21] Robert M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[22] Dusan Kozic, Benjamin Zwittnig, Janez Sterle, and Andrej Kos. Dnssec key management. *Elektrotehniski Vestnik/Electrotechnical Review*, 79, 2012.

[23] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.

[24] Haiyun Luo, Jiejun Kong, Petros Zerfos, Songwu Lu, and Lixia Zhang. Ursa: ubiquitous and robust access control for mobile ad hoc networks. *IEEE/ACM Transactions on Networking (ToN)*, 12(6):1049–1063, 2004.

[25] Zohar Manna and Amir Pnueli. Verification of concurrent programs. part i. the temporal framework. Technical report, DTIC Document, 1981.

[26] Mavroyanopoulos, N and Schumann, S. MHash Library. http://mhash.sourceforge.net/. Accessed: 2015-12-9.

[27] P.V. Mockapetris. Domain names: Concepts and Facilities. RFC 1034, IETF, November 1987.

[28] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[29] David Naccache. Double-speed safe prime generation. http://eprint.iacr.org/2003/175.pdf, 2003. Cryptology ePrint Archive: Report 2003/175.

[30] NIST. FIPS PUB 140-2: SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES, 2001.

[31] OpenBSD, security. http://www.openbsd.org/security.html. Accessed: 2015-09-14.

[32] Carl Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Germany, 1962.

[33] QNX technical articles, programming tools, opaque pointers. http://www.qnx.com/developers/articles/article_302_2.html. Accessed: 2015-09-14.

[34] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems, 1978.

[35] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[36] Mike Shema. *Hacking web apps: detecting and preventing web application security problems*. Newnes, 2012.

[37] Victor Shoup. Practical threshold signatures. *Advances in Cryptology—EUROCRYPT 2000*, pages 207–220, 2000.

[38] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, 1990.

[39] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[40] Team libtom. The LibTomCrypt Project. http://www.libtom.org. Accessed: 2015-12-9.

[41] Team libtom. The LibTomMath Project. http://www.libtom.org. Accessed: 2015-12-9.

[42] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org. Accessed: 2015-12-9.

[43] Xunhua Wang, Yih Huang, Yvo Desmedt, and David Rine. Enabling secure on-line dns dynamic update. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 52–58. IEEE, 2000.

[44] J Weise. Public key infrastructure overview. *Sun BluePrints OnLine, August*, pages 1–27, 2001.

[45] Michael J Wiener. Safe prime generation with a combined sieve. *IACR Cryptology ePrint Archive*, 2003:186, 2003.

[46] Bing Wu, Jie Wu, Eduardo B Fernandez, Mohammad Ilyas, and Spyros Magliveras. Secure and efficient key management in mobile ad hoc networks. *Journal of Network and Computer Applications*, 30(3):937–954, 2007.

[47] Hao Yang, Eric Osterweil, Dan Massey, Songwu Lu, and Lixia Zhang. Deploying cryptography in internet-scale systems: A case study on dnssec. *Dependable and Secure Computing, IEEE Transactions on*, 8(5):656–669, 2011.

[48] Seung Yi and Robin Kravets. Key management for heterogeneous ad hoc wireless networks. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on*, pages 202–203. IEEE, 2002.