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

IMPLEMENTACIÓN Y ESTUDIO DE SEGURIDAD DE UN SISTEMA DE RESPALDO DE DATOS DISTRIBUIDO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FERNANDO KRELL LOY

PROFESOR GUÍA:
ALEJANDRO HEVIA ANGULO

MIEMBROS DE LA COMISIÓN:
JOSÉ MIGUEL PIQUER GARDNER
JEREMY FELIX BARBAY

SANTIAGO DE CHILE
ABRIL 2009

ESTE TRABAJO HA SIDO FINANCIADO POR EL PROYECTO FONDECYT 1070332.

A mis padres Edgardo e Irene,

a mis hermanos Rodrigo y Javier,

y a Valentina.

Agradecimientos

Quiero agradecer al proyecto Fondecyt Nro. 1070332 por el apoyo económico entregado para la realización de esta investigación, a mi profesor guía, Alejandro Hevia Angulo, por toda su orientación durante el desarrollo de esta memoria y a mis compañeros Dusan, Sebastián, Pedro y Renato por su ayuda en ideas, discusiones y favores varios durante toda la realización del trabajo.

Fernando Krell Loy.

Santiago, Marzo de 2009

Índice General

Índice de figuras

Resumen

El trabajo consiste en la implementación y análisis de seguridad de un sistema distribuido de respaldo de datos, usando redes punto a punto, resistente ante fallas producto de ataques masivos de virus o gusanos.

En la fase inicial del trabajo se estudian dos protocolos de comunicación para el sistema: el protocolo simple (o básico) y el protocolo avanzado. En el protocolo simple, cada usuario, a través de un *password*, genera claves para cifrar los respaldos y para firmar los mensajes que se envían a los demás participantes. Ambos sistemas permiten garantizar la autenticidad, integridad y confidencialidad de los datos respaldados ante fallas masivas ocasionadas por ataques de virus y gusanos. Sin embargo, este protocolo no es seguro en escenarios donde los virus o gusanos pudieran activamente atacar el sistema de respaldo - por ejemplo, un virus que obtiene las claves y, con ello, realiza falsos respaldos o borra respaldos antiguos. Además, el protocolo alcanza algunas propiedades interesantes de seguridad sólo bajo el supuesto que ciertas condiciones - más bien exigentes - se cumplen, algo difícil de garantizar en todos los escenarios. Es por esto que se propone y analiza un protocolo avanzado, el cual logra evitar las limitaciones del protocolo básico ocupando técnicas criptográficas más sofisticadas aunque potencialmente de mayor costo.

Desde un punto de vista formal, la seguridad de ambos protocolos es analizada utilizando el paradigma de composición universal (o *Universal Composability* en Inglés). El objetivo es realizar una demostración matemática y rigurosa de la seguridad de los protocolos distribuidos propuestos.

En este trabajo también se documenta el diseño y desarrollo de una implementación del protocolo básico. La aplicación se implementó usando componentes de manera que sea extensible a incorporar nuevas herramientas y mejoras.

Finalmente, se analiza la incorporación de un sistema de reputación al sistema de respaldo propuesto. Con esto se pretende evitar que algunos usuarios hagan mal uso del sistema.

Capítulo 1

Introducción

Hoy en día Internet es altamente vulnerable. Virus y gusanos son cada vez más sofisticados y cada vez afectan a más usuarios. La principal razón es que estos virus y gusanos pueden esparcirse fácilmente vía correo electrónico y otras aplicaciones masivas. Los daños producidos por estas verdaderas epidemias varían en gran forma, algunos sólo sobrecargan el ancho de banda, otros botan el servicio de algún host, borran datos en discos, etc. Algunos ejemplos importantes son: el gusano *Code Red* en 2001 que afectó a 360.000 hosts en 14 horas [21], por su parte *Sapphire* inspeccionó la mayoría de las direcciones IP en menos de 10 minutos [22].

Con cada vez más frecuencia los usuarios guardan datos propios, que pueden ser de suma importancia, en su propio dispositivo (PC, PocketPC, celular) o en algún servidor que suponen confiable. El problema es que ningún computador conectado a Internet (o a alguna otra red masiva) está libre virus u otras fallas de seguridad que lo afecten. Además, los servidores supuestamente confiables pueden dejar de serlo y pasar a ser altamente desconfiables de un momento a otro. Por ejemplo, hoy en día se observa a una gran cantidad de usuarios utilizando y "guardando"[1] datos (fotos, videos, etc) en comunidades virtuales como *Facebook*. Nadie puede asegurar que esos datos estarán disponibles indefinidamente o que no se haga mal uso de ellos. Dado el problema, la pregunta que surge es ¿Dónde se deben respaldar los datos? ¿Quién debe hacerlo? ¿Cómo evitar que estas epidemias los afecten? En general, las soluciones propuestas a este problema son muy complejas o de muy alto costo para la mayoría de los usuarios.

[1] No sólo se guardan, también se comparten y se da acceso a otros usuarios.

1.1. Propuesta

En este trabajo se propone un sistema de respaldo de datos distribuido en donde los mismos usuarios del sistema son los que respaldan los datos de los demás usuarios. Basándose en el concepto de "informed replication", propuesto por Junqueira et al [17], los usuarios "servidores" son elegidos en forma cuidadosa de manera que una epidemia no pueda afectar a todos estos usuarios. Por lo tanto, siempre existirá al menos un usuario que sobreviva y tenga los datos del usuario "cliente". De esta manera, los datos no dependen de una sola entidad que pueda cambiar sus políticas y se disminuye la probabilidad de que los datos sean afectados por algún virus o que un ataque a una entidad centralizada comprometa los datos.

Una de las cualidades del sistema es que está diseñado para ser implementado en diferentes ambientes o redes (Internet, red de telefonía celular, etc.). Por ejemplo, sería de gran utilidad poder respaldar los datos que se guardan en el teléfono celular de algún usuario en un teléfono celular de otro usuario, para en caso de cambiar el celular, poder recuperar los datos estando en cualquier lugar geográfico. De esta manera, los datos dependen de los usuarios de la red, posiblemente amigos o conocidos del usuario que desea respaldar, y no de la empresa que provee el servicio (lo cual es un gran ventaja para el usuario y para la empresa[2]).

El sistema se define como un protocolo de comunicación genérico entre los participantes, que ocupa técnicas y herramientas criptográficas para garantizar ciertas propiedades de seguridad. Se proponen dos implementaciones posibles para el protocolo; el protocolo básico (o simple) y el protocolo avanzado. En el protocolo básico cada usuario, a través de una contraseña, genera claves para cifrar los respaldos y para firmar los mensajes del protocolo. Sin embargo, este protocolo no es seguro en escenarios en donde un virus o gusano pudiera atacar activamente el sistema obteniendo, por ejemplo, las claves. Por tal motivo es necesario proponer un segundo protocolo (protocolo avanzado) que ocupe técnicas criptográficas más avanzadas para evitar que un virus o gusano realice respaldos falsos o borre respaldos antiguos.

En la sección 3.1 del presente trabajo se da una breve introducción a "*Universal Composability*". *Universal Composability* es un marco de trabajo teórico utilizado en la demostración

[2]El usuario conoce y confía en quien respalda sus datos. La empresa, además de proveer el servicio, no requiere de gastos en disponibilidad de los datos, que pueden llegar a ser enormes y no asume la responsabilidad en caso de fallas o robo de información. Ello no impide que la empresa ofrezca un servicio de respaldo en forma adicional, con costo por ejemplo.

de seguridad para protocolos en computación distribuida. En esta introducción se dan las definiciones y teoremas básicos para poder demostrar, en forma rigurosa, que dado algunos supuestos mínimos, el protocolo básico y el avanzado cumplen las propiedades de seguridad.

Durante el trabajo se realizó una implementación del protocolo básico sobre TCP/IP en donde los participantes mantienen su IP fija. Esta implementación se puede extender a redes en donde los usuarios cambien su dirección IP e incluso cambien de dispositivo y, aún en este caso, los datos estén disponibles.

En el trabajo de memoria también se analiza la integración de un sistema de reputación que permita clasificar los usuarios en distintos niveles de confiabilidad, de manera de prevenir que usuarios hagan uso del sistema sin pagar sus costos[3].

En resumen el trabajo se divide en las siguientes tres etapas.

1. Definición e implementación de un sistema de respaldo de datos distribuido.

2. Análisis de la seguridad del sistema a través de una demostración matemática sobre las propiedades mínimas de seguridad.

3. Análisis de integración de un sistema de reputación que permita reforzar ciertos aspectos de seguridad no cubiertos por los protocolos descritos.

1.2. Motivación

Los dos principales aportes y desafíos de este trabajo son realizar una demostración matemática de la seguridad de los protocolo e implementar el protocolo de manera que sea extensible a incorporar nuevas funciones y mejoras.

En criptografía es muy común encontrar análisis y demostraciones de seguridad en protocolos de primitivas criptográficas (*commitment schemes*, firmas digitales, esquemas de encriptación, etc.) o en computación multi-partita de funciones. Sin embargo, es extremadamente inusual encontrar demostraciones para protocolos o sistemas distribuidos utilizados en la práctica (Sistemas de archivos, aplicaciones en comunidades virtuales como juegos, chats, etc.)

En el presente trabajo se presenta un sistema de respaldo de datos distribuido al cual se le realiza una demostración rigurosa de seguridad bajo el paradigma *Universal Compo-*

[3]El costo principal que debe pagar un usuario es respaldar los datos de otros usuarios.

sability(UC). Con esto se pretende incentivar el uso de UC para realizar demostraciones y proponer protocolos seguros en sistemas distribuidos de uso común. De la misma manera, se pretende mostrar un camino para que desarrolladores puedan efectuar demostraciones de seguridad sobre su sistema, de manera que cumplan con estándares altos de seguridad.

Además de la demostración teórica, se presenta una implementación básica del protocolo y una API para futuras implementaciones en otros ambientes o con objetivos de uso específicos. Esta implementación del sistema funciona actualmente en una red TCP/IP en donde los participantes mantienen su dirección IP fija. Sin embargo, el sistema es fácilmente extensible para ser utilizado en ambientes con IP dinámica.

También se discute en detalle sobre los diferentes ambientes en donde el protocolo puede ser implementado y los supuestos mínimos que se deben tomar para preservar la seguridad. Junto con esta discusión se propone extender el protocolo para incluir un sistema de reputación que permita clasificar a los usuarios según el uso que hacen del sistema. El sistema de reputación pretende evitar que usuarios deshonestos, que borran o modifican los datos que guardan de otros usuarios, puedan afectar significativamente el sistema. Definir la reputación de un usuario no es una tarea fácil en términos de seguridad; los usuarios no deben ser capaces de mentir sobre su reputación, ningún conjunto de usuarios pueden modificar la reputación de algún usuario, evitar que usuarios deshonestos cambien de identidad para subir rápidamente su nivel de reputación, evitar que usuarios con niveles altos de reputación puedan comportarse indebidamente sin ser "castigados", etc.

1.3. Objetivos

Implementar un sistema de respaldo de datos distribuido y estudiar en profundidad la seguridad de este sistema usando herramientas actuales de demostración matemática para seguridad de protocolos criptográficos.

Objetivos específicos

- Implementar un sistema de respaldo de datos que sea extensible para incorporar nuevos protocolos, herramientas criptográficas, mecanismos de comunicación e interfaces de usuario.

- Demostrar matemáticamente la seguridad del sistema.

- Estudiar un mecanismo de reputación que apoye la seguridad del sistema.
- Realizar una discusión sobre los ambientes en donde realizar otras implementaciones del protocolo y posibles usos.

Organización

En el capítulo 2 se da una breve introducción a protocolos de backup y se describen los protocolos a estudiar. En el capítulo 3 se describe el método de demostración UC y se utiliza para demostrar seguridad de los protocolos descritos en el capítulo 2. Luego en el capítulo 4 se analiza la integración de un sistema de reputación. El capítulo 5 se puede encontrar el diseño y las decisiones tomadas en la implementación realizada. Finalmente las conclusiones se presentan en el capítulo 6.

Capítulo 2

Protocolos

2.1. Enfoque

En el presente trabajo se pretende continuar el trabajo desarrollado de Junqueira et al. [17,18], en donde se propone un sistema de backup que resista a epidemias en redes. En [17] los autores proponen un diseño general para la construcción de sistemas distribuidos basado en la suposición de que las epidemias afectan a un conjunto de hosts que comparten parte de su configuración, por ejemplo, ocupan el mismo sistema operativo, navegador o servidor web. Una configuración de un usuario puede ser el par (*Windows, Firefox*), lo cual quiere decir que ese usuario ocupa el sistema operativo *Windows* y el navegador *Firefox*. La principal idea es diseñar sistemas distribuidos de manera que el servicio ofrecido esté disponible en hosts que tienen diferentes configuraciones. Aplicando esta idea a un sistema de backup, un usuario deberá copiar los datos a respaldar en un conjunto de otros usuarios que tengan diferentes configuraciones entre sí. De esta manera, un gusano que borre los datos, no podrá borrar los datos de un participante, pues este tendrá respaldados sus datos en al menos un host que no fue afectado.

El conjunto adecuado de usuarios que no comparten alguna configuración es llamado *core*, Junqueira et al. [17,19]. Como ejemplo, considerar el escenario en donde los posibles atributos para formar una configuración son:

- Sistema Operativo = $\{Unix,\ Windows\}$.
- Servidor Web = $\{Apache,\ IIS\}$.
- Navegador = $\{IE,\ Firefox\}$

Y los usuarios H_1,H_2,H_3 y H_4 tienen la siguientes configuraciones:

- $H_1 = \{Unix,\ Apache,\ Firefox\}$
- $H_2 = \{Windows,\ IIS,\ IE\}$
- $H_3 = \{Windows,\ IIS,\ Firefox\}$
- $H_4 = \{Windows,\ Apache,\ IE\}$

Un posible *core* es $\{H_1, H_2\}$, pues H_1 con H_2 no comparten ningún atributo en sus configuraciones. Otro posible *core* es $\{H_1, H_3, H_4\}$, pues ningún atributo se repite en los tres usuarios. En cambio si se considerara H_3 con H_4, se puede ver que ambos utilizan el sistema operativo *Windows*, por lo tanto una vulnerabilidad en este atributo afectará a ambos usuarios. El *core* al cual un participante pertenece no es fácil de encontrar, de hecho, encontrar uno de tamaño óptimo es NP-hard [1]. En [17] se propone una heurística para encontrar el *core* de un participante.

En un sistema de backup distribuido los participantes son clientes y servidores a la vez. Un participante es cliente cuando desea guardar sus datos en otro participante y es servidor cuando guarda los datos de otro participante.

Cualquier sistema de backup debe estar compuesto de tres fases principales; fase de respaldo, fase eliminación y fase recuperación. En [16] se propone que estas fases pueden ser compuestas por ocho funciones abstractas a implementar; cuatro para el cliente y cuatro para el servidor. A continuación se describen las tres fases y las funciones que se deben implementar:

- **Fase de respaldo:** Esta fase es ejecutada cada vez el cliente desea respaldar datos. En esta fase, pueden ejecutarse las funciones *client_gen_data*(D, did), *server_save_data*(M) y *server_gen_announce*(did, cid).

  En primer lugar el cliente debe generar un mensaje M llamando a la función *client_gen_data*(D,*did*), en donde D contiene los datos que se desean respaldar y *did* es el identificador de los datos. Ocupando la heurística de computo de *cores* [17], se calcula un *core* para el cliente y luego el mensaje M, llamado "data message", es enviado a todos los servidores en el *core*. Cada servidor, cuando recibe este mensaje, ejecuta la función *server_save_data*(M) que guarda el mensaje M. Una vez que el servidor ha

[1] Se puede reducir de SET-COVER [17].

guardado M, se debe publicar que ha guardado los datos (identificados por *did*) del cliente (identificado por *cid*). Al menos una vez el servidor debe generar un mensaje, llamado "announce message", ejecutando la función *server_gen_announce*(did, cid). Dependiendo de la implementación del protocolo el mensaje puede o no enviarse de inmediato. Algunas posibles opciones son: enviarlo a todos los clientes, enviarlo a un servidor central, enviarlo al e-mail del client o enviarlo cuando el cliente avisa que ha fallado.

- **Fase de eliminación:** Esta es una fase opcional que es ejecutada cada vez que un cliente decide que no necesita el respaldo de un cierto conjunto de datos. En esta fase, pueden ejecutarse las funciones *client_gen_release*(did) y *server_erase_data*(M).

  El cliente ejecuta la función *client_gen_release*(did) la cual genera un mensaje M, llamado "release message" para todos los servidores en el *core*. Luego este mensaje es enviado a todos estos servidores. Cuando cada servidor recibe este mensaje, ejecuta la función *server_erase_data*(M), el cual borra, si es que existen, los datos identificados por *did* del cliente *cid* que envió el mensaje.

- **Fase de recuperación:** Esta fase se ejecuta cuando un cliente ha sufrido una catástrofe y necesita recuperar sus datos. El objetivo es recuperar los datos que estaban guardados en el servidor. En esta fase, pueden ejecutarse las funciones *client_request_restore*(did,cid), *server_retrieve_data*(M) y *client_restore_data*(M').

  Cuando el cliente recibe un "announce message", debe generar un mensaje M, llamado "request_restore message", ejecutando la función *client_request_restore*(did,cid). El mensaje es enviado al servidor, el cual al recibirlo, ejecuta la función *server_retrieve_data*(M). Esta función extrae el identificador *did* y recupera los datos asociado a ese identificador generando un mensaje M', "restore message", que es enviado de vuelta al cliente. Apenas el cliente recibe este mensaje debe ejecutar la función *client_restore_data*(M'), entregando al usuario los datos (D) o $\perp$ si el mensaje M' no es válido.

2.2. Propiedades de seguridad para protocolos de backup

Un sistema de respaldo de datos debe proveer las siguientes tres propiedades o garantías de seguridad:

1. **Privacidad de los datos**: Ningún participante del sistema, excepto el cliente, debe obtener información alguna acerca de los datos que el cliente respaldó (con la posible excepción del largo de los datos).

2. **Integridad de los datos**: Cualquier alteración de los datos respaldados debe ser detectable por el cliente.

3. **Disponibilidad de los datos**: El cliente debe poder recuperar sus datos después de sufrir una catástrofe.

2.3. Protocolo básico

En este protocolo se asume que cada cliente recuerda una clave suficientemente larga, llamada *passphrase*, con la cual se pueden generar llaves locales. Además el cliente-servidor cuenta con las siguientes herramientas:

- Una tabla $T = T[cid, did]$ en donde el servidor guarda los datos did del cliente cid.
- Una función de hash $H(\cdot)$.
- Un algoritmo derivador de claves $KDF(\cdot)$.
- Un esquema de cifrado simétrico $\mathcal{SE} = (\mathcal{K}(\cdot), \mathcal{E}nc(\cdot,\cdot), \mathcal{D}ec(\cdot,\cdot))$.
- Un esquema de firmas $\mathcal{DS} = (\bar{\mathcal{K}}(\cdot), Sig(\cdot,\cdot), Vf(\cdot,\cdot))$.

Convención: Si A, B, C son strings, (A, B, C) denota una codificación inyectiva de ellos. Ejemplo, concatenación delimitada por separadores identificables.

Inicialización de claves

Para generar las claves, el cliente debe llamar la función *setup*(*passphrase*) que en forma determinista computa las llaves como $(r, k) \leftarrow KDF(H(passphrase))$, $(pk, sk) \leftarrow \bar{\mathcal{K}}(r)$ y entrega la tupla (pk, sk, k), en donde (pk, sk) son llaves usadas para firmar mensajes y k es la clave para encriptar los datos a respaldar.

Fase de Backup

- *client_gen_data*(D,*did*): Esta función recibe los datos a respaldar (D) y un identificador de los datos(*did*). Como salida entrega un mensaje M que contiene los datos encriptados, *did* y el identificador del cliente *cid*. Además contiene una firma generada con la clave secreta del usuario. Sea `ts` la hora local y *cid* el identificador único para el cliente que ejecuta la función.

  1. Recalcular las llaves (pk, sk, k) ejecutando *setup*(*passphrase*).
  2. Computar $C \leftarrow \mathcal{E}nc(k, D)$ como el cifrado de D con la clave k.
     Sea $R \leftarrow (C, \texttt{ts}, did, pk, cid)$.
  3. Computar la firma de R como $s \leftarrow Sig(sk, R)$ y definir M como (R, s).
  4. Borrar las claves (pk, sk, k).
  5. Entregar M como salida.

- *server_save_data*(M): Esta función recibe un mensaje M, posiblemente generado por *client_gen_data*$(\cdot, \cdot)$ y guarda (o actualiza) los datos *did* del cliente *cid* que envió el mensaje.

  1. Leer M como (R, s) y R como $(C, \texttt{ts}, did, pk, cid)$.
  2. Si *cid* es un nuevo cliente, incrementar el número de clientes.
  3. Si *cid* no es un nuevo cliente, chequear que la llave pública ya guardada sea igual a pk. Y si ya se había guardado datos con identificador *did* chequear que `ts` sea más reciente que el de los datos ya guardados.
  4. Chequear que el mensaje fue firmado correctamente por *cid* ($Vf(pk, R, s) = 1$). Si las verificaciones anteriores fueron exitosas, entonces guardar el mensaje en la tabla T ($T[cid, did] \leftarrow M$), sino, abortar.

- *server_gen_announce*(did, cid): Esta función genera un mensaje, *announce*, que informa que se tienen los datos *did* del cliente *cid*.

    1. Si $T[cid, did]$ existe, computar $M \leftarrow$ (“`announce`”, cid , did), si no, abortar.
    2. Entregar M.

Fase de Eliminación

- *client_gen_release*(did): Genera un mensaje con la petición que se liberen los datos *did*. Sea *cid* el identificador único para el cliente que ejecuta la función.

    1. Recalcular las llaves (pk, sk, k) ejecutando *setup*$(passphrase)$.
    2. Computar y firmar un pedido de liberación de datos como $R \leftarrow$ (“`release`”$, did, pk, cid)$ y $s \leftarrow Sig(sk, R)$.
    3. Definir M como (R, s).
    4. Eliminar las llaves (pk, sk, k).
    5. Entregar M como salida.

- *server_erase_data*(M): Si el mensaje M es válido, elimina los los datos *did* del cliente *cid*

    1. Leer M como (R, s) y R como $(\texttt{release}, did, pk, cid)$.
    2. Verificar que $T[cid, did]$ exista y que pk sea igual a la llave pública de *cid* ya guardada anteriormente.
    3. Verificar la firma del mensaje $(Vf(pk, R, s) = 1)$.
    4. Si ambas verificaciones fueron exitosas, borrar $T[cid, did]$.

Fase de Recuperación

- *client_request_restore*(did): Crea un mensaje M con la solicitud de recuperación de los datos *did*. Sea *cid* el identificador único para el cliente que ejecuta la función.

    1. Recalcular las llaves (pk, sk, k) ejecutando *setup*$(passphrase)$.
    2. Sea $R \leftarrow$(“`request_restore`”$, did, pk, cid)$.

3. Calcular la firma como $s \leftarrow Sig(R, sk)$.
4. Definir M como (R, s).
5. Borrar las llaves (pk, sk, k)
6. Entregar M como salida.

- *server_retrieve_data*(M): Recibe en la entrada un mensaje M, posiblemente generado por *client_request_restore*$(\cdot)$, verifica la validez del mensaje y devuelve el mensaje M' guardado por la función *server_save_data*(M'), cuando el cliente solicitó el backup.

  1. Leer M como (R, s) y R como ("`request_restore`" $, did, pk, cid$).
  2. Verificar que $T[cid, did]$ exista y que pk calce con la llave pública de cid ya guardada anteriormente.
  3. Verificar la firma del mensaje $(Vf(pk, R, s) = 1)$.
  4. Si las verificaciones fueron exitosas, entregar $M' \leftarrow T[cid, did]$. Sino, abortar.

- *client_restore_data*(M'): Sea cid el identificador único para el cliente que ejecuta la función.

  1. Recalcular las llaves (pk, sk, k) ejecutando *setup*$(passphrase)$.
  2. Leer M' como (R, s) y R como $(C', \texttt{ts}', did, pk', cid)$.
  3. Si $\texttt{ts}'$ es más reciente que la variable local *latest_ts_did*, entonces *latest_ts_did*$\leftarrow\texttt{ts}'$. Sino, abortar.
  4. Verificar que pk sea igual a pk' y que $Vf(pk, R', s') = 1$.
  5. Si todo verifica, computar $D \leftarrow \mathcal{D}ec(k, C')$, sino $D \leftarrow \perp$.
  6. Borrar llaves (pk, sk, k).
  7. Entregar D como salida.

Limitaciones del protocolo básico

A continuación se muestran algunas consideraciones a tomar en cuenta acerca del protocolo descrito en la sección anterior.

- El usuario debe ingresar una *passphrase* cada vez que desea ejecutar una función como cliente. Con esta *passphrase* se generan las llaves pk, sk y k en forma determinista. Cualquier error del usuario al momento de ingresar el *passphrase*, generará claves erróneas y el protocolo fallará. Una forma de arreglar esto es hacer que el usuario entregue el *passphrase* solamente la primera vez que ejecute el protocolo y cada vez que haya ocurrido una catástrofe. Al momento de entregar el *passphrase* se calculan las claves sk, pk y k. Para evitar recalcular estas claves cada vez, al usuario también se le puede pedir que ingrese una segunda clave (o *password*) de la cual se pueden obtener dos subclaves (K_1 y K_2). Con esto se aplica un esquema MAC[2] a (sk, pk, k) utilizando como clave K_1 y se cifra la tupla $((sk, pk, k), MAC(sk, pk, k))$ utilizando K_2, guardando el resultado en un archivo. Con este sistema cada vez que el usuario desee ejecutar alguna función del protocolo deberá proveer un *password* para que el sistema recupere las llaves. Si al momento de ejecutar una función el usuario ingresa un *password* incorrecto, entonces no se podrá leer correctamente el archivo que contiene las claves, esto puede ser verificado calculando el MAC correspondiente. Si el *password* es correcto se recuperan las claves y la función requerida se ejecuta sin problemas. De esta manera se previene que cualquier error al ingresar la clave genere claves incorrectas y, además, esto permite que el usuario pueda cambiar de clave cuando quiera sin cambiar su llave pública.

- Aunque se asuma que los usuarios son honestos[3], no es posible asegurar la disponibilidad de los datos a menos que el canal de comunicación sea autentificado. Si el canal no es autentificado, un adversario en el canal puede alterar los datos que se transfieren durante la fase de recuperación. Es posible suponer que los canales son autentificados en muchos ambientes, por ejemplo en redes cerradas. Sin embargo, en donde no existen canales autentificados es posible implementarlos si se provee de una infraestructura de clave pública.

[2]Código de Autentificación de Mensajes. Ver definiciones básicas en capítulo A .
[3]Siguen el protocolo, esto es, sus acciones son dictadas por las instrucciones del protocolo.

- Si los usuarios no son honestos, estos pueden eliminar los datos de otros usuarios cuando actúan como servidores y podrían ocupar el sistema sin pagar sus costos. La manera de solucionar esto es tener un sistema de reputación que permita clasificar a los usuarios según su comportamiento. Usuarios deshonestos (que no sigan el protocolo correctamente) tendrán una baja reputación. La integración de un mecanismo de reputación se analiza en el capítulo 4.

- Un supuesto fuerte tomado es que los usuarios son capaces de recordar un *passphrase* suficientemente grande que permita generar las claves sk, pk y k. Esto puede no ser viable, normalmente un usuario puede recordar un *passphrase* aleatorio de pocos caracteres, pero si el *passphrase* no es aleatorio tendría que ser demasiado grande, lo cual tampoco puede ser recordado por un ser humano. [4]

- Otro punto importante que no es tomado en cuenta, es que si un virus gana acceso al cliente durante la ejecución del protocolo y obtiene las claves secretas, puede generar mensajes de `release` o `backup` que borren o modifiquen los datos respaldados.

[4]Notar que una passphrase segura debiese tener el equivalente a al menos 128 bits de entropía, eso equivale a que el usuario debe recordar 16 caracteres ASCII totalmente aleatorios. Alternativamente, el usuario debe recordar una frase "normal" (en castellano o inglés) de al menos 64 caracteres asumiendo 2 bits de entropía por caracter [23].

2.4. Protocolo avanzado

En esta sección se describe un segundo protocolo que pretende mejorar algunas de las limitaciones del protocolo anterior. La gran ventaja de este protocolo es que un virus que obtenga las claves no podrá realizar respaldos falsos ni borrar respaldos antiguos a menos que la infección se realize en tres períodos de tiempo consecutivos. Los supuestos tomados para este protocolo son los siguientes:

- El usuario tiene acceso a un dispositivo con poder computacional limitado (por ejemplo, *smartcards* o dispositivos móviles similares). Las fases de respaldo y eliminación requieren acceso al dispositivo.

- Una infección de un virus en el cliente puede ser detectada en un período fijo tiempo de duración d horas, por ejemplo $d = 24$.

- Existe una infraestructura de clave pública (PKI) disponible. Con esto cada participante que reciba una clave publica conoce la identidad del dueño correspondiente. (Con PKI disponible se pueden implementar canales autentificados usando técnicas criptográficas estándares [4, 10]).

Las dos principales modificaciones de este protocolo respecto al anterior son:

1. Cada vez que un usuario respalda sus datos en el servidor, el servidor envía una firma que puede ser utilizada para certificar que ese servidor tiene los datos de ese usuario. Si el servidor borra los datos, el cliente puede demostrar a un tercero que ese servidor no ha seguido el protocolo, y por ende debe ser castigado de alguna manera[5]. El problema es que esto podemos implementarlo mediante un *protocolo de intercambio justo*, lo cual involucraría agregar una entidad centralizada confiable.

2. Para borrar y actualizar datos se requieren tres períodos consecutivos. En el primer período se pide al servidor borrar o actualizar los datos, en el segundo período no se realiza nada y en el tercer período se confirma la petición. Si la clave secreta es obtenida por un adversario en un cierto período de tiempo y pide actualizar o borrar datos, el

[5]Como se mencionó anteriormente se pretende agregar un sistema de reputación para implementar el mecanismo de castigo.

servidor, si es honesto, debe esperar una confirmación firmada en el período subsiguiente en donde se puede asegurar que la clave secreta del cliente ya fue actualizada (utilizando *key-insulated signatures*, Yung et al. [12, 13]). Esto previene que el adversario pueda borrar o actualizar datos de un usuario afectado en un período. La razón de tener un período intermedio entre la petición de backup y la confirmación, es que la infección de un virus puede ser detectada durante el segundo período, por lo que sólo en el tercer período se puede enviar una confirmación confiable.

El usuario guarda en un dispositivo (*smartcard*) una clave $K = K_1 \circ K_2$, en donde $|K_1| = |K_2| = \ell$. El parámetro de seguridad ℓ debe ser suficientemente largo, por ejemplo 128 bits.

Fase Configuración:

(Período 0: Antes que el usuario contacte a algún servidor)

- Input dispositivo: K_1.
- Output dispositivo: pk, la clave pública para el cliente.
- Input cliente: no tiene.
- Output cliente: no tiene.

1. El dispositivo auxiliar del usuario genera en forma determinista el par (msk, pk) a partir de K_1, en donde msk es la clave secreta maestra y pk es la clave pública. El dispositivo envía al cliente la clave pública. La clave pública se mantiene constante durante todos los períodos.
2. El cliente registra pk con la autoridad *PKI*.

Fase de Respaldo

(Período $i > 0$)

- Input dispositivos (para el cliente y servidor): La clave secreta maestra msk y la clave auxiliar K_2.

- Output dispositivos: Las claves secretas para el período actual $sk[i], k[i]$.
- Input cliente: D, did, cid, pk.
- Output cliente: Mensaje `data message`, que contiene los datos encriptados, el identificador de los datos y el identificador del cliente.
- Input servidor: Una tabla T en donde guarda los backups del cliente.
- Output servidor: s, una notificación de respaldo que permite al cliente demostrar a un tercero que el servidor tiene un backup suyo. $T'[\cdot,\cdot]$, la tabla T actualizada.

1. El dispositivo calcula las claves secretas $sk[i]$, $k[i]$ para el período actual utilizando sk y K_2. Luego $sk[i], k[i]$ son enviadas al cliente.
2. El cliente calcula $M \leftarrow \mathcal{E}nc(k[i], (D, \texttt{ts}))$ y el cliente envía M al servidor como parte del "`data message`".
3. El servidor verifica que el pedido fue firmado utilizando la clave pública correspondiente al identificador del usuario.
4. Si en el servidor existe una entrada $T[cid, did]$, se espera una confirmación de update en el período $i + 2$. Esta confirmación puede ser repetir el protocolo con las mismas entradas, pero con distinto `ts`.
5. El servidor envía una notificación de respaldo, $A =$(`backup`,cid,i,$H(M)$), y su firma $s = Sig(sk_s, A)$ al cliente. El servidor respalda los datos en $T[cid, did]$.
6. El cliente verifica la firma s del servidor.

Fase de Eliminación

- Input dispositivo cliente : La clave secreta maestra msk y la clave auxiliar K_2.
- Output dispositivos: Las claves secretas para el período actual $sk[i], k[i]$.
- Input cliente: El identificador de los datos a borrar did, el identificador del cliente cid.
- Output cliente: Mensaje " `release message`".

- Input servidor: Una tabla T en donde guarda los backups del cliente.
- Output servidor: T', la tabla T actualizada.

1. El dispositivo auxiliar genera las correspondientes claves para el período actual $sk_c[i]$ y $k[i]$ y las envía al cliente.
2. El cliente envía un `release message` al servidor.
3. El servidor verifica el mensaje y marca a $T[cid, did]$ como "`a borrar`".
4. El servidor espera una confirmación en el período subsiguiente. Si el cliente no envía confirmación, se elimina la etiqueta *a borrar* de la entrada $T[cid, did]$. Si el cliente envía confirmación en el período subsiguiente, entonces el servidor borra $T[cid, did]$.

Fase de Recuperación

- Input dispositivo cliente : La clave secreta maestra msk y la clave auxiliar K_2.
- Output dispositivos: Las claves secretas para el período actual $sk[i], k[i]$.
- Input cliente: No tiene.
- Output cliente: Mensaje `request_restore message`. Los datos D.
- Input servidor: Una tabla T en donde guarda los backups del cliente.
- Output servidor: M, el mensaje enviado por el cliente en la fase de backup.

1. El dispositivo auxiliar genera las correspondientes claves para el período actual $sk[i]$ y $k[i]$ y las envía al cliente.
2. El cliente espera hasta que recibe un mensaje de `announce` desde un servidor. El cliente envía un `request_restore message` al servidor.
3. El servidor envía los datos de vuelta al cliente.
4. El cliente verifica la autenticidad de los datos recibidos. Si falla, se abortar la comunicación con el servidor.

5. El cliente descifra los datos respaldados utilizando $k[i]$ y compara el `timestamp` con uno de algún dato recibido previamente. Si el nuevo `timestamp` es más reciente, los datos son guardados, si no se borran.

6. Si cualquier mensaje `announce` es recibido, repetir los pasos anteriores hasta que termine el período. Por simplicidad se supone que los `announce` serán enviados al menos una vez por período si los servidores son honestos.

Capítulo 3

Seguridad de protocolos

3.1. Seguridad bajo Universal Composability

Universal Composability (*UC*, Canetti 2000 [8]) es una herramienta de demostración de seguridad para protocolos criptográficos y aplicaciones. Esta herramienta permite definir las propiedades deseadas de protocolos criptográficos en términos de tareas o *funcionalidades*, capturando estas propiedades aún cuando la tarea se ejecute en composición y/o en forma concurrentemente con otras tareas. Una *funcionalidad* es un programa ejecutado por una "entidad confiable" que obtiene sus entradas directamente desde los participantes, ejecuta ciertas instrucciones sobre estas entradas y provee a cada participante su salida correspondiente. La funcionalidad también puede recibir entradas y dar salidas al adversario para modelar la "influencia permitida" del adversario en el cómputo de la tarea. Tal protocolo es llamado el *protocolo ideal* y la funcionalidad usada es la *funcionalidad ideal*. Informalmente hablando, un protocolo implementa en forma segura una determinada tarea si, ejecutando el protocolo con un adversario real, "emula" un *proceso ideal* en donde la tarea es computada por una funcionalidad que interactúa con los participantes y un adversario limitado (*adversario ideal*). El modelo asume que existe una tercera entidad, el ambiente $\mathcal{Z}$, que interactivamente entrega instrucciones a los participantes y obtiene sus salidas, posiblemente varias veces durante su ejecución. Además el ambiente interactúa libremente con el adversario, que puede ver todos los mensajes entre los participantes y corromper a un conjunto de ellos. El objetivo del ambiente es distinguir si la ejecución observada y controlada por él corresponde a la ejecución del protocolo en la "vida real" o si corresponde a un proceso ideal en donde se utiliza una funcionalidad ideal.

Una idea crucial detrás de este concepto es la definición de dos escenarios o mundos

(procesos); un mundo real que describe todas las interacciones entre los participantes en el protocolo y un mundo ideal (o antes llamado *proceso ideal*), el cual describe como debiera funcionar idealmente el sistema garantizando ciertas propiedades de seguridad. Si fuera posible demostrar que los dos mundos son de alguna manera equivalentes (o indistinguibles), entonces la descripción del mundo real es segura, pues todas las acciones que pueden tomar uno o un conjunto de participantes en este, se pueden realizar en un mundo ideal en donde se cumplen las garantías de seguridad. Más bien dicho, las acciones que puedan tomar algunos participantes están acotadas por la descripción del mundo ideal.

Ambos mundos son controlados por la entidad adversarial $\mathcal{Z}$, el ambiente, quien decide las entradas y operaciones para cada participante. En este esquema, además del ambiente y los participantes, se agrega un adversario (también controlado por $\mathcal{Z}$). Si después de la ejecución de $\mathcal{Z}$ en alguno de los dos mundos, este no puede distinguir en cual mundo se encuentra, entonces los dos mundos son *indistinguibles*.

Definición 1 *Una función* $\mu(k) : \mathbb{N} \to \mathbb{R}$ *es* ***despreciable*** *en* k *si* $\forall c \in \mathbb{N}\ \exists\ N > 0$ *tal que* $\forall k \geq N$

$$|\mu(k)| < \frac{1}{k^c}$$

Definición 2 *Sea* $k \in \mathbb{N}$ *un parámetro de seguridad. Dos mundos (o procesos), mundo 0 y mundo 1, son* ***computacionalmente indistinguibles*** *si para todo ambiente* $\mathcal{Z}$ *con entrada* x *que retorne un sólo bit, se cumple que* $|P_1[\mathcal{Z}(x) = 1 - P_0[\mathcal{Z}(x) = 1]]$ *es despreciable en* $k = |x|$. ***Notación:*** *Se denota a* $P_b[E]$ *a la probabilidad de un evento* E *en el espacio de probabilidades definido por el mundo* b.

Formalmente $\mathcal{Z}$ es un algoritmo probabilista de tiempo esperado polinomial, que retorna un sólo bit; 0 si fue ejecutado en el mundo real o 1 si fue ejecutado en el mundo ideal.

Intuitivamente un protocolo π es seguro bajo una funcionalidad $\mathcal{F}$ si los mundos `real` e `ideal` asociados son *computacionalmente indistinguibles*. A continuación se describe en forma general ambos escenarios.

Sea $\mathcal{P} = \{P_1, ..., P_n\}$ un conjunto de n participantes.

- **El mundo real:**

  Sea π el protocolo a ser ejecutado. El proceso de ejecución de π es parametrizado por un ambiente $\mathcal{Z}$ y un adversario $\mathcal{A}$.

  Inicialmente, el sistema consiste sólo de $\mathcal{Z}$ y $\mathcal{A}$. Durante la ejecución, $\mathcal{Z}$ puede ejecutar varias instancias de π e invoca a tantos participantes como quiera, determinando sus

identidades (compuesta por el par (pid, sid), en donde pid es el número del participante y sid es el número identificador de la instancia del protocolo). Estos participantes corren π_{sid}. Además $\mathcal{Z}$ puede dar inputs adicionales a los participantes durante el protocolo, y los participantes dan sus outputs a $\mathcal{Z}$ durante la ejecución de π. Adicionalmente, $\mathcal{Z}$ puede dar inputs a $\mathcal{A}$ cuantas veces quiera y puede obtener varios outputs de $\mathcal{A}$. Es decir, todas las entidades (participantes y el adversario) reciben varios inputs y dan varios outputs durante la ejecución del protocolo.

Una vez que un participante es activado, ya sea por el input o por un mensaje entrante, este sigue su código y potencialmente genera un mensaje de salida. Todos los mensajes de salida son vistos y manejados por el adversario (bajo la sola condición que no pueda bloquear mensajes por siempre). Los outputs son dados a $\mathcal{Z}$. Los participantes pueden invocar nuevas subrutinas (subprotocolos), ya sea que corran π u otro código. Sin embargo, estas subrutinas no se pueden comunicar con $\mathcal{Z}$ directamente.

Una vez que el adversario es activado, puede entregar mensajes a los participantes y generar outputs para $\mathcal{Z}$.

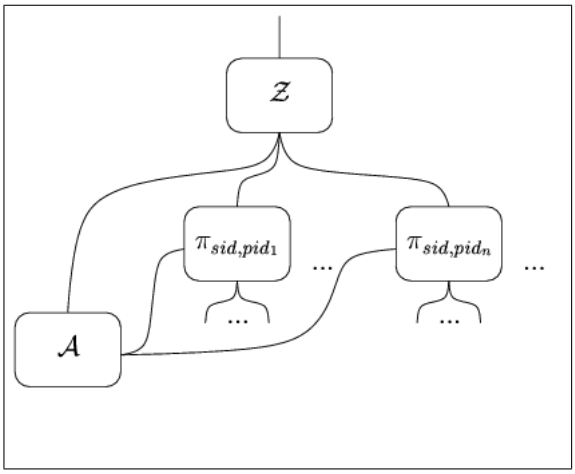

Figura 3.1: Mundo Real: El ambiente $\mathcal{Z}$ entrega los inputs y recibe los outputs de los participantes que corren el protocolo, mientras que el adversario $\mathcal{A}$ controla la comunicación. Adicionalmente, $\mathcal{Z}$ y $\mathcal{A}$ interactúan libremente.

- **El mundo ideal:** Recordar que una funcionalidad es simplemente un algoritmo PPT ejecutado por una entidad confiable. Este programa se comunica con los participantes y el adversario. Dada una funcionalidad ideal $\mathcal{F}$, se define el protocolo ideal $I_{\mathcal{F}}$ como sigue: Cuando un participante (pid, sid) corre $I_{\mathcal{F}}$ y obtiene un input, se lo da inmediatamente

a la funcionalidad $\mathcal{F}_{sid}$ ($\mathcal{F}_{sid}$ es una instancia específica de $\mathcal{F}$, aquella con número de sesión sid) . Cuando un participante recibe un output de $\mathcal{F}_{sid}$, se lo da inmediatamente a $\mathcal{Z}$. Es decir, en este mundo los participantes no ejecutan acciones, sólo reciben y dan mensajes del ambiente de la funcionalidad $\mathcal{F}$.

La funcionalidad $\mathcal{F}_{sid}$ ignora todos los input cuya instancia de protocolo no es sid.

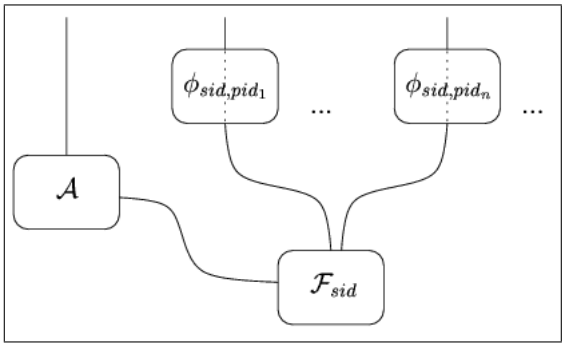

Figura 3.2: Mundo Ideal: Los participantes, que corren la instancia sid del protocolo ϕ, entregan sus inputs a la funcionalidad $\mathcal{F}_{sid}$. El adversario sólo se comunica con $\mathcal{F}_{sid}$

Convención: En ambos mundos se cumple que una entidad (ya sea el ambiente, el adversario o algún participante) puede generar un mensaje de salida sólo si este recibió un mensaje en su entrada. El primer mensaje lo da el ambiente. Para el resto de los participantes, el orden de recepción de los mensajes (llamado activación en [8]) está controlado por el adversario, quien despacha los mensajes uno a la vez. Con esto se puede definir un orden en los mensajes enviados en el protocolo. Notar que esto es suficiente para modelar concurrencia, dado que la entrega de los mensajes es controlada por el adversario y es este quien decide cuando, y en qué orden, entregar los mensajes.

Notación: Sea UC-EXEC$_{\pi,\mathcal{A},\mathcal{Z}}(x)$ la variable aleatoria que describe el output del ambiente $\mathcal{Z}$ cuando interactúa con el adversario $\mathcal{A}$ PPT en $|x|$, utilizando un input x, y corre el protocolo π. Se denota por UC-EXEC$_{\pi,\mathcal{A},\mathcal{Z}}$ a la familia $\{\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(x)\}_{x\in\{0,1\}^*}$

Definición 3 *Un protocolo π UC-emula un protocolo ϕ bajo adversarios en una clase $\mathcal{C}$ si para todo adversario $A \in \mathcal{C}$ existe un adversario S tal que para todo ambiente $\mathcal{Z}$ que entrega un solo bit, se cumple que:*

$$UC\text{-}EXEC_{\pi,\mathcal{A},\mathcal{Z}} \approx UC\text{-}EXEC_{\phi,\mathcal{S},\mathcal{Z}}$$

En donde $\approx$ denota indistinguibilidad computacional.

Definición 4 *Un protocolo π UC-realiza una funcionalidad $\mathcal{F}$ bajo adversarios en clase $\mathcal{C}$ si π UC-emula un protocolo ideal $I_{\mathcal{F}}$ para $\mathcal{F}$ bajo adversarios en clase $\mathcal{C}$.*

Composición de protocolos

Dados los protocolos ρ,π y ϕ, donde ρ llama a ϕ como subrutina. Entonces, $\rho^{\pi/\phi}$ denota el cambio de instancias de ϕ en ρ por π.

Este siguiente teorema (demostrado para diferentes modelos en [2, 8, 9, 11, 20] dice que si un protocolo π es seguro en el sentido de UC, entonces se puede ocupar en forma segura al componerlo[1] con cualquier otro protocolo.

Teorema 5 *Sea π y ϕ subrutinas de protocolos tal que π UC-emula ϕ. Entonces $\rho^{\pi/\phi}$ UC-emula ρ para cualquier protocolo ρ.*

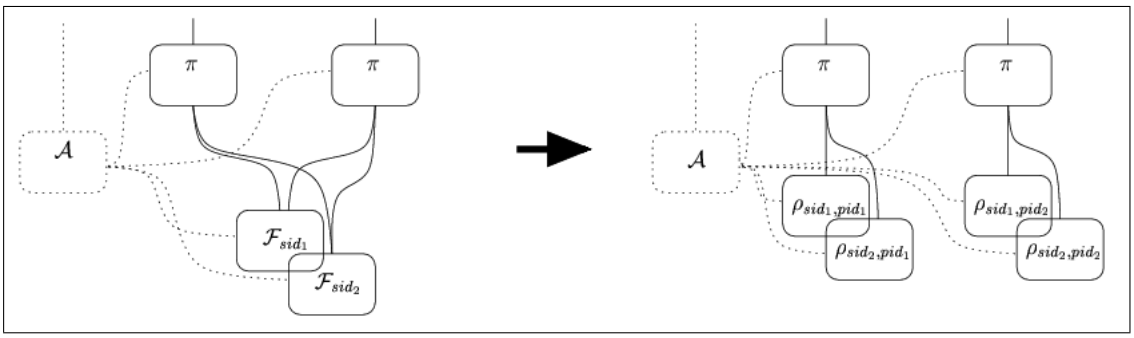

Figura 3.3: Composición universal para el caso en que el protocolo reemplazado es una funcionalidad $\mathcal{F}$. Cada instancia de $\mathcal{F}$ es reemplazada por una instancia del protocolo ρ. Las líneas sólidas representan inputs y outputs. Las líneas punteadas representan comunicación.

Modelo Híbrido

El modelo híbrido es una variación del modelo real antes definido. En este modelo el protocolo del mundo real puede llamar a instancias de funcionalidades ideales de otros protocolos. A estos protocolos se les llama *protocolos híbridos* pues son una mezcla de protocolos reales e ideales. Más precisamente, invocar una funcionalidad ideal es ejecutar un protocolo ideal para esa funcionalidad. Esto es, un protocolo $\mathcal{F}$-híbrido es un protocolo que incluye llamadas a $I_{\mathcal{F}}$ (el protocolo ideal para $\mathcal{F}$).

De modo de simplificar la demostración de seguridad de los protocolos descritos en este trabajo, se ocupará un modelo híbrido en donde el cálculo de los *cores* se realizarán llamando una funcionalidad $\mathcal{F}_{cores}$. De la misma manera, se puede invocar a una funcionalidad para el cálculo de la reputación de cada participante.

[1]Ejecutarlo en forma secuencial, concurrente, como subrutina, etc.

La principal idea del modelo híbrido, es poder separar tareas dejando ciertas subrutinas del protocolo como llamadas a funcionalidades. Luego se pueden construir protocolos para estas funcionalidades y argumentar su seguridad bajo la propiedad de composición de UC. Con esto, se puede cambiar el modelo híbrido por uno normal reemplazando las funcionalidades por protocolos reales.

Adversarios

Formalmente,los adversarios son algoritmos PPT interactivos. El conjunto de adversarios que cumplan una cierta propiedad p se pueden clasificar en una clase $\mathcal{C}_p$.

$$\mathcal{C}_p = \{A | A \text{ es PPT y cumple la propiedad } p\}$$

En general, es común y viable considerar adversarios que cumplan ciertas propiedades o limitaciones. Esto permite modelar los supuestos tomados en la descripción de los protocolos. Modelar la seguridad en las aplicaciones es muy difícil, o bien imposible en algunos casos, si se consideraran adversarios sin limitaciones. Además estos adversarios, en general, no corresponden a la vida real pues, dependiendo del ambiente de ejecución, estos tienen diferentes herramientas o poder computacional. Tal como se verá, en el caso de los protocolos propuestos en este trabajo, los adversarios considerados pertenecen a una clase determinada y luego se discute si esta clase de adversarios es razonable o no para el protocolo propuesto.

3.2. Seguridad del protocolo básico

Para demostrar bajo el paradigma UC que el protocolo básico, bajo los supuestos mencionados, cumple las propiedades de seguridad descritas en la sección 2.2, se debe definir cuidadosamente la funcionalidad $\mathcal{F}_{basic}$. Antes de definir la funcionalidad formalmente, se da una explicación intuitiva de las operaciones que esta realiza de acuerdo a los mensajes que recibe de los participantes y del adversario. Notar que los mensajes que esta funcionalidad entregue al adversario definiran la seguridad del sistema.

Se ocupará una funcionalidad adicional, $\mathcal{F}_{cores}$, que permite calcular *cores* para cada participante. Recordar que un *core* para un participante es un conjunto de participantes tales que, ante la infección de un virus o gusano, no pueden fallar todos. Esta funcionalidad puede recibir dos tipos de mensajes de los participantes: `init` y `core`. Cuando recibe `init` de un participante P, la funcionalidad escoge una configuración c para el participante P, la guarda en una tabla T ($T[P] \leftarrow c$) y le devuelve al participante su configuración. Al recibir un mensaje `core`, la funcionalidad escoge un *core* para P utilizando la herística de computo de *core* [17] sobre la tabla T.

$\mathcal{F}_{basic}$ puede recibir mensajes tanto de algún participante como del adversario. Como se asume que el adversario es pasivo, éste sólo puede bloquear o retrasar la entrega de algunos mensajes.

Los posibles mensajes que puede recibir $\mathcal{F}_{basic}$ de algún participante P_i son: `backup` para respaldar datos y `release` para borrar datos. Para el caso del protocolo básico se asume que el adversario no puede corromper arbitrariamente a los participantes, sólo los puede hacer fallar. Por lo tanto, el adversario sólo puede enviar el mensaje ($fail$,P_i) a $\mathcal{F}_{basic}$, que significa que P_i ya no tiene sus propios datos, ni los datos de los participantes que respaldaron en él. Para el caso del protocolo avanzado se puede eliminar este supuesto dejando al adversario con el poder de corromper a los participantes obteniendo sus claves privadas.

Cuando $\mathcal{F}_{basic}$ recibe un mensaje $M_{backup} = (\texttt{backup}, D, did)$ desde algún participante P_i, se debe pedir a $\mathcal{F}_{cores}$ que calcule un *core* para P_i. Luego de obtener el *core* ($\texttt{Core}_{(P_i,did)}$), se genera un mensaje hacia el adversario $\mathcal{S}$, el cual contiene $\texttt{Core}_{(P_i,did)}$, el identificador de P_i, $|D|$ y did. Con esto se asegura que el adversario no puede aprender más que el *core* y el largo de los datos (y no aprende nada sobre el contenido de los datos). Luego la funcionalidad guarda en una tabla $T_{backups}[P_i, did]$ los datos del participante y la descripción del conjunto

$\texttt{Core}_{(P_i,did)}$. Notar que si el adversario es dinámico, puede decidir corromper a todos los participantes de un *core*, lo cual afectaría a la propiedad de disponibilidad. Para evitarlo, en este análisis se supone que $\mathcal{F}_{cores}$ elige el *core* de manera que el adversario no pueda afectar a todos los integrantes de un *core*[2].

Al recibir un mensaje `release`, $\mathcal{F}_{basic}$ debe obtener y borrar el *core* y los datos almacenados en la tabla $T_{backpus}$. Además, envía a $\mathcal{S}$ el mensaje de *release* que contiene el identificador de P_i y el identificador de los datos.

Si se recibe un mensaje ($fail$,P_i) del adversario, se debe iniciar una fase de recuperación para el participante P_i. Para todos los datos con diferente *did* que P_i ha respaldado, verificar que exista algún participante que pertenezca al $\texttt{Core}_{(P_i,did)}$ que no haya fallado después de la hora en la que se hizo el backup. Además se debe guardar en una tabla T_{fail} la hora en que ocurrió la falla.

Cuaquier otro mensaje que se reciba debe ser ignorado.

Funcionalidad $\mathcal{F}_{basic}$:

- Cuando se reciba un mensaje $M_P = (t, ...)$ de un participante P:
  - Si t es `backup`:
    1. Obtener de M_P los datos D y su identificador *did*.
    2. Llamar a $\mathcal{F}_{cores}$ para obtener un *core* para P.
    3. Enviar a $\mathcal{S}$ la tupla ("`backup`",P,*did*,$|D|$,$\texttt{Core}_P$).
    4. Cuando $\mathcal{S}$ responde `OK`, entonces $T_{backups}[P, did] \leftarrow (\texttt{Core}_P, D)$.
  - Si t es `release`:
    1. Obtener de M_p el identificador de los datos a borrar *did*.
    2. $(\texttt{Core}_P, D) \leftarrow T_{backups}[P, did]$.
    3. Enviar a $\mathcal{S}$ la tupla ("`release`",P,*did*,$\texttt{Core}_P$).
    4. Cuando $\mathcal{S}$ responde `OK`, entonces eliminar $T_{backups}[P, did]$.
- Cuando se reciba un mensaje $M_{\mathcal{S}}$ =(`fail`,P) del adversario $\mathcal{S}$.
  1. $T_{fail}[P] \leftarrow$ `ts`.
  2. Enviar a P un mensaje `fail`.
  3. Sea $L = \{did|$ did en el identificador de un conjunto de datos respaldados por $P\}$. $\forall did \in L$, entregar el mensaje (`to-restore`,P, *did*) al adversario. Al recibir cada respuesta (`OK`,*did*), Recuperar los datos identificados por *did*, esto es, $(\texttt{Core}_{P,did}, D) \leftarrow T_{backups}[P, did]$ y enviar (`restore`,P,*did*,D) a P.

Figura 3.4: Funcionalidad $\mathcal{F}_{basic}$: Notación: `ts`: hora local o timestamp.

[2]Este supuesto refleja el modo de operación de gusanos y virus en internet.

En la figura 3.4 se muestra una funcionalidad que provee al adversario muy poca información relevante. Cuando recibe un mensaje de backup, el adversario sólo puede aprender quien fue el emisor, el *core*, el identificador de los datos y el largo de los datos. Recordar que la funcionalidad no provee los datos al adversario.

Lo que sí aprende el adversario es el tipo de mensaje que se envían todos los participantes, por lo que sabe quién solicitó `backup` o `release`.

Definición 6 *La clase de adversarios $\mathcal{C}_{fail}$ corresponde al conjunto de algoritmos PPT interactivos que sólo hacen fallar a los participantes borrando todos sus datos.*

Los adversarios pertenecientes a $\mathcal{C}_{fail}$ no corrompen a los participantes. Es decir, no obtienen sus claves privadas y por ende, no pueden realizar operaciones maliciosas. Lo único que pueden hacer estos adversarios es borrar los datos del participante afectado y reiniciarlos. Más aún, al reinicar el participante, este vuelve a ser honesto.

Teorema 7 *El protocolo descrito en la sección 2.3 UC-realiza la funcionalidad $\mathcal{F}_{basic}$ en el modelo $\mathcal{F}_{cores}$-híbrido bajo adversarios en $\mathcal{C}_{fail}$.*

Demostración

Para demostrar el teorema anterior se debe probar que

$$\forall \text{ ambiente } \mathcal{E},\ \forall \text{ adversario } \mathcal{A} \in \mathcal{C}_{fail},\ \exists \text{ adversario } \mathcal{S} \text{ tal que}$$

$$\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \text{UC-EXEC}_{I_{\mathcal{F}},\mathcal{S},\mathcal{E}}$$

Por claridad se resumen todas las acciones que el ambiente $\mathcal{E}$ puede realizar. Los participantes y el adversario obedecen a $\mathcal{E}$, quien puede solicitar las siguientes instrucciones:

- Hacer backup (mensaje `backup`): $\mathcal{E}$ le pide a algún participante P que haga backup de ciertos datos especificados por $\mathcal{E}$.
- Desechar backup (mensaje `release`): $\mathcal{E}$ le pide a algún participante P que deseche algún backup hecho.
- Fallar (mensaje (`fail`,P)): $\mathcal{E}$ le pide al adversario que el participante P falle. Es decir, P_i ya no sabe que datos respaldó, quien los podría tener, ni los datos que respaldó de otros usuarios.

La idea principal es que se debe construir un adversario ideal $\mathcal{S}$ (*Simulador*), que ocupe $\mathcal{A}$. Este adversario es utilizado por la entidad confiable $\mathcal{F}_{basic}$ para simular la ejecución del

protocolo π bajo $\mathcal{A}$, de manera que $\mathcal{E}$ crea que en verdad se encuentra en un mundo real. El simulador propuesto se muestra en la figura 3.5. Informalmente, este simulador funciona de la siguiente manera:

- Ejecuta $\mathcal{A}$ rescatando todas las salidas de este adversario y dándoselas como entrada a $\mathcal{F}_{basic}$.
- Todos los mensajes que genere $\mathcal{A}$ hacia $\mathcal{E}$ son enviados a $\mathcal{E}$ y los mensajes generados por $\mathcal{A}$ para hacer fallar a un participante (mensaje (`fail`,P)) son entregados a $\mathcal{F}_{basic}$.
- Genera aleatoriamente las claves para cada nuevo participante y las guarda en tablas.
- Cada vez que se realiza un backup se encriptan datos falsos, se firma este mensaje falso de backup y se le entrega a $\mathcal{A}$ tantos mensajes como participantes tenga el *core*.
- Para cada operación (`backup` o `release`) se entrega un mensaje al adversario. Apenas el adversario responde se le entrega `OK` a $\mathcal{F}_{basic}$.
- Cuando $\mathcal{F}_{basic}$ avisa que un participante ha fallado, se debe simular la fase de recuperación, generando mensajes de `announce`, `request restore` y `restore` falsos entregándoselos al adversario en el orden correcto. Cuando un mensaje de restore es contestado por $\mathcal{A}$ se avisa a $\mathcal{F}_{basic}$ que puede entregar los datos al participante que ha fallado.

Dado el simulador de la figura 3.5, se debe demostrar que si el esquema de cifrado simétrico utilizado ($\mathcal{SE}$) es seguro en el sentido *ind-cpa*[3] y el esquema de firmas utilizado ($\mathcal{DS}$) es *uf-cma*[4], entonces de cumple que

$$\forall \text{ ambiente } \mathcal{E},\ \forall \text{ adversario } \mathcal{A} \in \mathcal{C}_{fail}$$

$$\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \text{UC-EXEC}_{I_{\mathcal{F}},\mathcal{S},\mathcal{E}}$$

Por contradicción,

$$\text{Si } \exists \text{ ambiente } \mathcal{E} \wedge\ \exists \text{ adversario } \mathcal{A} \in \mathcal{C}_{fail} \text{ tal que}$$

$$\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}} \not\approx \text{UC-EXEC}_{I_{\mathcal{F}},\mathcal{S},\mathcal{E}},$$

entonces $\mathcal{SE}$ no es *ind-cpa* seguro o $\mathcal{DS}$ no es *uf-cma* seguro.

[3]Indistinguibilidad ante ataques de texto plano escogido, ver apéndice B .
[4]No falsificación de firmas ante ataques de mensajes escogidos, ver apéndice C .

Simulador $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{basic}$:

- Al iniciarse, generar las claves para cada participante vía los algoritmos $\mathcal{K}(\cdot)$ y $\bar{\mathcal{K}}(\cdot)$ guardándolas en las tablas TK, TPK y TSK.
- Cualquier mensaje que genere $\mathcal{A}$ al ambiente $\mathcal{E}$ se entrega directamente al ambiente.
- Para los mensajes que genere $\mathcal{A}$ a algún participante P:
  - Si el mensaje es `fail`: entregárselo a $\mathcal{F}$.
  - Si el mensaje no es `fail`: Comprobar la firma del mensaje. Si verifica abortar, sino, ignorar.
- Cuando reciba un mensaje $M_{\mathcal{E}}$ del ambiente $\mathcal{E}$, entregárselo a $\mathcal{A}$.
- Cuando recibe un mensaje $M_{\mathcal{F}}$ de la funcionalidad $\mathcal{F}$.
  - Si $M_{\mathcal{F}}$ =(`backup`,P,did,ℓ,$\texttt{Core}_P$):
    1. $d \xleftarrow{\$} \{0,1\}^{\ell}$
    2. Calcular C como $\mathcal{E}nc(TK[P], d)$.
    3. Sea $R \leftarrow (C \circ did \circ TK[P] \circ P)$.
    4. Computar la firma de R como $s \leftarrow Sig(TSK[P], R)$ y definir M como $R \circ s$.
    5. $\forall\ Q \in \texttt{Core}_P,\ T[P, did, Q] \leftarrow M$.
    6. $\forall\ Q \in \texttt{Core}_P$, entregar M a $\mathcal{A}$.
    7. Cuando $\mathcal{A}$ responda, entregar `OK` a $\mathcal{F}$.
  - Si $M_{\mathcal{F}}$ =(`release`,P,did,$\texttt{Core}_P$):
    1. $\forall\ Q \in \texttt{Core}_P$, entregar (`release`,$P$,$did$) a $\mathcal{A}$.
    2. Cuando todas las llamadas a $\mathcal{A}$ respondan, entregar `OK` a $\mathcal{F}$.
  - Si $M_{\mathcal{F}} = (\texttt{to-restore}, P, did)$
    1. Generar mensajes `announce` simulando que lo crearon los participantes pertenecientes a $\texttt{Core}_{P,did}$ que no han fallado. Entregar estos mensajes a $\mathcal{A}$.
    2. Cada vez que $\mathcal{A}$ responda, para todo $Q \in \texttt{Core}_{P,did}$ generar un mensaje `request restore` simulando que lo creo el participante P hacia un participante Q y entregárselo a $\mathcal{A}$.
    3. Cuando se reciba la respuesta del `request restore`, entregar $T[P, did, Q]$ a $\mathcal{A}$ como un mensaje de `restore`.
    4. Al recibir la respuesta de $\mathcal{A}$, entregar $(\texttt{OK}, did)$ a $\mathcal{F}$.

Figura 3.5: Simulador $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{basic}$.

Se debe notar que el simulador $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{basic}$ de la figura 3.5 y la funcionalidad $\mathcal{F}_{basic}$ son tal que envían la misma cantidad y tipo de mensajes que el ambiente y el adversario esperan recibir en el mundo real. Por ende, si el ambiente $\mathcal{E}$ distingue en que mundo se encuentra, es por el hecho que el adversario puede, o bien distinguir entre el cifrado de un mensaje aleatorio y el cifrado de un mensaje real, o bien falsificar una firma, o ambos.

Para ello, se construye un adversario que quiebre $\mathcal{SE}$ (llamado $B_{\mathcal{SE}}$) y otro que quiebre $\mathcal{DS}$ (llamado $B_{\mathcal{DS}}$). Luego mostraremos que si el ambiente distingue en que mundo se está ejecutando, entonces algunos de los dos adversarios debe triunfar en su respectivo experimento.

El adversario $B_{\mathcal{SE}}$ que intenta quebrar $\mathcal{SE}$ en el sentido *ind-cpa* ocupa como subrutina el siguiente algoritmo, parametrizado por $i \in \{1, .., n\}$.

$B^i_{\mathcal{SE}}$: Simula la ejecución del ambiente $\mathcal{E}$ interactuando con el adversario $\mathcal{A}$, jugando el rol de los n participantes. Durante la ejecución interviene todos los mensajes entre los participantes que involucren el cifrado de algún texto plano M. Para los primeros $i-1$ participantes, entregar el cifrado de un mensaje aleatorio ($\mathcal{E}_{K_j}(\$)$, donde K_j es la clave de encriptación usada por el j-ésimo participante). Para los últimos $n-i$ participantes, entregar el cifrado del mensaje original ($\mathcal{E}_{K_j}(M)$). Para el i-ésimo participante entregar la salida del oráculo de encriptación $C = \mathcal{E}_{K_i}(LR(M, \$, b))$ con entrada el mensaje original M y un mensaje aleatorio[5]. Todos lo mensajes deben ser guardados para poder recuperarlos en caso de que el ambiente solicite un *restore* a algún participante, al finalizar la ejecución entrega como salida, la salida de $\mathcal{E}$.

Notar que $B^0_{\mathcal{SE}}$ ejecuta $\mathcal{E}$ exactamente como en el mundo real. Además si $B^i_{\mathcal{SE}}$ es ejecutado en el mundo donde el oráculo encripta siempre el mensaje aleatorio ($b = 1$), entonces $B^n_{\mathcal{SE}}$ (con n la cantidad de participantes) ejecuta $\mathcal{E}$ simulando el mundo ideal, siempre cuando el adversario $\mathcal{A}$ no falsifique ninguna firma. Otro punto importante es que $B^i_{\mathcal{SE}}$, ejecutado con el oráculo de encriptación con $b = 0$, es igual a $B^{i-1}_{\mathcal{SE}}$, pero ejecutado con el oráculo de encriptación con $b = 1$. Por lo tanto, se tiene la siguiente relación:

$$P[B^i_{\mathcal{SE}} = d | b = 0] \quad = \quad P[B^{i-1}_{\mathcal{SE}} = d | b = 1],\ d \in \{0, 1\} \tag{3.1}$$

Ahora que está definido $B^i_{\mathcal{SE}}$ podemos definir el adversario $B_{\mathcal{SE}}$ como sigue:

Adversario 1: $B_{\mathcal{SE}}(k)$

1 $\ell \xleftarrow{\$} \{1..n\}$

2 $g \leftarrow B^\ell_{\mathcal{SE}}(k)$

3 output g

Ahora describiremos un adversario $B_{\mathcal{DS}}$ que quiebra $\mathcal{DS}$ en el sentido *uf-cma*. Este adversario utiliza el siguiente algoritmo como subrutina, parametrizado por $i \in \{1, .., n\}$.

$B^i_{\mathcal{DS}}(pk, k)$: Simula el ambiente $\mathcal{E}$ interactuando con el adversario $\mathcal{A}$, jugando el rol de los n participantes. Para todos los participantes, exceptuando el i-ésimo, crea las claves de firma ocupando $\bar{\mathcal{K}}(\cdot)$. Para el i-ésimo participante, asigna la clave pública pk. Durante la ejecución

[5]Ver apéndice B .

de $\mathcal{E}$ con el adversario $\mathcal{A}$ y los n participantes, $B^i_{\mathcal{DS}}$ interviene los mensajes debiera firmar el participante P_i, entregándole el mensaje al oráculo de firma. La salida de este oráculo es entregada como la firma del participante P_i. Además este mensaje es guardado en una tabla mantenida por $B^i_{\mathcal{DS}}$. Si alguna vez, sucede que el adversario $\mathcal{A}$ entrega un par (M, σ), válido en donde M no fue guardado, entonces $B^i_{\mathcal{DS}}$ entrega (M, σ) como salida. Sino, $B^i_{\mathcal{DS}}$ retorna $(\perp, \perp)$.

El adversario $B_{\mathcal{DS}}$ es el siguiente:

Adversario 2: $B_{\mathcal{DS}}(pk, k)$

1 $\ell \xleftarrow{\$} \{1..n\}$

2 $(M, \sigma) \leftarrow B^\ell_{\mathcal{DS}}(pk, k)$

3 output (M, σ)

A continuación se muestra que existe un ambiente $\mathcal{E}^*$ y un adversario $\mathcal{A}^*$ tales que $\mathcal{E}^*$ puede distinguir con probabilidad no despreciable entre un mundo real (mundo 0) bajo $\mathcal{A}^*$ y el mundo ideal (mundo 1) bajo $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{basic}$, entonces alguno de los adversarios anteriores ($B_{\mathcal{SE}}$ o $B_{\mathcal{DS}}$) tiene una ventaja no despreciable en su respectivo experimento.

Sea $\epsilon(|x|)$ la probabilidad de que $\mathcal{E}$ distinga en que mundo se encuentra dado que fue ejecutado con un input $x \in \{0,1\}^*$. Por lo tanto

$$\epsilon(k = |x|) = P_1[\mathcal{E}(x) = 1] - P_0[\mathcal{E}(x) = 1]$$

Donde el subíndice 0 representa ejecución en el mundo real, y el subíndice 1 en el mundo ideal. Sea F el evento "$\mathcal{S}^{\mathcal{FA}}_{basic}$ aborta" y μ la probabilidad de que ocurra el evento F. Recordar que $\mathcal{S}^{\mathcal{FA}}_{basic}$ aborta sólo si $\mathcal{A}$ entrega una firma válida.

Desarrolando el término $P_1[\mathcal{E}(x) = 1]$

$$\begin{aligned} P_1[\mathcal{E}(x) = 1] &= P_1[\mathcal{E}(x) = 1|F] \cdot \mu + P_1[\mathcal{E}(x) = 1|\overline{F}] \cdot (1 - \mu) \\ &\leq \mu + P_1[\mathcal{E}(x) = 1|\overline{F}] \end{aligned}$$

Por lo tanto,

$$\epsilon(k) \leq \mu + P_1[\mathcal{E}(x) = 1|\overline{F}] - P_0[\mathcal{E}(x) = 1] \tag{3.2}$$

A continuación se analiza la ventaja del adversario $B_{\mathcal{SE}}$ en el experimento del apéndice B .

$$Adv^{ind\text{-}cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) = 2 \cdot P[Exp^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) = 1] - 1$$

$$\begin{aligned}
P[Exp^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) = 1] &= P[b \xleftarrow{\$} \{0,1\},\ B_{\mathcal{SE}}(k) = b] \\
&= \frac{1}{n}\sum_{i=1}^{n} P[b \xleftarrow{\$} \{0,1\},\ B^i_{\mathcal{SE}}(k) = b] \\
&= \frac{1}{2n}\sum_{i=1}^{n}(P[B^i_{\mathcal{SE}}(k) = 1|b = 1] + P[B^i_{\mathcal{SE}}(k) = 0|b = 0]) \\
&= \frac{1}{2n}\sum_{i=1}^{n}(P[B^i_{\mathcal{SE}}(k) = 1|b = 1] + 1 - P[B^i_{\mathcal{SE}}(k) = 1|b = 0]) \\
&= \frac{1}{2} + \frac{1}{2n}\sum_{i=1}^{n}(P[B^i_{\mathcal{SE}}(k) = 1|b = 1] - P[B^i_{\mathcal{SE}}(k) = 1|b = 0])
\end{aligned}$$

Ocupando la ecuación 3.1 se obtiene:

$$\begin{aligned}
P[Exp^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) = 1] &= \frac{1}{2} + \frac{1}{2n}\sum_{i=1}^{n}(P[B^i_{\mathcal{SE}}(k) = 1|b = 1] - P[B^{i-1}_{\mathcal{SE}}(k) = 1|b = 1]) \\
&= \frac{1}{2} + \frac{1}{2n}P[B^n_{\mathcal{SE}}(k) = 1|b = 1] - P[B^0_{\mathcal{SE}}(k) = 1|b = 1]) \\
&= \frac{1}{2} + \frac{1}{2n}P[B^n_{\mathcal{SE}}(k) = 1|b = 1] - P[B^0_{\mathcal{SE}}(k) = 1])
\end{aligned}$$

Por lo tanto la ventaja del adversario $B_{\mathcal{SE}}$ en el experimento *ind-cpa* es:

$$Adv^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) = \frac{1}{n} \cdot (P[B^n_{\mathcal{SE}}(k) = 1|b = 1] - P[B^0_{\mathcal{SE}}(k) = 1]) \tag{3.3}$$

Si el adversario $\mathcal{A}$ no falsifica ninguna firma, entonces $P[B^n_{\mathcal{SE}}(k) = 1|b = 1]$ es exactamente $P_1[\mathcal{E}(x) = 1|\overline{F}]$ y $P[B^0_{\mathcal{SE}}(k) = 1])$ es $P_0[\mathcal{E}(x) = 1]$, por lo tanto se puede reemplazar la ecuación 3.3 en la ecuación 3.2:

$$\epsilon(k) \leq \mu + n \cdot Adv^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) \tag{3.4}$$

Por otro lado la ventaja del adversario $B_{\mathcal{DS}}$ es:

$$
\begin{aligned}
Adv^{uf-cma}_{\mathcal{DS},B_{\mathcal{DS}}}(k) &= P[Exp^{uf-cma}_{\mathcal{DS},B_{\mathcal{DS}}}(k) = 1] \\
&= P[B_{\mathcal{DS}}(k) \text{ entregue firma válida }] \\
&= \frac{1}{n} P[\mathcal{S}^{\mathcal{FA}}_{basic} \text{ aborte }] \\
&= \frac{1}{n} P[F] \\
&= \frac{1}{n} \mu
\end{aligned}
$$

Reemplazando la ecuación anterior en la ecuación 3.4:

$$
\begin{aligned}
\epsilon(k) &\leq n \cdot Adv^{uf-cma}_{\mathcal{DS},B_{\mathcal{DS}}}(k) + n \cdot Adv^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k) && (3.5) \\
&= n \cdot (Adv^{ind-cpa}_{\mathcal{DS},B_{\mathcal{DS}}}(k) + Adv^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k)) && (3.6)
\end{aligned}
$$

De la ecuación anterior se puede concluir que si $\epsilon(|x|)$ no es despreciable, entonces necesariamente tiene que ocurrir que, o bien, $Adv^{uf-cma}_{\mathcal{DS},B_{\mathcal{DS}}}(k)$ es no despreciable en k, o bien, $Adv^{ind-cpa}_{\mathcal{SE},B_{\mathcal{SE}}}(k)$ es no despreciable en k.

Por lo tanto, si el protocolo descrito en la sección no es seguro bajo UC, entonces el esquema de firmas no es seguro en el sentido *uf-cma*, o bien, el esquema de cifrado no es seguro en el sentido *ind-cpa*, o bien, ambos no son seguros. Por lo tanto se alcanza la contradicción.

Por contradicción se concluye que el ambiente $\mathcal{E}$ no puede distinguir en que mundo se encuentra, pues se supone que ni el esquema de firmas ni el esquema de cifrado son quebrables en el sentido *uf-cma* y *ind-cpa* respectivamente.

Observaciones

Para modelar seguridad en el protocolo básico, se utilizó adversarios pertenecientes a $\mathcal{C}_{fail}$. Esto es, adversarios que no pueden corromper y obtener las claves privadas de los participantes. Por esto, $\mathcal{E}$ no puede adivinar en cual mundo se encuentra. De hecho, si el adversario pudiera corromper, es fácil poner un ejemplo en donde $\mathcal{E}$ adivine con probabilidad 1, es cosa de que $\mathcal{E}$ le pida a $\mathcal{A}$ que corrompa a un participante P y luego pedirle a $\mathcal{A}$ que ejecute funciones como si fuera P. Para los casos donde no es factible suponer el tipo

de corrupción del adversario, el protocolo avanzado propuesto en la sección 2.4 debiera ser utilizado. Éste permite mantener las propiedades la seguridad aún cuando el adversario tenga la habilidad se corromper a los participantes obteniendo sus claves privadas.

Notar que la propiedad de disponibilidad de los datos se basa en que el adversario no puede hacer fallar a todos los participantes de un *core*.

La privacidad se muestra claramente en el simulador, pues este simula el protocolo entregando mensajes al adversario sin conocer los datos reales.

Con respecto a la integridad de los datos, esta es claramente preservada. Esto es evidente de la descripción de la funcionalidad $\mathcal{F}_{basic}$, la cual no permite al adversario modificar los backups. Junto con el hecho que el protocolo básico UC-emula $\mathcal{F}_{basic}$, ningún adversario, aún en el mundo real, puede afectar la integridad de los backups.

3.3. Seguridad del protocolo avanzado

Para el protocolo avanzado se requiere de una funcionalidad extra llamada $\mathcal{F}_{SYN}$, que permite emular el cambio de un período a otro. Esta funcionalidad es una versión simplificada de la presentada por Canetti en [8][6]. Esta fucionalidad permite modelar en UC la sincronización entre los participantes. Cuando un participante desee avanzar al siguiente período, deberá consultar a $\mathcal{F}_{SYN}$.

En este caso las operaciones que puede solicitar el ambiente $\mathcal{E}$ a los participantes son:

- Backup: Solicita al participante que inicie una fase de backup de un cierto conjunto de datos.
- Release: Solicita al participante que libere un cierto conjunto datos.
- Next: Solicita al participante que le pida a la funcionalidad $\mathcal{F}_{SYN}$ un cambio de período. En este caso el participante queda bloqueado hasta que $\mathcal{F}_{SYN}$ le responda.
- Confirm Backup: Solicita al participante que confirme la solicitud de backup.
- Confirm Release: Solicita al participante que confime la solicitud de release.

Las operaciones que puede solicitar $\mathcal{E}$ al adversario son: *fail* y *corrupt* para hacer fallar y corromper a algún participante respectivamente.

De modo de simplificación la funcionalidad $\mathcal{F}_{SYN}$ será incorporada en la funcionalidad $\mathcal{F}_{enhanced}$. Notar que en el mundo real esta funcionalidad de sincronización será una entidad independiente.

A continuación se describe informalmente la funcionalidad $\mathcal{F}_{enhanced}$. Al activarse inicializa el período actual t en 1 y el conjunto de participantes corruptos como vacío.

Al recibir una solicitud de backup de un participante P, se le consulta a $\mathcal{F}_{core}$ por un nuevo *core* para P. Al igual que F_{basic} se entrega al adversario ideal un mensaje que no incluye los datos. Cuando el adversario ideal responde, los datos y el *core* se guardan en una tabla temporal de backups no confirmados, T_{BNC}. Cualquier otro mensaje de backup de ese participante es ese período y en el sigiuente es ignorado. De manera similar, una solicitud

[6]Ver apéndice E .

de *release* se entrega al adversario ideal una solicitud de release. Cuando el adversario ideal responde, marcar en la tabla T_{to_erase} de los datos *did* a borrar.

Al recibir una petición de cambio de período de un participante (mensaje `Next`), se marca ese participante como listo para el próximo período y se le comunica al adersario la petición. El período se cambia si el adversario solicita cambio de período y todos los participantes no corruptos ya lo pidieron o solicitaron backup o *relesase* en este período o en el anterior.

Cuando se inicia un nuevo período, los participantes que solicitaron backup (resp. *release*) en el período anteanterior, pueden enviar mensajes de confirmación de backup (resp. *release*). En cualquiera de los dos casos, antes de realizar la operación se le envía el mensaje al simulador.

Cuando se recibe un mensaje del adversario haciendo fallar a un participante, la funcionalidad actúa de la misma manera que la funcionalidad $\mathcal{F}_{basic}$.

Cuando se recibe un mensaje del adversario corrompiendo a un participante, entonces se debe incluir este participante al conjunto de corruptos e ignorar todos los mensajes de este participante durante el presente período y en el siguiente.

Cualquier mensaje de backup o *release* del adversario controlando las acciones de un participante debe ser ignorado.

En la figura 3.6 se muestra una funcionalidad que, al igual que la funcionalidad $\mathcal{F}_{basic}$, provee al adversario muy poca información relevante. Por lo que $\mathcal{A}$ no puede conocer los datos ni alterarlos.

Definición 8 *La clase de adversarios $\mathcal{C}_{fail-corrupt^*}$ corresponde al conjunto de algoritmos PPT interactivos que corrompen de las siguientes dos maneras:*

- *Hacen fallar un participante borrando todos sus datos y reiniciándolos.*
- *Corrompen un participante obteniendo sus claves privadas, pero no pueden hacerlo durante tres perídos de tiempo consecutivos.*

Teorema 9 *El protocolo descrito en la sección 2.4 UC-realiza la funcionalidad $\mathcal{F}_{enhanced}$ en el modelo $(\mathcal{F}_{SYN}, \mathcal{F}_{cores})$-híbrido bajo adversarios de clase $\mathcal{C}_{fail-corrupt^*}$.*

Demostración

El adversario ideal (o simulador) para este caso funciona de forma similar a $\mathcal{S}_{simple}$, salvo que ahora debe actualizar las claves para cada participante al inicio de un nuevo período y firmar los mensajes de confirmación que lleguen desde la funcionalidad. Formalmente el simulador se muestra en la figura 3.7.

Funcionalidad $\mathcal{F}_{enhanced}$:

- Al momento de activación inicializa la variable t en 1 (primer período de tiempo), y se avisa a todos los participantes.
- Ignorar todos los mensajes de participantes que solicitaron `backup` o `release` en el período anterior.
- Cuando se reciba un mensaje $M_P = (type, ...)$ de un participante P:
  - Si $P \in \mathcal{C}$, ignorar mensaje.
  - Si *type* es `backup`:
    1. Obtener de M_P los datos D y su identificador *did*.
    2. Ocupar $\mathcal{F}_{cores}$ para calcular un *core* para P.
    3. Enviar a $\mathcal{S}$ la tupla (`backup`,P_i,*did*,$|D|$,$\texttt{Core}_P$) y esperar respuesta.
       a) Si $T_{BNC}[P_i, did, t] = \perp$, entonces $T_{BNC}[P_i, did, t] = \texttt{Core}_{p_i} \circ D$
       b) Sino ignorarlo.
  - Si *type* es `release`:
    1. Obtener de M_P el identificador de los datos a borrar *did*.
    2. $Core_P \circ D \leftarrow T_{backup}[P, did]$.
    3. Enviar a $\mathcal{S}$ la tupla (`release`,P_i,*did*,$\texttt{Core}_P$).
    4. Cuando $\mathcal{S}$ responde `OK`, entonces $T_{to\_erase}[P, t] \leftarrow did$.
  - Si *type* es `next` (funcionalidad $\mathcal{F}_{SYN}$)
    1. Enviar `next`-P al adversario.
    2. Cuando $\mathcal{S}$ $T_{next}[P] \leftarrow true$
  - Si *type* es `Confirm backup`
    1. Enviar (`Confirm backup`,P,*did*) a $\mathcal{S}$ y esperar respuesta.
    2. Si existe la entrada $T_{BNC}[P, did, t-1]$, entonces
       $T_{backup}[P, did] \leftarrow T_{BNC}[P, did, t-1]$
       y borrar $T_{BNC}[P, did, t-1]$.
  - Si *type* es `Confirm release`
    1. Enviar (`Confirm release`,P,*did*) a $\mathcal{S}$ y esperar respuesta.
    2. Si $T_{toerase}[P, t-1] = did$, eliminar $T_{backup}[P, did]$.
- Cuando se reciba un mensaje $M_{\mathcal{S}}$ =(`fail`,P) reciba un mensaje del adversario $\mathcal{S}$.
  1. $T_{fail}[P] \leftarrow \texttt{ts}$.
  2. Enviar a P un mensaje `fail`.
  3. Sea $D = \{did|$ did en el identificador de un conjunto de datos respaldados por $P\}$. $\forall did \in D$, entregar el mensaje (`to-restore`,P, *did*). Al recibir cada respuesta `OK`-*did*, enviar los datos identificados por *did* a P.
- Al recibir un mensaje $M_{\mathcal{S}}$ =(`corrupt`,P) del adversario $\mathcal{S}$, $\mathcal{C} \leftarrow \mathcal{C} \cup \{P, t\}$.
- Al recibir `Next` del adversario, (funcionalidad $\mathcal{F}_{SYN}$)
  1. Si $\exists P \notin \mathcal{C}$ tal que $T_{next} = false \wedge T_{BNC}[P, *, t-1] = \perp$, ignorar.
  2. Sino
     $\forall\ P\, T_{next}[P] \leftarrow false$,
     $t \leftarrow t+1$, $\mathcal{C} \leftarrow \{(P', T') \in \mathcal{C} | t' = t-1\}$
     y avisar a todos los participantes el cambio de período.

Figura 3.6: Funcionalidad $\mathcal{F}_{enhanced}$: Notación: `ts`: hora local o timestamp.

Simulador $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{enhanced}$:

- Al inicializar, elegir aleatoriamente las claves de todos los participantes para el primer período y guardarlas en las tablas TK, TPK y TSK.
- Cualquier mensaje que genere $\mathcal{A}$ al ambiente $\mathcal{E}$ se entrega directamente al ambiente.
- Los mensajes `fail` o `corrupt` que genere $\mathcal{A}$ a algún participante P, entregárselo a $\mathcal{F}$. Para los demás mensajes de $\mathcal{A}$ hacia un participante P, comprobar la firma del mensaje. Si verifica abortar, sino ignorar.
- Cuando reciba un mensaje $M_{\mathcal{E}}$ del ambiente $\mathcal{E}$, entregar $M_{\mathcal{E}}$ a $\mathcal{A}$.
- Cuando recibe un mensaje $M_{\mathcal{F}}$ de la funcionalidad $\mathcal{F}$.
  - Si $M_{\mathcal{F}}$ es `next period` actualizar las claves de todos los participantes.
  - Si $M_{\mathcal{F}}$ es `next-`P, entregar el mensaje al adversario $\mathcal{A}$
  - Si $M_{\mathcal{F}}$ =(“`backup`”,P,did,ℓ,$\texttt{Core}_P$):
    1. $d \xleftarrow{\$} \{0,1\}^{\ell}$
    2. Calcular C como $\mathcal{E}nc(TK[P], d)$.
    3. Sea $R \leftarrow (C \circ did \circ TK[P] \circ P)$.
    4. Computar la firma de R como $s \leftarrow Sig(TSK[P], R)$ y definir M como $R \circ s$.
    5. $\forall\ Q \in \texttt{Core}_P,\ T[P, did, Q] = M$.
    6. $\forall\ Q \in \texttt{Core}_P$, entregar M a $\mathcal{A}$.
    7. Cuando $\mathcal{A}$ responda, entregar `OK` a $\mathcal{F}$.
  - Si $M_{\mathcal{F}}$ =(“`release`”,P,did,$\texttt{Core}_P$):
    1. $\forall\ Q \in Core_P$, entregar (“`release`”,P,did) a $\mathcal{A}$.
    2. Cuando todas las llamadas a $\mathcal{A}$ respondan, entregar `OK` a $\mathcal{F}$.
  - Si $M_{\mathcal{F}} =$ `to-restore`P, did
    1. Generar un mensaje de `announce` simulando que lo crearon los participantes pertenecientes $\texttt{Core}_{P,did}$ que no han fallado y no son corruptos. Entregar estos mensajes a $\mathcal{A}$.
    2. Cada vez que $\mathcal{A}$ responda, generar un mensaje `request restore` simulando que lo creo el participante P para el participante Q y entregárselo a $\mathcal{A}$.
    3. Cuando se reciba la respuesta, entregar $T[P, did, Q]$ a $\mathcal{A}$ como un mensaje de restore.
    4. Al recibir a respuesta de $\mathcal{A}$, entregar `OK`$-did$ a $\mathcal{F}$.
  - Si $M_{\mathcal{F}}$ es el del tipo `Confirm` firmar el mensaje con la clave del período actual y enviársela al adversario como si fuera para cada uno de los participantes en el *core*. Cuando el adversario responda todos los mensajes, entregar `OK` a $\mathcal{F}$

Figura 3.7: Simulador $\mathcal{S}^{\mathcal{F},\mathcal{A}}_{enhanced}$.

De la misma manera que en la demostración del protocolo simple de debe demostrar el siguiente lema:

Lemma 10 *Dado el simulador de la figura 3.7, si los esquemas utilizados ($\mathcal{SE}$) y ($\mathcal{DS}$) para encriptar y firmar respectivamente son $(N-1, N)$-key-insulation*[7] *entonces se cumple que*

$$\forall \text{ ambiente } \mathcal{E},\ \forall \text{ adversario } \mathcal{A} \in \mathcal{C}_{fail-corrupt^*}$$
$$\text{UC-EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \text{UC-EXEC}_{I_{\mathcal{F}},\mathcal{S},\mathcal{E}}$$

La demostración de este lema es análoga a la del protocolo básico realizada en la sección 3.2. La diferencia en este caso es que se debe construir un adversario que quiebre el esquema de cifrado y firmas *key-insulation*. Para quebrar este esquema es necesario que el adversario, dada las claves de algunos períodos, pueda generar obtener las claves privadas de otro período (ver apéndice D).

Observaciones

La clase de adversarios $\mathcal{C}_{fail-corrupt^*}$ corresponden a adversarios que no pueden corromper a un participante durante tres períodos consecutivos. El supuesto clave detrás de esta clase de adversarios es que ningun virus o gusano puede apoderarse de un host en tres períodos de tiempo consecutivos. Este supuesto, aunque no parece adecuado a la vida real, si lo es. Cuando un usuario detecta que fue afectado por un gusano o virus lo más seguro es que detecte cual parte de su configuración fue atacada y, por ende, sepa cuál es su vulnerabilidad. Si un usuario es afectado en el período t, lo podrá detectar en el período $t+1$ (donde posiblemente aun este infectado), por lo tanto debera cambiar su configuración para el período $t+2$ (y los siguientes). Al cambiar de configuración, el mismo gusano no lo podrá afectar. En el caso de usuarios que no ocupen el sistema periódicamente, y por lo tanto no actualicen sus claves privadas en algunos períodos, tampoco hay problemas, pues sólo el usuario (con su *smartcard*) puede generar claves para cada períodos. De esta manera, ningún virus o gusano que obtenga las claves puede enviar una confirmación de backup o eliminación no deseada. Si se considerara adversarios más poderosos que puedan corromper indiscriminadamente a los participantes, ninguna de las propiedades de seguridad se cumplirian.

[7] Ver apéndice D .

Capítulo 4

Reputación

Un problema no cubierto por el modelo anterior es que usuarios maliciosos hagan uso del sistema sin pagar sus costos, es decir, que hagan de clientes queriendo respaldar sus datos, pero no paguen el costo de ser servidores guardando datos de otros usuarios y mantenerse disponible periódicamente. Para prevenir este problema se propone agregar un sistema de reputación que permita a los usuarios clasificarse según su comportamiento. De esta manera, un cliente puede descartar un servidor elegido en el *core* por su mala reputación[1].

Gupta et al. [15] definen dos tipos de sistemas de reputación (*credit-only reputation* y *credit-debit reputation*) y se describe como mantener la reputación en forma confiable solamente utilizando una tercera parte confiable. En `credit-only reputation` los puntos de reputación son todos positivos, en cambio en `credit-debit reputation` un usuario puede también tener puntos negativos lo cual podría significar que debe puntos positivos. Dada la naturaleza del sistema el tipo `credit-debit reputation` parece una excelente opción, los usuarios que tengan puntos negativos, deberán comportarse correctamente para subir su reputación.

Una gran idea es ocupar un sistema de dinero electrónico (*e-cash*) como puntos de reputación, en este caso la reputación de cada usuario depende de la cantidad de "monedas" que este tenga. Algunas las ventajas de ocupar *e-cash* son:

- Permite integrar otros sistemas distribuidos que compartan el sistema de reputación.
- Es razonable que un usuario servidor venda o ofrezca su espacio en disco.

[1]También podría ser que la reputación sea parte de la configuración del usuario, y por ende la elección del *core* descartará usuarios con mala reputación.

- Con *e-cash* nadie puede saber de donde se obtuvo la moneda utilizada en una cierta transacción[2].

Un enfoque como éste es el descrito por Androulaki et al [1] donde los autores abarcan problemas de privacidad y anonimato. Aunque este trabajo parece ser el más completo y el que abarca mayor cantidad de problemas, no es el más apropiado para el presente trabajo, pues su enfoque principal es preservar anonimato. Por este motivo, se prefiere basarse en un sistema algo mas simple y razonable como el descrito por Belenkiy et al. [3]. Los autores proponen un sistema de contabilidad electrónica basado en dinero electrónico para sistemas de *filesharing*. La principal idea de los autores es que el vendedor del archivo envía los datos cifrados con una llave elegida en forma aleatoria y luego el receptor debe comprar la llave al vendedor. Este proceso se hace mediante un *protocolo de intercambio justo* (ver glosario en apéndice A). Por lo tanto, el sistema de reputación puede ser pensado, mas bien, como un mecanismo de contabilidad que evita el problema descrito.

El problema de adaptar este sistema a un backup cooperativo es que no se puede saber exactamente cuando finalizó una transacción y por ende cuando realizar el traspaso de dinero. Con el siguiente ejemplo se puede observar este punto:

Alicia quiere respaldar sus datos en el computador de Bob. Si Alicia paga a Bob al momento de realizar el respaldo, Bob no tiene incentivos para seguir guardando los datos, Bob ya tiene el dinero. Sin embargo, si Alicia paga a Bob una vez que Alicia realizó una operación de restore satisfactoria, Bob es quien se arriesga a guardar los datos de Alicia, esperando que ella cumpliera al final de la transacción.

En [3] se propone que Alicia pague a Bob al momento de realizar el respaldo, pero Bob da una garantía, vía un protocolo de intercambio justo, que contiene un *Hash* de los datos, de manera que un árbitro puede verificar fácilmente si los usuarios se han comportado correctamente o no. En caso de que Bob entregue datos alterados a Alicia al momento de la recuperación, el arbitro, con la garantía entregada por Bob, puede entregar dinero de Bob a Alicia por el daño cometido.

La manera más correcta de realizar esto es que en cada período de tiempo el cliente le pague al servidor por el siguiente período de respaldo y la garantía se va actualizando. Al fin de cada período Bob debe comprobar que aún tiene los datos de Alicia mediante el cálculo

[2]No se puede saber el participante que entregó la moneda.

del hash o un MAC de los datos y verificando su validez.

Una pregunta que surge es ¿Cuánto vale la garantía? La garantía debe valer al menos lo mismo que el cliente le paga al servidor. Como se quiere prevenir el mal uso del sistema, se preferiría que usuarios maliciosos entreguen una garantía mucho mayor que los usuarios honestos. El problema es que si un usuario servidor no entrega los datos en la fase de recuperación no se puede saber, si bien, es un usuario honesto que fue afectado por alguna epidemia o virus, o bien, es en realidad un usuario malicioso. En el caso que el servidor sea honesto pero no tenga los datos, se preferiría tener una garantía baja, que sencillamente devuelva el dinero que pago el cliente y tal vez un poco más. En el caso que el servidor sea realmente deshonesto, la garantía debiera ser más alta. El problema es que no es claro como distinguir entre ambos casos, por lo que no podemos hacer una diferencia en el valor de la garantía.

Para atacar el problema de la garantía, se sugiere que los propios usuarios acuerden el valor de la garantía. Para esto cada usuario debe poder demostrar que tiene un cierto nivel reputación[3]. Los usuarios que demuestren tener mayor nivel de reputación podrán ofrecer garantías pequeñas, pero lo usuarios con nivel de reputación bajo deberán ofrecer garantías mayores.

También se sugiere que si el cliente quiere respaldar sus datos por un largo período de tiempo esto se haga a través de períodos pequeños, por ejemplo de una semana, en donde al fin de cada semana el servidor demuestra en forma eficiente que aún tiene los datos del cliente. Luego el cliente paga por el siguiente período y la garantía se actualiza mediante un protocolo de intercambio justo.

Otro problema de este tipo de sistema es que un nuevo usuario parte sin dinero y por lo tanto no puede ocupar el sistema hasta ganar algo. Una solución es que otros usuarios le presten dinero inicialmente; esta idea se adapta muy bien al uso del sistema en una red social, pues los "amigos" del nuevo usuario, que confían realmente en él, serán los que le presten el dinero inicial. Otra opción es que el usuario pueda comprar dinero con dinero real. Este punto permite evitar que malos usuarios con baja reputación (sin dinero) creen una nueva identidad que les permita parecer inocentes por un período de tiempo. La otra opción, es que los nuevos usuarios se arriesguen y ofrezcan garantías muy altas.

La cantidad de dinero disponible debe ser manejada por el banco, los primeros usuarios,

[3]Esto se puede hacer mediante el protocolo descrito por Androulaki et al. en [1].

que se suponen honestos, recibirán dinero en forma gratuita e instantánea cuando se registren en el banco. A medida que el sistema vaya creciendo el banco deberá proveer dinero en forma gratuita a los usuarios de manera que el sistema siga siendo utilizable. Es decir, cuando el dinero sea escaso para la cantidad de usuarios en el sistema, no quedará otro remedio que el banco agregue dinero al sistema.

Para integrar un protocolo de este estilo, se debe proveer de dos entidades centralizadas confiables, uno es el banco, quien mantiene la contabilidad de cada usuario, permite hacer giros y depósitos y además permite que cualquier usuario pueda demostrar que tiene un cierto nivel de reputacion[4]. La segunda entidad es el árbitro que permite realizar el protocolo de intercambio justo detectando a servidores que alteren los datos del cliente.

Cada vez que un usuario se integra al sistema debe registrarse y crear una cuenta en el banco. Si algún usuario intenta comportarse maliciosamente en una interacción, el usuario honesto puede invocar al árbitro que permite realizar las transacciones de manera confiable y justa.

Sería de gran utilidad que el sistema de reputación permita introducir nuevos sistemas peer-to-peer y que estos ocupen los mismos puntos de reputación. Por ejemplo, dada una serie de sistemas distribuidos: *file-sharing*, *CPU-sharing*, etc, que funcionan bajo una red social como *facebook* o *MySpace*, estos sistemas podrían compartir el sistema de reputación. Es decir, un peer que gane puntos de siendo un buen usuario en el sistema de file-sharing, puede ocupar esos mismos puntos de reputación en el sistema de CPU-sharing. La gran ventaja de agrupar distintos sistemas peer-to-peer bajo un mismo sistema de reputación es que permite la fácil integración de nuevos usuarios lo cual es bastante beneficioso debido a la propiedad de escalabilidad de estos sistemas (mientras mayor es la cantidad de usuarios, mayor es la capacidad del sistema). Sin embargo, se asume implicitamente que si un usuario es buen usuario en un sistema específico, también lo es en los otros. Esta suposición no siempre es cierta, por ejemplo, un usario puede comportarse como un buen servidor en el sistema de backup, pero en el sistema de *file-sharing* introduce archivos maliciosos.

[4]El nivel de reputación es una función creciente que depende de la cantidad de dinero que tiene el usuario en el banco. Ejemplo: Un usuario tiene un nivel i de reputación si tiene más de 2^i monedas depositadas en el banco.

Con respecto a las ideas mencionadas sobre el sistema de reputación, cabe destacar que hay una serie de problemas importantes no resueltos. El primer problema es evitar que usuarios deshonestos cambien de identidad para mejorar su reputación fácilmente y volver a ser deshonestos poco tiempo después. Un segundo problema es evitar la centralización de dinero en pocos usuarios. Para resolver ambos problemas es necesario que los nuevos usuarios no sean capaces de llegar a tener algún buen nivel de reputación en poco tiempo o ocupando pocos recursos, y a la vez, usuarios con un nivel de reputación muy alto no deberian poder subir su nivel fácilmente, previniendo la centralización de dinero en pocos usuarios. No es claro como implementar un sistema de reputación con las características recien mencionadas. De alguna manera las transacciones no pueden ser 100 % "libres" y deberán ser controladas con reglas específicas. Finalmente se tiene que al utilizar *e-cash* y protocolos de intercambio justos estamos introduciendo entidades centralizadas supuestamente confiables. La principal idea de los protocolos descritos en el trabajo es evitar entidades centralizadas que pueden ser corrompidas en cualquier momento. No está claro si se pueden descentralizar estas entidades.

Capítulo 5

Detalles de la implementación

5.1. Trabajo realizado

Se implementó un sistema distribuido de respaldo de datos basado en el protocolo simple descrito en la sección 2.3.

El código desarrollado y documentación puede ser encontrado en `http://proyectos.clcert.cl/`.

Las diferencias entre protocolo simple y la implementacion realizada son:

- Las claves de cifrado y de firmas no son generadas a través del *passphrase* cada vez que se necesiten. El usuario genera sus claves sólo una vez, al comienzo, y luego son cifradas y guardas en un archivo (como se describe en la sección 2.3). De esta manera, se puede asegurar que el usuario siempre utiliza las claves correctas evitando que un mal tipeo del *passphrase* genere claves erroneas.
- Hasta el momento no se ha implementado el algoritmo para encontrar el *core* de un usuario. Por lo que el usuario debe ingresar en forma manual su *core*.
- La forma de anunciar que un usuario ha fallado es a través de un servidor central, este servidor se preocupa de avisar quien ha fallado y a que hora.

5.2. Componentes

Las principales componentes de software desarrolladas son las siguientes

5.2.1. Criptografía

La componente de criptografía es la que se preocupa de proveer herramientas de firma digitales, cifrado simétrico de archivos y un esquema de autentificación de mensajes MAC.

La herramienta de firmas digitales y cifrado de archivos es ocupado directamente por el protocolo para proveer integridad y privacidad de datos respectivamente. El esquema de MAC es utilizado para guardar las claves del usuario.

La implementación de esta componente se encuentra dentro del paquete java *cryptography*.

Las clases desarrolladas en este componente son:

- AES: permite cifrar archivos.
- RSAKeyManager: Genera par clave pública, clabe privada para firmar y verificar mensajes.
- MAC: Implementa un esquema de autentificación de mensajes.

5.2.2. Comunicación

La componente de comunicación provee una forma fácil de transferir archivos y datos entre los participantes.

El paquete java *communication* implementa esta componente.

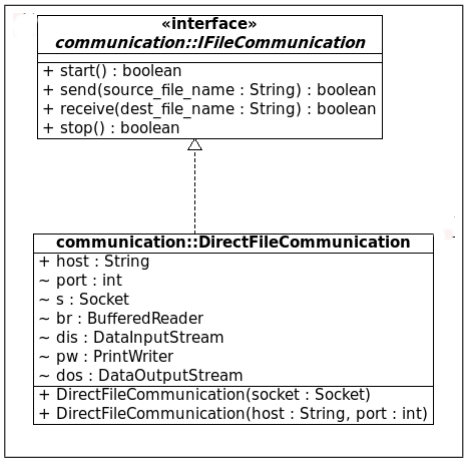

Figura 5.1: Diagrama de Clases: paquete *communication*

5.2.3. Protocolo

Esta componente implementa las fases de un protocolo de respaldo de datos. Además aquí se encuentra la implementación del protocolo simple y permite la integración de nuevos protocolos.

La implementación de esta componente se encuentra en el paquete *protocol*. En la figura 5.2 se muestra en el diagrama de clases de este paquete.

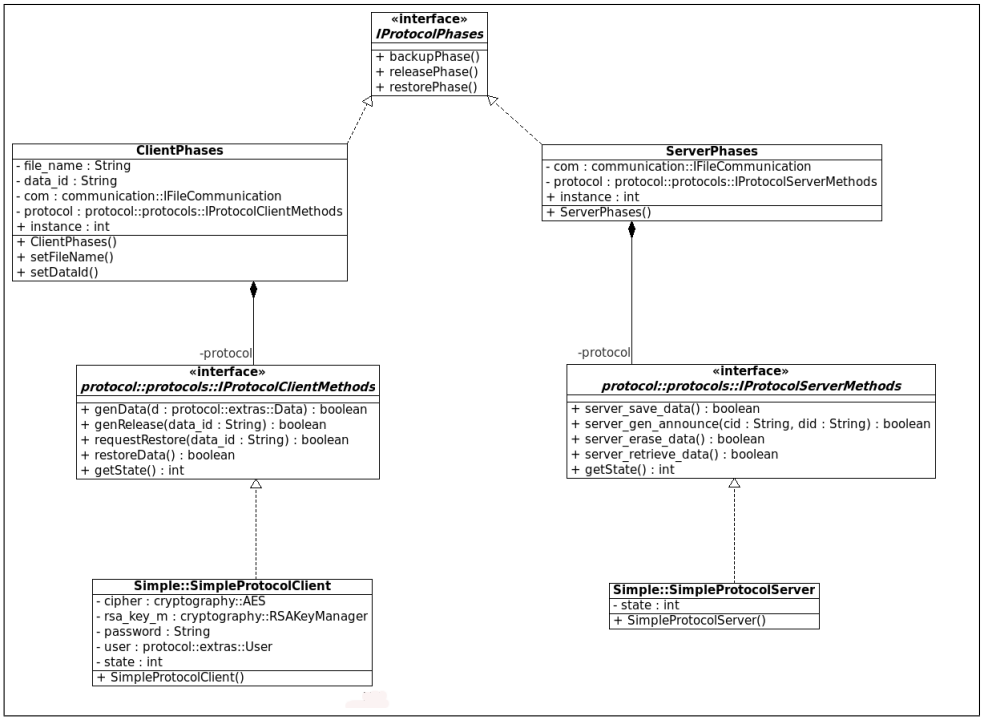

Figura 5.2: Diagrama de Clases: paquete *protocol*

5.2.4. Interfaz de usuario

La componente de interfaz de usuario implementa las interfaces gráficas para que el usuario utilice el sistema de forma fácil. Esta componente está implementada en el paquete java *gui*.

Las clases que implementan la interfaz con el usuario son: PcGUI, LoginUserGUI, ShowInfoGUI, IpGUI, NewUserGUI, WelcomeGUI y PassGUI.

5.2.5. Mensajes

Provee una forma fácil de realizar el *parsing* de los mensajes enviados entre los participantes y generarlos.

El paquete que implementa esta componente es *messages*.

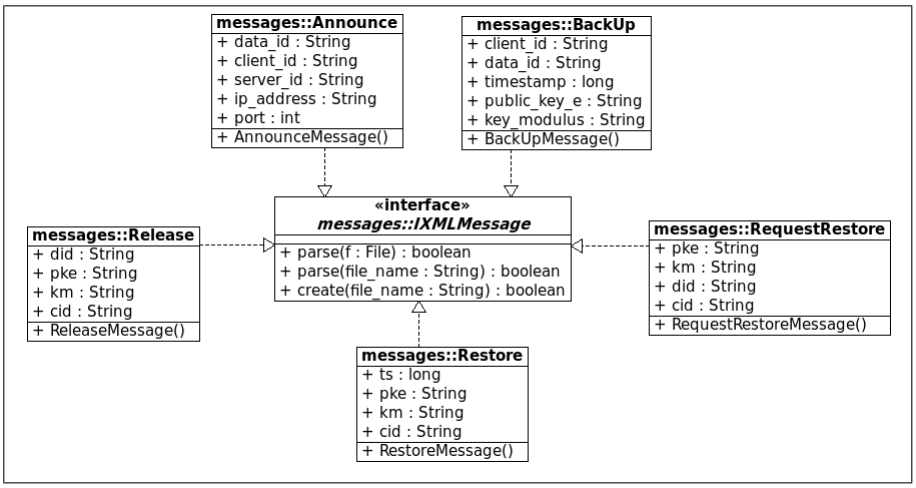

Figura 5.3: Diagrama de Clases: paquete *messages*

5.2.6. Otros Componentes

Hay otros componentes o paquetes complementarios a los componentes anteriores. Estos son:

- *configuration*: El paquete de configuración maneja aspectos propios del usuario que está ejecutando la aplicación, por ejemplo, donde guarda los archivos generados y recibidos, su nombre de usuario, etc.

- *constants*: El paquete de constantes permite obtener de manera fácil algunas constantes como el nombre de los archivos a utilizar, los puertos que utiliza el programa, el número *IP* de algún servidor central, etc. Estas constantes pueden ser moodificadas en un archivo de configuración.

- *utilities*: El paquete utilities contiene clases auxiliares utilizados por las demas clases. Un ejemplo es la clase `KeyMath` que permite manejar las claves en diferentes formatos.

- *setup*: El paquete setup contiene 2 clases: `InitialSetup` y `SimpleProtocolSetup`. `InitialSetup` permite generar un nuevo usuario, generando los archivos y directorios necesarios, además cifra y guardas las claves generadas a partir del *passphrase* que entrega el usuario al iniciarse. `SimpleProtocolSetup` contiene el código utilizado para obtener las claves privadas cada vez que un usuario ejecute una operación del protocolo simple.

- *main*: El paquete contiene las clases que inician el sistema, ejecutan los thread necesarios y las interfaces gráficas.

5.3. Diagrama de clases

En el apéndice F se muestra el diagrama de clases general del sistema. Se puede observar que el sistema es extensible para adaptar nuevos protocolos, esquemas criptográficos y mecanismos de comunicación. También se puede ver que las clases referentes a al interfaz gráfica (terminadas en GUI) están bien correlacionadas entre ellas, pero no con el resto del sistema. Esto permite cambiar de manera fácil las interfaces con el usuario para incluir, por ejemplo, *applets* y adaptar el sistema a una aplicación en una red social.

5.4. Discusión: Ambientes en donde implementar y posibles usos

El sistema implementado funciona actualmente en una red cerrada en donde los usuarios tienen computadores con dirección IP fija. Esta implementación es fácilmente extensible para ser utilizada en computadores con IP variable (basta con extender la capa de comunicación) y con esto los usuarios ya no tienen que estar ligados a un computador en específico para obtener sus datos. Esto lleva a un problema, si un usuario del sistema puede cambiar constantemente de computador, ¿Donde debe guardar los respaldos que otros usuarios hicieron en él?

El protocolo no especifica absolutamente nada sobre este punto, por lo que los datos pueden ser guardados en cualquier lugar o espacio que el usuario disponga. Por ejemplo, los datos los puede guardar en su notebook personal, en un servidor propio al cual siempre se tenga acceso, en su correo electrónico y pedirlos mediante un simple cliente *IMAP* o en cualquier cuenta de *Filehosting* que se provea (por ejemplo, RapidShare, Amazon Simple Storage Service, etc). Guardarlos en su notebook personal no es la mejor idea, pues como el usuario se conecta al sistema desde diferentes computadores no siempre tendrá acceso a los datos y su reputación bajará (pues afecta a la propiedad de disponibilidad). El lugar en donde se respaldan los datos puede ser agregado a la configuración del usuario, con esto otros usuarios pueden saber claramente donde el usuario servidor respaldará los datos.

Una pregunta obvia que surge es ¿Para qué querría un usuario respaldar sus datos en una cuenta de *FileHosting* de otro usuario, si él mismo puede respaldarlos en su propia cuenta de *FileHosting*? Hay que imaginar que un usuario A respalda todos los backups de otros usuarios en su cuenta mail en *gmail*. Un usuario B, quien también tiene una cuenta en *gmail*, se pregunta para qué respaldar en el usuario A, si puede respaldar directamente en su propia cuenta. La respuesta es simple, es mejor tener respaldados los datos en varios otros usuarios, aunque también ocupen *gmail*, por si la cuenta del usuario A se ve comprometida y un adversario borra todos sus datos. Ahora la estrategia ideal del usuario A es respaldar en varios usuarios que tengan diferentes configuraciones; uno que ocupe un *Filehosting* como *RapidShare*, otro que ocupe *gmail*, otro que guarde en un disco duro personal, etc.

Un punto importante a mencionar es que la implementación actual del sistema, aunque es correcta, no provee una mejora en el sentido de la usabilidad y aplicabilidad a otros sistemas de backup ya existentes. Es decir, existen sistemas de respaldo de datos ya probados y usados

por una gran cantidad de usuarios. Sin embargo, el protocolo presentado en el trabajo es una descripción que puede ser implementada en muchos ambientes diferentes y para objetivos diferentes. Por ejemplo, el protocolo puede ser implementado en una red de telefonía celular, en una red social o en una red cerrada en una empresa, en donde las soluciones existentes son caras o no proveen las mismas garantías de seguridad (por ejemplo, sistemas centralizados).

Una posible implementación sería en una red social como *Facebook* o *MySpace*. La gran ventaja es la facilidad de encontrar usuarios-servidores una vez que un usuario-cliente ha fallado y perdido todos sus datos. Cuando un usuario es afectado por un virus o su computador ha sufrido una catástrofe, este ya no sabe qué usuario tiene respaldado sus datos. Al ocupar una red social se puede obtener una lista de los posibles usuarios-servidores y preguntarle a cada uno de ellos si tiene un *announce* de los datos. Además facilita integración de nuevos usuarios, lo que permite tener un sistema con mayores beneficios debido a la gran escalabilidad que tienen este tipo de sistemas distribuidos. Otra gran ventaja de ocupar algún tipo de red social, es que, generalmente, los usuario-servidores son conocidos realmente por el cliente, de manera que baja la probabilidad de tener servidores deshonestos.

Un uso un poco diferente a los demás sistemas de backup ya existentes se puede dar al implementar el sistema en un ambiente en donde los usuarios estén conectados gran parte del tiempo(por ejemplo, MSN messenger, Facebook, cuenta de email, GTalk). Se podría implementar un tipo de cartera digital, en donde se puedan dejar todo tipo de datos (texto, links interesantes, fotos, videos, etc.) La idea principal es que cada vez que un usuario deja de utilizar esta cartera, los datos guardados en ella se respalden en otros usuarios de manera que, luego, cuando el usuario se reconecte pueda abrir su cartera y seguir ocupando sus datos en forma transparente. Este sistema es de gran utilidad para usuarios que cambian de dispositivo a través del día, por ejemplo, un usuario que en la oficina ocupa un PC, en la calle ocupa su celular, en una cafetería ocupa su laptop y en su casa ocupa otro dispositivo.

5.5. Trabajo futuro

Como complementación a la implementación realizada se puede agregar la integración del protocolo avanzado, el algoritmo de detección de *cores* y adaptar el protocolo de manera que no sea necesario utilizar un servidor central para el caso de que un usuario falle.

Para la integración del protocolo avanzado se necesitaría que cada usuario tuviese algún tipo de dispositivo[1] que permita guardar una clave secreta maestra y generar las claves para cada período. Es importante notar que si se tiene un dispositivo con algo de memoria utilizable, el *core* de cada usuario se puede guardar ahí, por lo que no se necesitaría tener un servidor que comunique quien ha fallado. El problema con esto es que, al poder escribir en el dispositivo, no se puede asegurar invulnerabilidad de ataques en él.

Otra forma de complementar el sistema e incluir nuevos usuarios en forma fácil y rápida, es la integración de applets en la componente de *Interfaz de Usuario*, de manera de integrar el sistema en una red social web. El diseño por componentes permite realizar esto facilmente, bastanto integrando nuevas funciones a la componente de *Interfaz de Usuario*.

[1] teléfonos celulares, smartcards, etc.

Capítulo 6

Conclusiones

Los objetivos del trabajo de memoria fueron implementar un sistema de respaldo de datos distribuido y estudiar la seguridad del sistema utilizando herramientas actuales de demostración de seguridad para protocolos criptográficos.

Con respecto al primer objetivo, se logró implementar en Java un sistema de respaldo de datos que corresponde al protocolo básico presentado en la sección 2.3. Esta implementación funciona actualmente bajo una red cerrada TCP/IP en donde los participantes mantienen su dirección IP fija. Asimismo, se está desarrollando en forma adicional un cliente basado en java *applets* de manera de utilizar el sistema en un navegador web y con un mecanismo de comunicación que permita a los participantes cambiar de dirección IP.

El segundo objetivo, la parte más teórica del trabajo, fue el estudio de seguridad del sistema. Para esto se estudió la herramienta Universal Composability y se aplicó a los protocolos propuestos para el sistema. La razón de aplicar esta herramienta es que permite definir claramente la funcionalidad del sistema dejando claro que es lo que hace y que es lo que no hace y, a la vez, obtener una demostración matemática sobre las propiedades de seguridad definidas.

Específicamente en el trabajo se logró lo siguiente:

- Definir un sistema de respaldo de datos distribuido.
- Definir el significado de seguridad para el sistema.
- Proponer un primer protocolo y realizar una discusión sobre las posibles limitaciones de éste.

- Proponer un segundo protocolo que, utilizando herramientas criptográficas más poderosas, mejora las garantías de seguridad del primer protocolo.
- Realizar un resumen de la herramienta *Universal Composability*, explicando su objetivo, modo de uso, diferentes modelos posibles (modelos híbridos, por ejemplo) y cómo definir matemáticamente seguridad bajo *Universal Composability*.
- Ocupar Universal Composability para demostrar seguridad tanto en el primer como en el segundo protocolo.
- Proponer un mecanismo de reputación que extienda los protocolos definidos de modo de complementar la seguridad en el sistema.
- Implementar el sistema de forma que sea extensible en las herramientas criptográficas usadas, cambios en el protocolo y en los mecanismos de comunicación.
- Realizar una discusión sobre los posibles usos y ambientes en donde ocupar el sistema.

Se puede concluir que se logró el objetivo general, realizando y desarrollando todos los objetivos específicos. Como resultado se presentaron protocolos para sistemas de backup demostrando que cumplen con estándares altos de seguridad mediante el uso de UC. Adicionalmente, se encuentra disponible[1] la API y el código que implementa el protocolo básico, dejando abierta la posibilidad de complementar el sistema.

El trabajo futuro a desarrollar comprende de los siguientes aspectos:

- Extensión del sistema desarrollado en este trabajo para que incorpore el protocolo avanzado. Para esto es necesario tener disponibles *smartcards* (u otro dispositivo similar) y una infraestructura de clave pública de manera poder implementar el esquema de firmas *key insulated signatures* [12, 13] y canales autenticados de comunicación.
- Diseñar e incorporar un sistema de reputación basado en las ideas propuestas en el capítulo 4.
- Implementar la heurística de búsqueda de *cores* [17].
- Implementar un sistema de aviso de fallas totalmente distribuido (en la implementación actual hay un servidor central que se ocupa de este punto).

[1] http://proyectos.clcert.cl

Referencias

[1] E. Androulaki, S. G. Choi, S. M. Bellovim, and T. Malkin. Reputation systems for anonymous network. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 202–218, 2008.

[2] M. Backes, B. Pfitzmann, and M. Waidner. A general composition theorem for secure reactive systems. pages 336–354. Springer, 2004.

[3] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making p2p accountable without losing privacy. In *WPES '07: Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 31–40, New York, NY, USA, 2007. ACM.

[4] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract). In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 419–428, New York, NY, USA, 1998. ACM.

[5] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. pages 394–403, 1997.

[6] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with rsa and rabin. pages 399–416. Springer-Verlag, 1996.

[7] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 2000.

[8] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/.

[9] R. Canetti, L. Cheung, D. Kaynar, N. Lynch, and O. Pereira. Compositional security for task-pioas. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 125–139, Washington, DC, USA, 2007. IEEE Computer Society.

[10] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. pages 453–474. Springer-Verlag, 2001.

[11] A. Datta, R. Küsters, J. C. Mitchell, and A. Ramanathan. On the relationships between notions of simulation-based security. In *In TCC 2005*, pages 476–494. Springer-Verlag, 2005.

[12] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *EUROCRYPT '02: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, pages 65–82, London, UK, 2002. Springer-Verlag.

[13] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In *PKC '03: Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography*, pages 130–144, London, UK, 2003. Springer-Verlag.

[14] J. T. Gill, III. Computational complexity of probabilistic turing machines. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 91–95, New York, NY, USA, 1974. ACM.

[15] M. Gupta, P. Judge, and M. Ammar. A reputation system for peer-to-peer networks. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 144–152, New York, NY, USA, 2003. ACM.

[16] A. Hevia, F. Junqueira, and G. Voelker. Secure archival-storage systems.

[17] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving internet catastrophes. In *In Proceedings of the Usenix Annual Technical Conference*, 2005.

[18] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. Voelker. The phoenix recovery system: Rebuilding from the ashes of an internet catastrophe. *In Proc. of HotOS-IX*, Mayo 2003.

[19] F. P. Junqueira and K. Marzullo. Synchronous consensus for dependent process failures. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 274, Washington, DC, USA, 2003. IEEE Computer Society.

[20] R. Kusters. Simulation-based security with inexhaustible interactive turing machines. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 309–320, Washington, DC, USA, 2006. IEEE Computer Society.

[21] D. More, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. *In Proc. of the ACM/USENIX Internet Measurement Workshop*, Noviembre 2002.

[22] D. More, V. P. S.Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE privacy and Security*, Julio 2003.

[23] C. E. Shannon. Prediction and entropy of printed English. *The Bell System Technical Journal*, 30:50–64, 1951.

Apéndices

A . Glosario

En este capítulo se definen algunos conceptos básicos para una lectura clara del informe.

- **Esquemas de encriptación simétricos:** Esquema de cifrado de datos en donde la clave utilizada para encriptar el mensaje es la misma que la utilizada para desencriptar el mensaje [5] (ver apéndice B).
- **Esquema de firmas:** Esquema criptográfico que permite asegurar la identidad del remitente de un mensaje [6] (ver apéndice C).
- **Esquema de firmas *key-insulated***: Este esquema funciona como un esquema de firmas, pero separando la clave secreta en dos dispositivos distintos, uno físicamente protegido (*smartcard* u otro dispositivo móvil similar) y un computador posiblemente inseguro. Al separar la clave, se garantiza que, aunque el computador inseguro sea atacado por un adversario en un período t, las firmas que se realizan en todos los períodos distintos a t son válidas y seguras, ya que la clave secreta evoluciona en cada período [12, 13](ver apéndice D).
- **Universal Composability**: Marco de trabajo teórico utilizado para demostraciones de seguridad en protocolos que se ejecutan concurrentemente [8] (ver sección 3.1).
- **Backup**: Término en inglés para respaldo de datos.
- **Infraestructura de Clave Pública** (Public Key Infraestructure, PKI): Tecnología utilizada en redes que permite autentificación de usuarios y sistemas, firmas digitales, comunicación privada y garantía de no repudio. Las principales componentes son:
  - Autoridad certificadora: Entidad confiable que emite y revoca certificados.

- Autoridad de registro: Entidad encargada de verificar el enlace entre certificados y la identidad de sus titulares.
- Repositorios de certificados y listas de revocación de certificados (certificados que ya no son válidos).
- Usuarios: Poseen un certificado asociado a su clave pública.

Una PKI equivale a una base de datos manejada por una entidad confiable que contiene los pares $< usuario, clave\ publica >$. Todos los usuarios pueden preguntar por la clave pública de los demás usuarios.

- **Código de autentificación de mensajes** (Message Authentification Code, MAC): Esquema criptográfico que permite autentificar un cierto mensaje, en donde el remitente y el destinatario comparten la clave de autentificación.
- **Función de Hash**: Es una función que mapea un string de largo arbitrario a string de largo fijo:

$$H : \{0,1\}^* \rightarrow \{0,1\}^n$$

Para algún n fijo.

Una **función criptográfica de hash** es una función de hash que cumple las siguientes tres propiedades:

- Resistente a la preimagen: Dado un string $Y \in \{0,1\}^n$, es difícil encontrar X tal que $H(X) = Y$.
- Resistente a la segunda preimagen: Dado un string X, es difícil encontrar $X' \neq X$ tal que $H(X) = H(X')$.
- Resistente a colisiones: Es difícil encontrar el par (X, X') tal que $H(X) = H(X')$ $(X \neq X')$.

- **Función derivadora de claves** (Key Derivation Function, KDF): Es una función que genera una o más claves secretas a partir de un valor inicial, por ejemplo una contraseña.

- **Protocolo y Participantes**: Un protocolo es un conjunto de reglas que controlan la sintaxis, la semántica y la sincronización en la comunicación entre participantes que quieren llevar a cabo una cierta tarea.

- **Adversario**: En un protocolo un adversario es una entidad maliciosa que, posiblemente, puede interceptar los mensajes enviados entre los participantes retrasándolos, modificándolos, analizándolos, etc. Además, el adversario puede ser capaz de controlar un conjunto de participantes haciendo que ejecuten algún código malicioso. El objetivo del adversario es atacar alguna de las propiedades de seguridad del protocolo. Los adversarios permiten, por ejemplo, modelar el hecho que exista una coalición maliciosa entre algunos de los participantes para obtener información privada de otros participantes.

- **Adversario Estático**: Un adversario estático (o no adaptativo) puede controlar un conjunto arbitrario, pero fijo, de participantes corruptos [7].

- **Adversario Dinámico**: Un adversario dinámico (o adaptativo) puede elegir las entidades a corromper durante la ejecución del protocolo [7].

- **Adversario Pasivo**: Un adversario pasivo (o semi honesto) sigue el protocolo, pero ve y aprende los mensajes que se envían los participantes, no los modifica [7].

- **Adversario Activo**: Un adversario activo (también llamados "Bizantinos") pueden controlar a participantes ya corruptos haciendo ejecutar algún código malicioso [7].

- **Algoritmos PPT**: Los algoritmos denominados PPT son algoritmos aleatorizados que corren en tiempo polinomial [14]. En su caracterización más simple, un algoritmo A es PPT si existe un polinomio $p(\cdot)$ tal que la cantidad de operaciones que ejecuta $A(x)$ es menor o igual a $p(|x|)$. Donde x es una entrada. La definición usada en UC es algo más compleja y remitimos al lector interesado a revisar [8].

- **Protocolo de Intercambio Justo**: Es un protocolo ejecutado por dos participantes que asegura que la salida de cada participante es la entrada del otro participante [3]. Permite realizar en forma confiable el intercambio de las entradas de cada participante.

B . Seguridad de sistemas de encriptación simétricos: Indistinguibilidad ante ataques de texto plano escogido

Definición 11 *Un **esquema de encriptación simétrico**, $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, consiste de los siguientes tres algoritmos:*

- *El algoritmo aleatorizado **generador de claves** $\mathcal{K}$. Se denota $k \xleftarrow{\$} \mathcal{K}$ a la operación de ejecutar $\mathcal{K}$, la clave retornada es k.*
- *El algoritmo de **encriptación** $\mathcal{E}$, toma una clave $k \xleftarrow{\$} \mathcal{K}$ y el **texto plano** $M \in \{0,1\}^*$ y retorna el **texto cifrado** $C \in \{0,1\}^* \cup \{\perp\}$. Se denota $C \leftarrow \mathcal{E}_k(M)$ a la operación de ejecutar $\mathcal{E}$ con entrada la clave k y el texto M, su salida es el texto cifrado C.*
- *El algoritmo de **desencriptación** $\mathcal{D}$, toma una clave $k \xleftarrow{\$} \mathcal{K}$ y el texto cifrado C y devuelve algún $M \in \{0,1\}^* \cup \{\perp\}$. Se denota $M \leftarrow \mathcal{E}_k(C)$ a la operación de ejecutar $\mathcal{E}$ con entrada la clave k y el texto cifrado C, su salida es el mensaje M.*

La idea básica detrás de indistinguibilidad ante ataques de texto plano escogido (*IND-CPA*[2]), es considerar un adversario que escoge dos mensajes del mismo largo. Uno de los dos mensajes es cifrado y enviado al adversario. El esquema de cifrado simétrico es considerado seguro si el adversario no puede distinguir cual mensaje fue cifrado. Al adversario se le da un poco más de poder dejandolo elegir toda una secuencia de pares de textos planos. El juego se define como:

El adversario elige una secuencia de pares de mensajes $(M_{0,1}, M_{1,1})$,...,$(M_{0,q}, M_{1,q})$ en donde, en cada par, los mensajes son del mismo largo. Se le da al adversario una secuencia de textos cifrados $C_1, ..., C_q$, en donde, $\forall\ i\ \in\ \{1, ..., q\}$, C_i es el cifrado de $M_{0,i}$, o bien, C_i es el cifrado de $M_{1,i}$. Luego el adversario debe adivinar si los textos cifrados corresponden a la secuencia $(M_{0,1}, ..., M_{0,q})$ o a $(M_{1,1}, ..., M_{1,q})$. Además se le deja al adversario actuar en forma adaptativa, esto es, elegir cada par de mensajes luego de recibir el cifrado del par anterior.

Formalmente, dado un esquema de cifrado simetrico $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, un adverasario $\mathcal{A}$ es un programa que tiene acceso a un oráculo que recibe el par de mensajes y entrega un texto cifrado. Existen dos posibilidades de computo del texto cifrado, que corresponden a dos posibles "*mundos*" en donde el adversario es ejecutado. Para hacer esto, primero se define el oráculo de cifrado *left-or-right*, $\mathcal{E}_K(LR(\cdot, \cdot, b))$ el cual cifra uno de los dos mensajes según el bit b. Ahora de definen ambos mundos.

[2]En inglés: *Indistinguishability under chosen-plaintext attack.*

Mundo 0: El oráculo dado al adversario es $\mathcal{E}_K(LR(\cdot,\cdot,0))$. Entonces, cuando el adversario realice una consulta (M_0, M_1), el oráculo computa $C \xleftarrow{\$} \mathcal{E}_K(M_0)$ y retorna C.

Mundo 1: El oráculo dado al adversario es $\mathcal{E}_K(LR(\cdot,\cdot,1))$. Entonces, cuando el adversario realice una consulta (M_0, M_1), el oráculo computa $C \xleftarrow{\$} \mathcal{E}_K(M_1)$ y retorna C.

Definición 12 *Sea $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ un esquema de encriptación simétrico y sea $\mathcal{A}$ un algoritmo con acceso a un oráculo. Sea k un parámetro de seguridad. Considerar el siguiente experimento:*

Experimento 3: $Exp^{ind\text{-}cpa}_{\mathcal{SE},\mathcal{A}}(k)$

```
b ←$ {0,1};
K ←$ K(k);
b' ←$ A^{E_k(LR(·,·,b))};
if b = b' then
    output 1 ;
else
    output 0 ;
end
```

Se define la ventaja del adversario $\mathcal{A}$ como

$$Adv^{ind\text{-}cpa}_{\mathcal{SE},\mathcal{A}}(k) = 2 \cdot P[Exp^{ind\text{-}cpa}_{\mathcal{SE},\mathcal{A}}(k) = 1] - 1$$

Si la ventaja es pequeña para todo algoritmo $\mathcal{A}$, entonces el esquema es seguro en el sentido *IND-CPA*, pues el adversario no puede ditinguir con probabilidad no despreciable en cual mundo se encuentra.

C . Seguridad de esquemas de firmas: No falsificación de firmas ante ataques de mensajes escogidos

Definición 13 *Un **esquema de firmas digitales**, $\mathcal{DS} = (\mathcal{K}, \mathcal{S}ign, \mathcal{V}f)$, consiste de los siguientes tres algoritmos:*

- *El algoritmo aleatorizado **generador de claves** $\mathcal{K}$. Genera el par (pk, sk) correspondiente a la **clave pública** y **clave secreta**. Se denota $(pk, sk) \xleftarrow{\$} \mathcal{K}$ a la operación de ejecutar $\mathcal{K}$ retornando el par (pk, sk).*
- *El algoritmo de **firmas** $\mathcal{S}ign$, toma una clave secreta sk y un mensaje $M \in \{0,1\}^*$ y retorna una **firma** $\sigma \in \{0,1\}^* \cup \{\perp\}$. Se denota $\sigma \leftarrow \mathcal{S}ign_{sk}(M)$ a la operación de ejecutar $\mathcal{S}ign$ con entrada sk y M, el valor retornado es la firma σ.*
- *El algoritmo **verificador** $\mathcal{V}f$, toma una clave pública pk, un mensaje M y un candidato a firma σ y retorna un bit. Se denota $b \leftarrow \mathcal{V}f_{pk}(\sigma, M)$ a la operación de ejecutar $\mathcal{V}f$ con entrada pk, M y σ, el valor retornado es el bit b.*

Se requiere que $\mathcal{V}f_{pk}(M, \sigma) = 1$, para cualquier par (pk, sk) que pudo ser retornado por el algoritmo $\mathcal{K}$, cualquier mensaje M y cualquier $\sigma \neq \perp$ que pudo ser retornado por $\mathcal{S}ign_{sk}(M)$.

La manera de definir seguridad en un esquema de firmas digitales es considerar un adversario que teniendo acceso a un oráculo de firmas $Sign(pk, \cdot)$, crea un par (M, σ) en donde σ es la firma del mensaje M y este no fue consultado al oráculo. Es decir, el adversario puede crear una firma falsa teniendo acceso a la clave pública y a una cantidad limitada de preguntas al oráculo.

Definición 14 *Sea $\mathcal{DS} = (\mathcal{K}, Sign, VF)$ un esquema de firmas digitales y sea $\mathcal{A}$ un adversario que tiene acceso a un oráculo y que retorna un par de strings. Sea k un parámetro de seguridad. Considerar el siguiente experimento:*

Experimento 4: $Exp^{uf\text{-}cma}_{\mathcal{DS},\mathcal{A}}(k)$

```
(pk, sk) ←$ K(k);
(M, σ) ← A^{Sign_sk(·)}(pk);
if VF_pk(M, σ) = 1 ∧ M no fue consultado al oraculo por A then
    output 1 ;
else
    output 0 ;
end
```

Se define la ventaja del adversario $\mathcal{A}$ como

$$Adv^{uf\text{-}cma}_{\mathcal{DS},\mathcal{A}}(k) = P[Exp^{uf\text{-}cma}_{\mathcal{DS},\mathcal{A}}(k) = 1]$$

Si la ventaja es despreciable para todo algoritmo $\mathcal{A}$, entonces el esquema es seguro en el sentido *UF-CMA*[3], pues ningún adversario puede crear una firma válida sin tener acceso a la clave privada.

[3]En inglés: *Unforgeability under chosen-message attack.*

D . Esquema de encriptación asimétricos y firmas digitales key-insulated

La idea principal detrás del concepto *key-insulated* es minimizar el daño causado tras un ataque que comprometa la clave privada en un sistema de encriptación asimétrico o firmas digitales. En el modelo propuesto en [12], el tiempo se divide en N periodos, tales que, la(s) clave(s) secretas se almacenan en un dispositivo posiblemente inseguro, pero estas son actualizadas en cada período. La manera en que son actualizadas las claves es interactuando con otro dispositivo físicamente protegido, el cual mantiene una clave secreta maestra que se mantiene fija en todos los períodos. Todos los computos son hechos en el dispositivo inseguro en donde la clave secreta para el período actual puede ser comprometida. La clave pública se mantiene fija para todos los períodos.

Un esquema (t, N)-key-insulated es tal que un adversario que ataca el dispositivo inseguro y obtiene la clave secreta en t perídos de tiempo distintos es incapaz de obtener las claves secretas de los demás $N - t$ períodos.

D .1. Esquema

Definición 15 *Un esquema de cifrado* ***key-updating*** *es una 5-tupla de algoritmos polinomiales* $\mathcal{G}$,$\mathcal{U}^*$,$\mathcal{U}, \mathcal{E}, \mathcal{D}$ *tales que:*

- $\mathcal{G}$*, el algoritmo generador de claves, es un algoritmo probabilista que toma como entrada un parámetro de seguridad* 1^k *y número total de períodos* N*. Retorna una clave pública* PK*, una clave secreta maestra* SK^* *y una clave secreta inical* SK_0*.*
- $\mathcal{U}^*$*, el algoritmo* ***key-update*** *para el dispositivo, es un algoritmo determinista que toma como entrada un índice de período* $i < N$ *y la clave secreta maestra* SK^**. Retorna una clave secreta parcial* SK'_i *para el período* i*.*
- $\mathcal{U}$*, el algoritmo* ***key-update*** *para el usuario, es un algoritmo determinista que toma como entrada un índice* i*, una clave secreta* SK_{i-1} *y una clave secreta parcial* SK'_i*. Retorna la clave secreta* SK_i *para el período* i *(y borra* SK_{i-1}, SK'_i*).*
- $\mathcal{E}$*, el algoritmo de cifrado, es un algoritmo probabilista que toma como entrada una clave pública* PK*, un período de tiempo* i*, y un mensaje* M*. Retorna el texto cifrado* $\langle i, C\rangle$*.*
- $\mathcal{D}$*, el algoritmo de descifrado, es un algoritmo determinista que toma como entrada una clave secreta* SK_i *y un texto cifrado* $\langle i, C\rangle$*. Retorna un mensaje* M *o el símbolo especial* $\perp$ *en caso de error.*

D .2. Seguridad

Para modelar ataques de exposición de claves, al adversario se le concede acceso a dos tipos de oráculos. El primero es el *oráculo de exposición de clave* $Exp_{SK^*,SK_0}(\cdot)$, el cual con entrada i, retorna la clave secreta temporal SK_i. El segundo es un oráculo de encriptación izquierda derecha, $LR_{PK,b}(\cdot,\cdot,\cdot)$, en donde $b = b_1, .., b_N \in \{0,1\}^N$, definido como: $LR_{PK,b}(i, M_0, M_1) \stackrel{def}{=} \mathcal{E}_{PK}(i, M_{b_i})$. El adversario tiene permitido ocupar ambos oráculos en forma adaptativa en cualquier orden. Finalmente el adversario puede tener acceso a un tercer oráculo $D^*_{SK^*,SK_0}(\cdot)$ que, con entrada $\langle i, C\rangle$ computa $D_{SK_i}(\langle i, C\rangle)$ con C no devuelto por el oráculo $LR(i,\cdot,\cdot)$. Esto modela un ataque de texto cifrado escogido por el adversario.

El vector b para el oráculo izquierda-derecha es escogido en forma aleatoria. El adversario gana el juego si puede adivinar el valor de b_i para cualquier período i no expuesto por el oráculo *Exp*. Informalmente, un esquema es seguro si para cualquier adversario PPT tiene ventaja despreciable.

Formalmente:

Definición 16 *Sea* $\Pi = (\mathcal{G}, \mathcal{U}^*, \mathcal{U}, \mathcal{E}, \mathcal{D})$ *un esquema de cifrado* ***key-updating****. Para el adversario* $\mathcal{A}$ *se define lo siguiente:*

$$\begin{aligned} Adv_{\mathcal{A},\Pi}(k) \stackrel{def}{=} \; & 2 * P[(PK, SK, SK_0) \leftarrow \mathcal{G}(1^k, N); b \stackrel{\$}{\leftarrow} \{0,1\}^N; \\ & (i, b') \leftarrow \mathcal{A}^{LR_{PK,b}(\cdot,\cdot,\cdot), Exp_{SK^*,SK_0}(\cdot), \mathcal{O}(\cdot)}(PK) : b' = b_i] - 1 \end{aligned}$$

En donde i nunca fue dado como entrada a $Exp_{SK^*,SK_0}(\cdot)$ *y* $\mathcal{O}(\cdot) = \perp$ *para un ataque sólo de texto plano y* $\mathcal{O}(\cdot) = D^*_{SK^*,SK_0}(\cdot)$ *para ataques de texto cifrado escogido.*

Se dice que Π *es* (t, N)-***key-insulated*** *bajo ataques de texto plano (resp. texto cifrado) si para cualquier algoritmo PPT* $\mathcal{A}$ *el cual no llama más de t veces al oráculo* $Exp_{SK^*,SK_0}(\cdot)$*,* $Adv_{\mathcal{A},\Pi}(k)$ *es despreciable en el parámetro de seguridad k, donde* $\mathcal{O}(\cdot) = \perp$ *(resp.* $\mathcal{O}(\cdot) = D^*_{SK^*,SK_0}(\cdot)$*).*

E . Funcionalidad $\mathcal{F}_{SYN}$

Funcionalidad $\mathcal{F}_{SYN}$:

- $\mathcal{F}_{SYN}$ espera su SID de la forma $sid = (sid', \mathcal{P})$, en donde $\mathcal{P}$ es una lista de participantes entre los cuales la sincronización será llevada a cabo. Procede de la siguiente forma.

1. En su primera activación, inicializa el contador de ronda $r \leftarrow 1$.

2. Al recibir entrada (`Enviar`,sid,M) de un participante $P \in \mathcal{P}$, en donde $M = \{(m_i, R_i)\}$ es un conjunto de pares de mensajes m_i y identidades de receptores $R_i \in \mathcal{P}$, guardar (P, M, r) y entregar como salida (sid,P,M,r) al adversario. (Si luego P es corrompido por el adversario, entonces se borra la tupla (P, M, r).)

3. Al recibir un mensaje (`Avanzar-Ronda`,sid,N) del adversario: Si existen participantes no corruptos $P \in \mathcal{P}$ para los cuales no se tiene guardado la tupla (P,M,r), entonces ignorar el mensaje. Si no:

   a) Interpretar N como la lista de mensajes enviados por los participantes corruptos en esta ronda. Esto es, $N = \{(S_i, R_i, m_i)\}$ en donde cada $S_i, R_i \in \mathcal{P}$,m_i es un mensaje, y S_i es corrupto. (S_i es el emisor del mensaje m_i y R_i es el receptor.)

   b) Preparar para cada participante $P \in \mathcal{P}$ la lista L_P^r de mensajes que le fueron enviados en la ronda r por todos los participantes en $\mathcal{P}$.

   c) Incrementar el contador de ronda: $r \leftarrow r + 1$.

4. Al recibir como entrada (`Recibir`, sid) de un participante $P \in \mathcal{P}$, entregar como salida (`Recibido`,sid,r,L_P^{r-1}) a P. (Sea $L_P^0 = \perp$.)

Figura 6.1: Funcionalidad de comunicación síncrona, $\mathcal{F}_{SYN}$ [8].

F . Diagrama de clases

Figura 6.2: Diagrama de clases general