Centro Politécnico Superior
Universidad de Zaragoza

Supporting Application's Evolution in Multi-Application Smart Cards by Security by Contract

Ingeniería en Informática
Proyecto Fin de Carrera

Realizado en:
Technical University of Denmark

Rubén Romartínez Alonso
Noviembre 2010

Director:
Nicola Dragoni
Department of Informatics and Mathematical Modelling
Technical University of Denmark

Ponente:
Álvaro Alesanco Iglesias
Departamento de Informática e Ingeniería de Sistemas
C.P.S, Universidad de Zaragoza

Supporting Application's Evolution in Multi-Application Smart Cards by Security by Contract

Resumen

La tecnología de Java Card ha evolucionado hasta el punto de permitir correr tanto servidores como clientes Web dentro de una tarjeta inteligente. Además, las tarjetas inteligentes actuales permiten tener instaladas en si mismas múltiples aplicaciones, las cuales pueden ser descargadas y actualizadas a lo largo de la vida de la tarjeta. Esta nueva característica de las tarjetas inteligentes las hace muy atractivas para ambos usuarios y desarrolladores de tarjetas inteligentes debido a las nuevas posibilidades que estas proveen. El uso de estas tarjetas inteligentes no supone ningún problema cuando las aplicaciones han sido instaladas antes de que la tarjeta haya sido sellada porque las interacciones entre las aplicaciones instaladas han sido comprobadas a priori con sus respectivas políticas. El problema surge cuando las aplicaciones pueden ser descargadas dinámicamente y la seguridad de la información intercambiada entre estas aplicaciones no puede ser asegurada. Por ello, el uso de las tarjetas inteligentes como tarjetas multi-aplicación es todavía extremadamente raro debido a que las aplicaciones en ellas instaladas, las cuales contienen información sensible, provienen de diferentes proveedores. Debido a esto es necesario un método que controle las posibles interacciones entre las aplicaciones instaladas.

Dado que los actuales modelos y técnicas de seguridad para tarjetas inteligentes no soportan este tipo de evolución, es necesario un nuevo método donde el comportamiento referente a la seguridad se ajuste con la política de seguridad de la tarjeta anfitriona en caso de nuevas descargas o actualizaciones. La conformidad entre el comportamiento de la tarjeta y la política de la tarjeta debe ser comprobada durante la petición de instalación o de actualización evitando la necesidad de los costosos métodos de monitorización en tiempo de ejecución. Además deberá asegurarse que no existirán fugas de información en su intercambio entre las aplicaciones. Este nuevo modelo propuesto, que será llamado seguridad por contrato (SxC usando las siglas en inglés) tratará con los posibles cambios tanto en los contratos de las aplicaciones como en los de la política de la plataforma dinámicamente.

En el presente PFC se presenta y desarrolla un modelo de políticas y contratos así como los algoritmos que se encargarán de asegurar la certificación de las aplicaciones. También, debido a la limitación de memoria en las tarjetas inteligentes el sistema será testeado con un ejemplo real.

Agradecimientos

I would like to thank to my supervisor Nicola Dragoni for give me the opportunity of developing this project which has allowed me to dive into the world of smart cards.

What is collected in this thesis is information that I found in articles or in books. A special thanks to the authors listed in the bibliography page. Without them, this thesis would have taken years off my life (and I don't have many to spare).

Thanks also to all the people who have shared with me this amazing year in Denmark; I'm never going to forget you.

A Álvaro Alesanco por la ayuda aportada en el desarrollo del proyecto y por aconsejarme que es mejor hacer las cosas de un modo correcto aunque haya que sacrificar algo que realizarlas aprisa y peor cuando las consecuencias pueden acarrear peores problemas.

Last but not least I would like to thank my parents their support during these months abroad. They guide me when I was lost in a foreigner country and best of all they gave me the unique chance to be a scientist for the science, with the only objective of knowledge, which probably won't be the same never again.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1
Introducción

1.1. Tarjetas Inteligentes

El uso de este tipo de dispositivos se ha incrementado notablemente a lo largo de los años debido tanto a su sencillez de uso como en las características de seguridad que proveen. Tanto las características hardware como software, así como las características de seguridad de las mismas son explicadas en detalle a lo largo de los Apéndices A y B. A continuación se ofrece una descripción de las características más importantes de estas tarjetas.

Dependiendo del modo en el que se comunican con las entidades externas se pueden clasificar como:

- Tarjetas de contacto: son las tarjetas que necesitan entrar en contacto con el lector de tarjetas a través de unos contactos metálicos. A través de estos contactos el lector alimenta eléctricamente a la tarjeta y provee un medio de comunicación.

- Tarjetas sin contacto: este tipo de tarjetas inteligentes funcionan gracias a un campo magnético creado por un lector externo que le provee energía y con el que se comunica mediante una antena a través de señales de radio frecuencia.

Ambas clases de tarjetas inteligentes están fabricadas con un chip integrado encargado de almacenar la información, llevar a cabo procesamiento local y ejecutar complejas operaciones. Las diferentes partes del chip, como pueden verse en la Figura A.3, son:

- Microprocesador: debido a que debe ser lo más fiable y seguro posible con el propósito de evitar fallos [19], los microprocesadores usados para la fabricación de las tarjetas inteligentes son modelos que han sido probados y mejorados durante largo tiempo.

- Memoria: dispone de tres tipos distintos de memoria, cada uno encargado de una tarea. La memoria ROM almacena datos como el Sistema Operativo (S.O) o algoritmos criptográficos durante toda la vida de la tarjeta. La memoria RAM almacena los datos recibidos de las entidades externas y variables y es borrada cuando la tarjeta se desconecta. La memoria EEPROM almacena la información variable de la tarjeta como las aplicaciones. Esta información se mantiene almacenada aun cuando la tarjeta está desconectada.

- Coprocesador: debido a que el microprocesador no es el más potente del mercado se incluyen otros procesadores más potentes para llevar a cabo tareas más complejas computacionalmente como la creación de claves criptográficas.

- Controlador Entrada-Salida: es el encargado del intercambio de información entre las entidades externas y el procesador.

Estas tarjetas inteligentes proveen tanto medidas de seguridad hardware como software que son explicadas en detalle en el Apéndice B.

1.2. Contexto del Problema

Aunque en la actualidad la mayoría de las tarjetas inteligentes son tarjetas con una sola aplicación, las tarjetas inteligentes son, básicamente, un ordenador con la suficiente memoria y poder computacional para gestionar varias aplicaciones así como almacenar información de diferentes fuentes y trabajar con ella. Por ello, años atrás se empezaron a desarrollar tarjetas inteligentes que pudieran soportar varias aplicaciones instaladas en ellas. Para ello se desarrollo un estándar entre varias organizaciones y empresas gubernamentales llamada GlovalPlatform con el objetivo de permitir portabilidad de S.O. y aplicaciones entre distintos desarrolladores de tarjetas. A partir de este estándar se desarrollaron varias implementaciones entre las que destacan Multos y Java Card. Tanto Global Platform como Java Card son ampliamente explicadas en el Apéndice A.

La utilización de Java Card en las tarjetas inteligentes amplia el concepto de tarjetas multi-aplicación añadiendo la posibilidad de descargar nuevas aplicaciones o modificar las existentes después de su emisión al mercado. Por ello las interacciones entre las aplicaciones necesitan ser controladas para evitar comportamientos indeseados (fugas de información). Aunque tanto Java Card como la misma tarjeta proveen métodos para el aislamiento de aplicaciones y compartición segura de información a través del cortafuegos mediante el uso de interfaces compartidas, ninguna de estas técnicas termina de resolver completamente este problema.

Por ello se propone el concepto de Seguridad-por-Contrato (SxC), donde cada aplicación va acompañada de un contrato que especifica su comportamiento. Así mismo, la plataforma define una política de seguridad que especifica el comportamiento de cada aplicación con respecto al resto de aplicaciones instaladas en la tarjeta. Cuando una aplicación llega a la tarjeta, su contrato es comparado con la política de seguridad de esta y, si cumple con los requisitos que esta impone, la aplicación es instalada y la política actualizada con el comportamiento de la nueva aplicación. En caso contrario, la instalación es rechazada para evitar la instalación de aplicaciones que puedan producir fugas de información o incluso el mal funcionamiento de la tarjeta.

1.3. Solución Aportada

Debido a que las tarjetas inteligentes tienen los recursos limitados, se define un sistema jerárquico en la definición de los contratos-políticas donde el nivel mas bajo permite la implementación de este sistema dentro de la tarjeta limitando la expresividad del mismo.

En este nivel de abstracción, el Contrato de las aplicaciones es definido como un conjunto de servicios que la aplicación provee, servicios que la aplicación invoca, servicios que la aplicación necesita y servicios que la aplicación permite usar y a qué aplicaciones se lo permite.

Con esta nueva definición, los cambios (eliminación o adición de servicios) en el contrato de las aplicaciones son definidos y clasificados como:

- Cambios que no pueden producir ningún tipo de error en la plataforma.
- Cambios que pueden producir errores funcionales y es necesario comprobarlos.

- Cambios que pueden producir errores de seguridad y es necesario comprobarlos.

Por último se define la política de la tarjeta como la unión de todas las políticas de las partes interesadas (proveedores de aplicaciones, autoridades de control, etc.) que indican el uso que se le puede dar a los servicios provistos por sus aplicaciones. Estas partes pueden modificar sus políticas si esta modificación no viola las antiguas reglas impuestas por el resto de las partes. Además, cuando una aplicación de un nuevo proveedor intenta instalarse en la tarjeta se comprueba la conformidad con la política de la tarjeta y, si se ajusta a ella, sus reglas son añadidas.

Con esta definición del tratamiento de la información se desarrollan teóricamente los algoritmos necesarios para el control de seguridad en las distintas modificaciones dentro de la tarjeta:

- Añadir una nueva aplicación
- Modificación de una existente (tanto política como contrato)
- Eliminación de una aplicación

La definición del funcionamiento de estas aplicaciones está definida a partir de los posibles cambios en los servicios. Además de estas funciones, se añaden otras funciones que realizan los cambios tanto del contrato de las aplicaciones como de la política de la plataforma tras una modificación.

1.4. Diseño y resultados

Una vez definida la implementación teórica del protocolo a desarrollar, se comienza con la implementación del prototipo diseñado. Primero es necesario realizar un programa que simule el entorno de una Java Card que se ajuste a la versión elegida para su desarrollo. Para ello se creará un applet que funcionará como punto de entrada del protocolo recibiendo la información procedente de las entidades externas de las cuales se obtendrán tanto la información de las aplicaciones como las acciones a realizar por el sistema. Con esta información se invocarán a los correspondientes algoritmos del protocolo, los cuales comprobarán la conformidad de la acción que se desea realizar y ejecutarán las acciones pertinentes (actualización de la política de seguridad y aplicaciones o rechazo de la acción). Por último, se realizará la validación del sistema mediante la medición del espacio necesario para poder almacenar la información necesaria para la ejecución del protocolo diseñado y con ello la viabilidad de exportar el sistema a un entorno real.

1.5. Descripción del Contenido

La memoria está compuesta por:

1. Análisis del problema detallando el marco del problema y como se ha orientado la solución del mismo.
2. Diseño de la solución donde se describe el diseño propuesto como solución al problema propuesto.
3. Implementación donde se dan los detalles técnicos de la implementación y en el que se presenta los tests realizados al sistema y el uso de memoria del sistema con un ejemplo real de la solución presentada.
4. Conclusión, donde se presentan las conclusiones obtenidas como la valoración del trabajo realizado.

5. Anexos donde se incluye la memoria en inglés realizada para la presentación de la misma en Dinamarca la cual incluye la batería de pruebas realizadas para el diseño del sistema.

Capítulo 2
Análisis del Problema

2.1. Introducción

Actualmente han surgido una gran cantidad de entornos donde las aplicaciones cooperan y comparten información y donde, además, estas aplicaciones pueden ser instaladas o modificadas añadiendo o eliminando algunas funcionalidades durante la vida del sistema. Esto es posible dado que en estos ambientes o la seguridad de la información instalada y compartida no es muy importante o se dispone de la suficiente potencia computacional para poder ejecutar herramientas que controlen la seguridad de la información compartida y almacenada en tiempo real. El problema surge cuando la información almacenada es la parte más importante del sistema y además no se dispone de suficiente potencia para usar una de esas herramientas, así que la seguridad de la información no se puede asegurar.

Este es el problema de las tarjetas inteligentes, los ordenadores más extendidos en la actualidad y que son usados en multitud de sistemas como tarjetas de transporte, tarjetas bancarias, tarjetas de acceso, etc. Como estas tarjetas inteligentes no soportan los costosos métodos que funcionan en tiempo real, la solución escogida para este sistema pasa por el desarrollo de estándares que ayuden a resolver los problemas de las tarjetas inteligentes multi-aplicación. Los estándares más comunes son Java Card, GlobalPlatform y MULTOS.

Aunque estos estándares fueron desarrollados para que diferentes proveedores pudieran cooperar en la misma tarjeta, en la actualidad en la mayoría de las tarjetas usadas como tarjetas multi-aplicación las aplicaciones provienen del mismo proveedor pues el problema sigue siendo la información compartida. Como las aplicaciones pueden ser añadidas, modificadas o eliminadas en tiempo real, es necesario poder asegurar a los emisores de las aplicaciones que los requisitos que se cumplirán cuando su aplicación fue añadida seguirán cumpliéndose a lo largo de la vida de su aplicación.

Esta información está ampliada en el Apéndice C con ejemplos y con el estado del arte explicados en detalle.

2.2. Seguridad por Contrato

La idea de Seguridad por Contrato (SxC) surge como solución al problema de intercambio ilegal de información entre aplicaciones, tanto en el momento de carga de las aplicaciones en la tarjeta como en las posibles modificaciones después de la emisión de esta.

El modelo de SxC surge la propuesta development-by-contract y de la idea Model Carrying Code (MCC) propuesto por Sekar et al. [26] diseñada y probada sistemas de código móvil y adaptada para tarjetas inteligentes. En este modelo, la aplicación llega a la plataforma con un contrato, que

es una descripción del comportamiento de la aplicación. La tarjeta inteligente dispone de la política de la plataforma, la cual describe el comportamiento de seguridad de la tarjeta, que es comparada con el contrato de la aplicación para ver si ambos encajan. En caso de ser así, la instalación o actualización de la aplicación es aceptada. Gracias a este método que comprueba que el contrato encaja con la política de la tarjeta, se evita el uso de los costosos métodos de monitorización en tiempo real que tendrían que realizarse en un sistema externo a la tarjeta.

Los problemas que se desean resolver son ([24]):

1. (Seguridad Estable) Una aplicación no puede interactuar con aplicaciones prohibidas que ya están instaladas en la tarjeta inteligente

2. (Funcionalidad Estable) Un cambio dinámico no debe afectar a la ejecución correcta de una aplicación en la tarjeta inteligente. En particular, una aplicación debe poder funcionar después de uno de estos cambios en la tarjeta.

   - Adición de una nueva aplicación a la tarjeta.
   - Actualización de una aplicación ya existente.
   - Cambios en la política de seguridad de la plataforma de la tarjeta.
   - Eliminación de una aplicación existente en la tarjeta.

2.3. Esquema de SxC

El esquema de trabajo que sigue el model de SxC es el representado en la figura 2.1. Como se puede observar, primero se comprueba que la evidencia es correcta, lo que puede hacerse mediante una firma digital de confianza o, para el caso de las tarjetas inteligentes como una prueba de que el código satisface el contrato. Una vez comprobado que esta evidencia es fidedigna, la plataforma comprueba que este se ajusta a la política que la tarjeta quiere hacer cumplir. En casi afirmativo, la aplicación puede ser ejecutada asegurando que durante su ejecución solamente las interacciones declaradas serán permitidas.

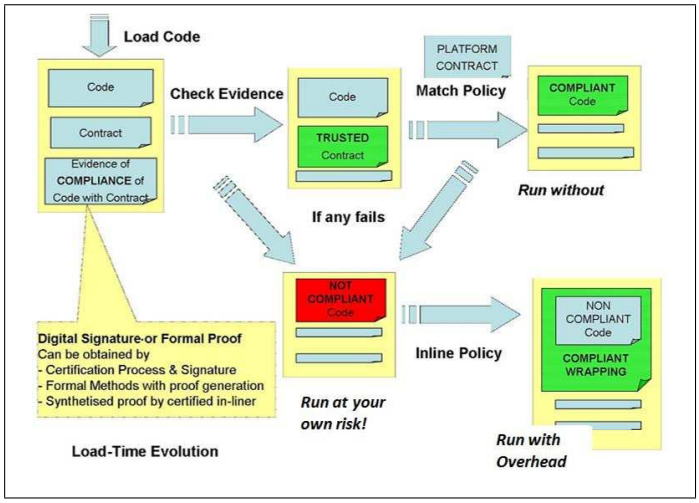

Figura 2.1: Esquema de trabajo SxC

2.4. Jerarquía de Modelos Política/Contrato

Debido a la existencia de diferentes sistemas en los que este modelo puede ser aplicado se definen tanto la política como los contratos utilizando diferentes niveles de expresión teniendo en cuenta la potencia computacional requerida para la ejecución del modelo como las limitaciones de expresividad dadas por el entorno. De este modo, en los niveles más bajos se obtienen beneficios computacionales pero perdiendo expresividad en la definición de contratos y políticas.

- L0: Aplicaciones como servicios. Este nivel modela aplicaciones como una lista de servicios requeridos y provistos.
- L1: Flujo de control permitido. Este nivel provee un grafo G1(A) de la aplicación, donde los vértices son los estados de la aplicación y los arcos representan la invocación de diferentes servicios. Con este modelo se puede conseguir un control de acceso basado en el historial y un control del intercambio de información más fino.
- L2: Flujo de control permitido y deseado. Este nivel añade al anterior las nociones de estados correctos e incorrectos. Puede ser necesario si se quiere comprobar si la eliminación de una aplicación (o un cambio en la política) no interfiere en funcionamiento de otras aplicaciones.
- L3: Flujo completo de información. Este nivel extiende el anterior considerando también el intercambio de información entre las variables.

Moverse de un nivel a otro superior supone añadir más detalle en la especificación del contrato/política, modelar de forma más precisa el comportamiento de las aplicaciones corriendo en la tarjeta.

2.5. Limitaciones del Entorno

Debido a las limitaciones computacionales presentadas por las tarjetas inteligentes, en los niveles L1, L2 y L3 la comparación contrato/política debe realizarse en el exterior de la tarjeta recayendo en la necesidad de securizar la comunicación mediante métodos criptográficos así como buscar una tercera parte de confianza donde se ejecute esta comprobación. Por ello, el nivel 0 es el seleccionado para la implementación de este modelo en la tarjetas inteligentes. Debido a que las actuales definiciones del nivel 0 no especifican el comportamiento real de las aplicaciones si no sólo el intercambio de información entre las aplicaciones, será necesario otra definición que solucione este problema.

Capítulo 3
Diseño de la Solución

Como se ha nombrado en el capítulo anterior, la actual definición del nivel cero dada por Dragoni et al en [24] no da una representación real del comportamiento de las aplicaciones de la tarjeta, por lo que es necesario la modificación de este modelo con el fin de obtener un modelo donde la seguridad de la información entre las aplicaciones pueda ser asegurado.

3.1. Marco de SxC

Como se explica en el capítulo anterior, el modelo de SxC se basa en comprobar la correcta correspondencia entre las distintas aplicaciones y la política de la tarjeta.
Un contrato es definido como la descripción formal completa del comportamiento de una aplicación en lo que respecta a acciones de seguridad. En este contrato se almacena tanto el comportamiento de esta aplicación con respecto a las demás como el comportamiento deseado de las demás hacia ella misma. El contrato de una aplicación es proporcionado por el emisor de la aplicación.
Así mismo, se define política de seguridad como una especificación formal completa del comportamiento de las aplicaciones que van a ser ejecutadas en la plataforma y el cual se refiere a acciones que conciernen a la seguridad. La política inicial de la tarjeta es definida por el emisor de la tarjeta y actualizada con la información proveniente de todos los emisores de aplicaciones de la tarjeta.
Una vez definidos los términos contrato y política de las tarjetas, se define la correspondencia contrato-política como:
Definición (Correspondencia contrato-política): *el contrato de una aplicación encaja con la política de la plataforma si no hay intercambio ilegal entre la aplicación a instalar y las aplicaciones ya instaladas en la tarjeta.*

3.2. Soportando la Evolución de las Aplicaciones con SxC

Debido a la falta de expresividad de la definición actual del nivel 0, es necesario realizar una nueva definición tanto del contrato como de la política de la tarjeta para resolver este problema y poder asegurar que la información almacenada e intercambiada en la tarjeta no va a ser comprometida.

3.2.1. Contrato de las Aplicaciones

Como se puede ver en la Figura 2.1, el contrato provisto por el emisor de la aplicación es comparado con el código de la aplicación para comprobar su correspondencia, por lo que es lógico pensar que este contrato pueda ser extraído directamente del código de la aplicación. De este modo, las únicas restricciones existentes para la invocación de alguno de los servicios que esta aplicación provee son las impuestas por el código. Puesto que es normal que los dueños de las aplicaciones apliquen restricciones sobre quién puede usar los servicios que sus aplicaciones proveen y es lógico que esta información se encuentre en el contrato, la información del contrato es dividida en dos

partes: el claim y la política de la aplicación. El claim describe el comportamiento de la aplicación con el resto de aplicaciones de la tarjeta, mientras que la política de la aplicación se refiere a cómo las otras aplicaciones que comparten servicios dentro de la tarjeta deben interactuar con ella. Este último es establecido por los proveedores de la aplicación, los dueños de los dominios seguros o las autoridades controladoras. Para cada aplicación en la tarjeta, el claim es un par (servicios invocados, servicios provistos). Los servicios invocados representan los servicios que el código de estas aplicaciones van a usar durante su ejecución en la tarjeta. Por otra parte, los servicios provistos forman un subset de los servicios que la aplicación ha implementado y suministra a la plataforma por lo que otras aplicaciones pueden usar. La política de la aplicación también es definida como un par (servicios permitidos, servicios necesitados). En este caso los servicios necesitados son un subset de los servicios invocados por la aplicación. Estos servicios serán necesarios durante la invocación de la aplicación. Por otra parte, los servicios permitidos es un set de pares, donde uno de los elementos es un servicio del set de proveídos y el otro el identificador de la aplicación a la que se le permite el uso. Esta aplicación puede estar o no instalada en la tarjeta. Con esta clasificación, un contrato queda representado como:

Aplicación	Claim		Política	
	Provistos	Invocados	Necesarios	Permitidos
Medicina@Farmacia	lista_medicinas	-	-	(lista_medicinas, ap1@Farmacia) (lista_medicinas, ap2@Hospital)

Tabla 3.1: Ejemplo de Contrato

Tras estas definiciones, el intercambio de información entre aplicaciones puede ser explicado usando los set concretos de servicios. Existe intercambio de información entre dos aplicaciones cuando el servicio provisto por una de ellas es invocado por la otra. Imaginemos dos aplicaciones A y B, existirá un intercambio de información entre ellas si existe una entrada en la lista de servicios provistos de la aplicación A llamada servicioA y una entrada en la lista de servicios invocados de la aplicación B llamada servicioA. Pero aunque exista un intercambio de información entre estas aplicaciones, no significa que este intercambio sea legal. Un intercambio legal se define como:
Definición (Intercambio legal): *Un intercambio de información es legal cuando la aplicación que provee el servicio permite a las aplicaciones que lo invocan usarlo.*

Con esto se puede concluir que la definición del Contrato puede ayudarnos a tratar con ambos: fallos funcionales (P2) y fallos de seguridad (P1) en la plataforma. Los posibles cambios junto con los posibles fallos derivados de estos son:

- Cambios en el Claim:
  - Añadir un servicio a la lista de servicios invocados puede provocar un fallo de seguridad debido a que puede ser que esa aplicación no tenga permiso para usar el servicio.
  - Eliminar un servicio de la lista de servicios invocados no puede provocar ningún fallo de seguridad.
  - Añadir un servicio a la lista de servicios provistos puede provocar un fallo funcional porque este puede ser invocado por una aplicación instalada en la plataforma pero no tener permitido su uso.
  - Eliminar un servicio de la lista de servicios provistos puede provocar un fallo funcional debido a que es posible que alguna aplicación necesite el servicio provisto.

- Cambios en la política de la aplicación:
  - Añadir un servicio a la lista de servicios necesarios puede provocar un fallo funcional si el servicio añadido no es provisto por las aplicaciones de la plataforma.
  - Eliminar un servicio de la lista de servicios necesario no puede provocar un fallo funcional.
  - Añadir un par (servicio, aplicación) a la lista de servicios permitidos no puede provocar un fallo funcional.
  - Eliminar un par (servicio, aplicación) de la lista de servicios permitidos puede provocar un fallo de seguridad si la aplicación referenciada en el par usa el servicio referenciado.

3.2.2. Política de Seguridad de la Tarjeta

Como se ha dicho con anterioridad, la política de seguridad inicial de la tarjeta es provista por el emisor de la tarjeta. Debido a la capacidad de evolución de las tarjetas después de la emisión es necesaria la definición de una política general para tarjetas inteligentes. Esta política debe estar formada y provista por todos los participantes en la tarjeta (proveedores de aplicaciones, dueños de dominios seguros y autoridades controladoras). Cuando un nuevo participante llega a la tarjeta, primero será necesario comprobar que su política encaja con el resto de políticas de sus compañeros y, segundo, deberá proveer sus propios requerimientos para las aplicaciones de otros participantes e incluir estos en la política de la tarjeta. Además, será posible la actualización de la política de la tarjeta por parte de los participantes modificando sus propias políticas, siempre y cuando esta modificación siga ajustándose a la política anterior.

Por lo tanto, se define política de la tarjeta como la combinación de las políticas de las aplicaciones almacenadas en la tarjeta. Esta información puede ser reordenada en dos sets, PolNeeds y PolAllows, dependiendo en el objetivo de los requerimientos de seguridad. El set PolNeeds se representa como $\{Dependientes_1,..., Dependientes_n\}$ donde cada elemento es una lista de las aplicaciones que necesitan alguno de los servicios que la $Aplicacion_i$ provee. Para el caso del set PolAllows, este es representado como $\{Permitidos_1,..., Permitidos_n\}$ donde cada elemento es la lista de servicios permitidos de la correspondiente aplicación.

3.2.3. Algoritmos para la Evolución a Tarjetas Inteligentes Multi-aplicación

Una vez que el problema ha sido expuesto, los algoritmos desarrollados para poder resolver el problema van a ser explicados. Estos algoritmos están divididos en dos clases, los algoritmos que actualizan la política y los algoritmos que comprueban la conformidad entre la actual política y los posibles cambios a realizar (actualización del contrato de una aplicación, instalación y eliminación de aplicaciones). Los algoritmos del primer grupo son:

- Actualización de la política al añadir una aplicación nueva
- Actualización de la política al eliminar una aplicación de la plataforma
- Actualización de la política al ocurrir un cambio en la política de una aplicación

Y en el segundo grupo:

- Comprobar la conformidad al añadir una nueva aplicación.
- Comprobar la conformidad al eliminar una aplicación.

- Comprobar la conformidad al modificar la política de la aplicación.
- Comprobar la conformidad al modificar el contrato de una aplicación.

El pseudocódigo de estos algoritmos está presentado en el Apéndice D.

3.2.3.1. Comprobación de Correspondencia de una Nueva Aplicación/ Check of Compliance of New Application

Cuando una nueva aplicación quiere ser instalada en la tarjeta inteligente es necesario comprobar si el contrato de la aplicación encaja con la actual política de la plataforma. Para realizar esto, el Algoritmo 1 realiza tres comprobaciones para asegurar que el nuevo contrato no puede provocar ningún fallo. La primera parte comprueba, por cada servicio que la nueva aplicación tiene en la lista de servicios invocados, si existe alguna aplicación en la plataforma que lo provee y que le permite usarlo. Si ninguna aplicación en la plataforma provee este servicio, su uso no puede producir ningún fallo y se comprueba el siguiente servicio. Si la aplicación que provee el servicio que la aplicación necesita no le deja usarlo (no hay una entrada en la lista de permitidos con el servicio y el identificador de la aplicación), la aplicación no cumple las condiciones de conformidad con la política y se rechaza la instalación. Por otra parte, si el servicio es provisto y la aplicación que lo provee también le permite usarlo, se comprueba el siguiente servicio hasta terminar con todos y seguir con el siguiente paso. En la segunda parte se comprueban los servicios de la lista de servicios necesitados. Para esta comprobación sólo es necesario mirar si todos los servicios de esta lista están provistos por una de las aplicaciones en la plataforma. Si todos lo están, se comprueba el siguiente paso y, si no, se cancela la instalación. Finalmente, para la tercera parte, los servicios provistos por la aplicación son comprobados. Si los servicios provistos no son invocados por ninguna aplicación en la plataforma, la aplicación es instalada. Pero si algún servicio es invocado, la aplicación que lo invoca debe tener permiso para ello. Si lo posee, se continúa con la siguiente, en caso contrario la instalación se cancela.

Algorithm 1 Check of Compliance of New Application

Require: $Contract_A, Policy_{\ominus}$.
Ensure: True if the application is compliance, false otherwise.

```
1: for s_A service in Calls_A do
2:   for A_j in Λ do
3:     for s_j in Provides_{A_j} do
4:       if s_A = s_j AND (s_A, A) ∉ Allows_{A_j} then
5:         return false
6: for s_A service in Needs_A do
7:   for A_j in Λ do
8:     if s_A ∈ Provides_B AND B ∉ Λ then
9:       return false
10: for s_A service in Provides_A do
11:   for A_j in Λ do
12:     if s_A ∈ Calls_{A_j} AND (s_A, A_j) ∉ Allows_{A_j} then
13:       return false
14: return true
```

3.2.3.2. Comprobación de la Eliminación de una Aplicación/ Check of Removal of Application

El Algoritmo 2 es el responsable de comprobar que no hay aplicaciones en la plataforma que dependa de la aplicación que va a ser eliminada. Para conseguir esto sólo se necesita comprobar que la lista Dependientes de esta aplicación está vacía. Si esta lista esta vacía porque ninguna aplicación necesita los servicios que ella provee, la aplicación será eliminada de la plataforma. En caso contrario se cancela su eliminación para evitar fallos funcionales.

Algorithm 2 Check of Removal of Application

Require: Application A, $Policy_{\ominus}$.
Ensure: True if it is possible to remove the application, false otherwise.

```
1: for e element in PolNeeds do
2:   if e.ID = A.ID then
3:     if e.Application_list = ∅ then
4:       return true
5:     else
6:       return false
7: return true
```

3.2.3.3. Comprobación de Correspondencia de la Actualización de la Política de una Aplicación/ Check of Compliance of AppPolicy Update

Cuando un proveedor de aplicaciones quiere modificar la política de una de sus aplicaciones añadiendo o eliminando servicios, será necesario comprobar mediante el Algoritmo 3 que esta política está conforme con la política de la tarjeta. Para ello se tendrán que comprobar los casos explicados con anterioridad sobre que cambios en la política de las aplicaciones puede provocar errores en el funcionamiento. Primero se comprueban los nuevos servicios añadidos a la lista de servicios necesarios. Es necesario realizar esta comprobación porque cada servicio de esta lista debe ser provisto por una aplicación en la plataforma. Para ello, se extraerán las diferencias entre la antigua lista de servicios y la nueva y se comprobará si para cada uno de estos servicios hay una aplicación que los provee. Si no es así, la actualización es cancelada. A continuación se comprueban los pares eliminados de la lista de permisos. Si alguno de los servicios de los pares de esta lista está todavía siendo invocado por alguna aplicación en la plataforma, la aplicación no puede ser eliminada y se cancela la acción. En cambio, si ninguno de los servicios de los pares eliminados es usado, la actualización puede ser realizada.

3.2.3.4. Comprobación de Correspondencia de un Nuevo Contrato con la Política/ Check of New Contract Compliance with the Policy

Si, en vez de modificar sólo la política de la aplicación se modifica todo el contrato, a parte de las comprobaciones anteriores es necesario comprobar las modificaciones correspondientes al Claim (Algoritmo 4). En cuanto a los cambios en el Claim, primero se compraban las modificaciones en la lista de servicios invocados. Para cada servicio añadido a esta lista es necesario confirmar si este es proporcionado por alguna aplicación en la plataforma. Si no es proporcionado no hay que comprobar nada más y se sigue con el siguiente servicio, pero si es proporcionado hay que comprobar también que la aplicación que lo provee le permite usarlo. En caso de no ser así se cancela la actualización. Por último, se tienen que comprobar las diferencias en la lista de servicios provistos. En este caso hay que comprobar ambos, los nuevos servicios añadidos y los antiguos eliminados. En el primero

Algorithm 3 Check of Compliance of AppPolicy Update

Require: $AppPolicy_{ANEW}$, $AppPolicy_{AOLD}$ where A $\in \Lambda$, $Policy_{\ominus}$.
Ensure: True if the update is compliance, false otherwise.

```
1: if Needs_ANEW ≠ Needs_AOLD then
2:    X ← s ∈ Needs_ANEW / Needs_AOLD
3:    for s service in X do
4:       if s ∉ Provides_Λ then
5:          return false
6: if Allows_ANEW ≠ Allows_AOLD then
7:    X ← e ∈ Allows_AOLD / Allows_ANEW
8:    for e element in X do
9:       for A_j in Λ do
10:         if e = (s, A_j) AND s ∈ Calls_Aj then
11:            return false
12: return true
```

de los casos (los servicios eliminados) hay que comprobar que estos servicios ya no son necesarios para las aplicaciones en la plataforma. Si alguno de estos servicios todavía es necesario, se cancela la actualización. Para los servicios añadidos hay que comprobar que, si alguno de esto nuevos servicios es invocado por alguna aplicación, esta tenga permiso para usarla. Como antes, si esto no se cumple se cancelara la instalación. Si todos estos pasos son superados, la actualización se llevará a cabo.

3.2.3.5. Actualización de la Política después de la Instalación de una Aplicación/ Policy Update for Approved New Application

Una vez que se ha confirmado que el contrato de la nueva aplicación encaja con la política de la tarjeta, es necesario añadir el contrato a la plataforma y modificar la política de la tarjeta. Para su realización, el Algoritmo 5 se ha separado en tres partes: la primera modifica la información almacenada en la antigua política, la segunda añade a la política las listas de Dependientes y de Permitidos de la nueva aplicación y la tercera añade el nuevo contrato a la lista de las aplicaciones. Para la primera parte se recorren los servicios necesitados por la nueva aplicación. Cada elemento de la lista es comparado con los servicios provistos en la plataforma. Si este servicio es provisto, el identificador de la nueva aplicación es añadido a la lista de Dependientes de la aplicación que provee el servicio. En la siguiente parte se crea una lista vacía que es añadida a la lista PolNeeds y su lista de Permitidos con el identificador de la nueva aplicación. Por último la lista de aplicación es actualizada con el contrato de la nueva aplicación.

3.2.3.6. Actualización de la Política después de la Eliminación de una Aplicación/ Policy Update for Approved Application Removal

Una vez que se ha confirmado la eliminación de la aplicación, esta puede ser eliminada de la plataforma. Para realizar esta acción es necesario actualizar la política de la plataforma y eliminar el contrato de la lista de aplicaciones. Como puede verse en el Algoritmo 6, las listas de Dependientes y de Permitidos son eliminadas de sus respectivas listas. Después de esto, se recorren las listas de Dependientes del resto de aplicaciones y se borran las referencias a la aplicación. Una vez realizado esto el contrato se elimina de la lista de aplicaciones.

Algorithm 4 Check of New Contract Compliance with the Policy

Require: $Contract_{A^{NEW}}$, $Contract_{A^{OLD}}$, $Policy_{\ominus}$.
Ensure: True if the new Contract is compliance, false otherwise.

```
1: if Needs_ANEW ≠ Needs_AOLD then
2:   X ← s ∈ Needs_ANEW / Needs_AOLD
3:   for s service in X do
4:     if s ∉ Provides_Λ then
5:       return false
6: if Allows_ANEW ≠ Allows_AOLD then
7:   X ← e ∈ Allows_AOLD / Allows_ANEW
8:   for e element in X do
9:     for A_j in Λ do
10:      if e = (s, A_j) AND s ∈ Calls_Aj then
11:        return false
12: if Calls_ANEW ≠ Calls_AOLD then
13:   X ← s ∈ Calls_ANEW / Calls_AOLD
14:   for s service in X do
15:     for B in Λ do
16:       if s ∈ Provides_B then
17:         if (s, B) ∉ Allows_B then
18:           return false
19: if Provides_ANEW ≠ Provides_AOLD then
20:   X ← s ∈ Provides_AOLD / Provides_ANEW
21:   for s service in X do
22:     for B in Λ do
23:       if s ∈ Needs_B then
24:         return false
25:   X ← s ∈ Provides_ANEW / Provides_AOLD
26:   for s service in X do
27:     for B in Λ do
28:       if s ∈ Calls_B AND (s, B)∉ Allows_ANEW then
29:         return false
30: return true
```

Algorithm 5 Policy Update for Approved New Application

Require: $AppPolicy_A$, $Policy_{\ominus}$.
Ensure: Updated $Policy_{\ominus}$ with the new AppPolicy.

```
1: for s_A service in Needs_A do
2:   for A_j in Λ do
3:     for s_j in Provides_Aj do
4:       if s_A = s_j then
5:         Dependents_Aj.addElement(A)
6: Dependents_A = ∅
7: Policy_⊖.PolNeeds.addElement(Dependents_A)
8: Λ.addElement(A)
9: return Policy_⊖
```

Algorithm 6 Policy Update for Approved Application Removal

Require: A, $Policy_{\ominus}$.
Ensure: Updated $Policy_{\ominus}$ without the Application information.
1: PolNeeds.removeElement($Dependents_A$)
2: PolAllows.removeElment($Allows_A$)
3: **for** e element in PolNeeds **do**
4: **for** B application in $Dependents_e$ **do**
5: **if** B = A **then**
6: $Dependents_e$.removeElement(A)
7: Applications.removeElement(A)
8: **return** $Policy_{\ominus}$

3.2.3.7. Actualización de la Política después de un Cambio Aprobado en la Política de una Aplicación/ Update of the Policy after Approved AppPolicy Change

Si el cambio en la política de la aplicación ha sido confirmado, la política de la plataforma debe ser actualizada mediante el uso del Algoritmo 7. Lo primero de todo hay que extraer los servicios añadidos a la lista de servicios necesarios. Cada uno de estos servicios es comparado con los servicios provistos por las aplicaciones. Si el servicio es provisto por alguna de ellas, el identificador de la aplicación modificada es añadido a la lista de Dependientes de la aplicación que provee este servicio. A continuación se extraen los servicios eliminados de la lista de servicios necesarios. Como en el caso anterior, se buscan las aplicaciones que proveen estos servicios. Una vez encontradas, se borra de su lista de Dependientes el identificador de la aplicación modificada. Por último se almacena la nueva lista de Permitidos donde antes estaba almacenada la antigua.

Algorithm 7 Update of the Policy after Approved AppPolicy Change

Require: $AppPolicy_{A^{NEW}}$, $AppPolicy_{A^{OLD}}$, $Policy_{\ominus}$.
Ensure: Updated $Policy_{\ominus}$ with the new AppPolicy change.
1: **if** $Calls_{A^{NEW}} \neq Calls_{A^{OLD}}$ **then**
2: X $\leftarrow s \in Calls_{A^{NEW}}/Calls_{A^{OLD}}$
3: **for** s service in X **do**
4: **for** B in Λ **do**
5: **if** $s \in Provides_B$ **then**
6: $Dependents_B = Dependents_B + A$
7: Y $\leftarrow s \in Calls_{A^{OLD}}/Calls_{A^{NEW}}$
8: **for** s service in Y **do**
9: **for** B in Λ **do**
10: **if** $s \in Provides_B$ **then**
11: $Dependents_B = Dependents_B - A$
12: **return** $Policy_{\ominus}$
13: **if** $Allows_{A^{NEW}} \neq Allows_{A^{OLD}}$ **then**
14: $Allows_A = Allows_{A^{NEW}}$
15: **return** $Policy_{\ominus}$

Capítulo 4
Implementación del Nivel 0 de SxC

Este capítulo examina:

- La información de la tecnología usada para el desarrollo del programa que resuelve el problema presentado en los capítulos anteriores.
- Implementación del sistema, incluyendo la búsqueda de información necesaria.
- Los problemas surgidos durante la implementación del sistema tanto con la tecnología como con la teoría del sistema.
- Batería de test para comprobar el correcto funcionamiento del protocolo propuesto.
- Por último, una evaluación del uso de memoria del sistema implementado tanto durante la ejecución del mismo como cuando la tarjeta esta desconectada.

4.1. Tecnología Adoptada

Para la resolución del problema presentado se ha seleccionado Java Card debido al hecho de ser el estándar más importante en uso así como el más sencillo, común y documentado. En la actualidad las versiones más usadas y modernas son: Java Card v2.2.2 y Java Card v3.0.1. Aunque es cierto que la primera de las dos, la primera es la más usada en los sistemas actuales, la versión 3.0.1 añade muchas nuevas características además de ser la versión que más se usará en un futuro próximo. Aunque es cierto que el sistema se podría desarrollar con ambas versiones, se ha elegido la más moderna porque el desarrollo es más sencillo con las nuevas herramientas y facilidades añadidas.

4.2. Implementación

Esta sección está divida en cuatro partes. La primera expone las estructuras de datos seleccionadas para la implementación del sistema y por qué han sido seleccionadas. La segunda parte explica en detalle la implementación de los algoritmos expuestos en el capítulo cuatro. La siguiente explica el sistema completo desarrollado. La última parte evalúa el espacio usado por cada una de las estructuras desarrolladas. Tras esto, se presenta y evalúa un ejemplo real del sistema.

4.2.1. Estructura de Datos

Primero de todo se decidió que el mejor método para la implementación de los sets definidos en la parte teórica era usando vectores. Se eligió vectores primero, porque esta clase acababa de ser añadida a Java Card con la versión 3.0 y, segundo porque esta clase permite la creación de estructuras anidadas, lo que no se permite con buffers, que era la otra posibilidad para la representación

de estos. Además, la clase Vector permite la creación de sets pequeños e incrementarlos después automáticamente cuando la capacidad inicial del vector no es suficiente para almacenar nuevos elementos. Este incremento puede ser definido por el programador dependiendo de los requerimientos del sistema. Para el desarrollo de este sistema los vectores han sido creados con tamaño inicial e incremento igual a uno con el objetivo de disminuir el tamaño inicial requerido por el sistema. Por último, se han considerado tres opciones para almacenar la información de los servicios y los identificadores de las aplicaciones: buffers, strings y stringbuffers. Entre usar strings o stringbuffers se eligió la primera ya que los stringbuffers están implementados para ser usados con cadenas que son modificadas y en este sistema la información es permanente. Por otra parte, entre elegir strings o buffers, ambos son similares en la manera de almacenar la información, pero los buffers ofrecen varias desventajas. Primero, el tamaño de los buffers debe ser definido antes de saber el tamaño de la información que va a almacenar mientras que el tamaño de las cadenas puede ser definido en tiempo de ejecución. Además, strings ofrecen varios métodos que ayudan tanto en la comparación de cadenas como en la búsqueda de similitudes.

4.2.1.1. Contrato

El diseño del contrato (Lista E.1) está basado en la parte teórica y está dividido en dos partes: el claim y la política de la aplicación. Por ello el contrato se ha desarrollado como un registro que almacenará en un campo el claim y en el otro la política. Además, se añade un campo extra para poder almacenar el identificador de la aplicación a la que pertenece el contrato. El constructor de esta estructura inicializa el identificador con el nombre de la aplicación y rellena los campos claim y política con los datos recibidos como parámetros. Por otra parte, el claim es definido como el par (servicios invocados, servicios provistos) donde ambos son listas de servicios. Por ello, esta estructura (Lista E.2) ha sido desarrollada como un registro con dos campos donde se almacenan estos dos sets. Cada uno de estos campos ha sido implementado como un vector de cadenas que almacenan la información de los servicios que provee e invoca respectivamente. Esta estructura dispone de dos constructores, uno que crea una instancia vacía de la estructura cuando no hay información disponible y otra que llena los respectivos campos con la información recibida. La política de las aplicaciones ha sido también definida como un par (permisos, servicios necesarios). Los servicios necesarios es una sublista de los servicios invocados. Por otro lado, permisos (Lista E.3) es un set de pares donde cada elemento tiene un servicio de la lista de servicios provistos y el otro elemento es el identificador de una aplicación. Parecido a como se define el Claim, la política de la aplicación (Lista E.4) ha sido implementado como un registro con dos vectores que almacenan la información de los servicios necesarios y lo permisos. Además, añade un campo con el identificador de la aplicación a la que pertenece esta información necesaria para los algoritmos en los que se modifica únicamente la información de la política.

4.2.1.2. Política

Como ha sido explicado en el capitulo anterior, la política está representada por dos grupos: (PolNeeds, PolAllows). PolNeeds almacena para cada aplicación las aplicaciones que necesitan algún servicio de ella. Cada campo de este vector ha sido implementado (Lista E.5) como un registro con un cadena que almacena el identificador de la aplicación y un vector con las aplicaciones que necesitan de sus servicios. Un ejemplo de este vector puede ser ($Dependiente_{A1}$,..., $Dependiente_{AN}$), donde $Dependiente_{A1}$ es un registro con el identificador de la aplicación = $Aplicacion_{A1}$ y un vector = ($Aplicacion_{A2}$,..., $Aplicacion_{AM}$), donde estas aplicación necesitan al menos uno de los servicios ofrecidos por esta aplicación y $m < n$ & $n = aplicaciones\ en\ la\ plataforma$. Por otra parte, PolAllows (Lista E.6) almacena para cada aplicación la lista de permisos de la aplicación y ha sido

implementada de la misma forma que PolNeeds pero almacenando un permiso por elemento. Por último la política (Lista E.7) es un registro que almacena las estructuras PolNeeds y PolAllows.

4.2.2. Implementación de los Algoritmos

La implementación de los algoritmos se ha realizado a partir de la descripción teórica de los mismos explicada en la sección anterior. Además, esta sección esta desarrollada en detalle en el Apéndice E.

4.2.3. Implementación del Sistema

Aunque son los algoritmos los que en realidad implementan el marco del protocolo presentado, para la comprobación del funcionamiento de este es necesario simular el funcionamiento de una tarjeta inteligente. La parte correspondiente a los algoritmos y las estructuras de datos han sido empaquetadas en dos clases: Data_Structures y Multi_Applica- tion_Framework. El primero almacena las estructuras de datos y el segundo, a parte de los algoritmos necesarios para simular el sistema, implementa también:

- Un algoritmo que comprueba la existencia de una aplicación en la plataforma comparando el identificador de la aplicación con las existentes en la plataforma.
- Dos funciones más, las cuales sirven para comprobar la información almacenada en la plataforma.
- Dos funciones más para almacenar los cambios en los contratos después de la actualización.

Pero la parte principal del sistema, la que simula el funcionamiento de la tarjeta, es la clase Input_applet. Este applet recibirá los comandos APDU del sistema externo de la tarjeta, comprobará su corrección y realizará las acciones necesarias dependiendo del comando recibido. Primero dispone de una función que recibe los comandos APDU, comprueba la corrección de estos (longitud del comando y acción a realizar) y los clasifica dependiendo de la acción que se va a realizar con ellos (byte CLA). Los posibles valores son:

- 0x20 para añadir una nueva aplicación
- 0x30 para eliminar una aplicación de la tarjeta
- 0x40 para modificar la política de una aplicación de la tarjeta
- 0x50 para modificar el contrato de una aplicación en la tarjeta

Si la instrucción no se corresponde con ninguno de estos se devuelve un error al sistema externo. En caso contrario, esta función llamará a las distintas funciones que extraerán de estos mensajes la información necesaria (contrato, política o identificador de la aplicación) para realizar las acciones pertinentes. Estas funciones son:

1. *manage_check_of_compliance* es la función encargada de las acciones de añadir una nueva aplicación y de modificar una existente. Para ello extrae del comando APDU el contrato de la aplicación o su política dependiendo de la acción a realizar. Con el identificador de la aplicación se comprueba si esta existe en la tarjeta para evitar la existencia de dos aplicaciones con el mismo nombre o la modificación de una aplicación que no existe. Una vez comprobado esto, se invoca a la correspondiente función del protocolo.

2. *manage_policy_update_remove_app* esta función únicamente extrae el identificador de la aplicación a eliminar de la tarjeta e invoca a la función correspondiente. Si ninguna aplicación depende de ella la aplicación será eliminada de la tarjeta. En caso de no existir la aplicación, se devuelve un error al sistema externo de la tarjeta.

Este applet también almacena la información que en el sistema real se almacena en la tarjeta: la política de la plataforma. También tiene la referencia a la clase que implementa las funciones del protocolo desarrollado.

4.2.3.1. Problemas

Los principales problemas surgidos durante la implementación del sistema son debido a las limitaciones de Java Card en el uso de las funciones que proporciona. En el caso de la clase Vector, algunas funciones que podrían haber facilitado la implementación del sistema no están presentes. Por ejemplo, de haber existido la función *clone()*, los constructores de las estructuras que poseen estos habrían sido más sencillos. Solamente asignando a las variables de la clase los datos de entrada con la función clone, no sería necesario la implementación de los bucles. Otro de los problemas es que, al disponer de algunas estructuras dobles dentro de los vectores como por ejemplo con Permitidos, no es posible el uso de las funciones suministradas por la clase Vector.

4.2.4. Batería de Tests

Para cada algoritmo implementado se han realizado una serie de tests en los cuales se suministran distintos tipos de entradas (correctas e incorrectas) para comprobar si los algoritmos funcionan como han sido definidos siguiendo la especificación realizada de SxC. En la Tabla 4.1 se muestran los resultados de dichas pruebas. Además, dichas pruebas pueden ser encontrados en el Apéndice G donde se muestran los comandos APDU de entrada y se explican en detalle los resultados obtenidos.

4.3. Rendimiento

Dado que uno de los principales problemas en las tarjetas inteligentes es el espacio disponible para almacenar la información, esta debe ser tenida en cuenta a la hora de asegurar que lo implementado en el simulador encajará en un entorno real. Para comprobar el espacio usado por el applet desarrollado se han implementado varios algoritmos que comprobarán el espacio de cada estructura y del sistema completo. El comprobador de memoria ha sido implementado en la clase *MemoryTestBench*, la cual se compone de dos algoritmos. El primero de ellos se llama *calculateMemoryUsage* y es el encargado de calcular la memoria usada por la clase recibida como parámetro. Para ello, se inicializa tanto la variable sobre la que va a crear el objeto a testear y las variables que van a almacenar el espacio ocupado antes de crear el objeto y después. Una vez inicializadas, se llama al recolector de basura para que elimine de la memoria toda la información que no esta siendo usada y se mide el espacio ocupado. Después de esto se crea el objeto a medir, se vuelve a llamar al recolector de basura y se vuelve a medir el espacio ocupado. Restando estos dos valores se obtiene el espacio que ocupa el objeto a medir, el cual es devuelto al algoritmo principal que mostrará los resultados. Para poder comprobar las diferentes estructuras instanciadas en el programa, se crea una interfaz publica llamada *Object_test*, la cual tiene una función abstracta llamada *makeObject* y que se encarga de devolver la instancia del objeto a medir. Para cada una de las estructuras implementadas se crea una nueva interfaz que extiende Object_test donde se implementa la clase a medir. Por último, la clase *Main* es la que crea la instancia de la clase memoryTestBench y la que invoca a cada una de las clases mostrando por pantalla el espacio ocupado por cada una de ellas.

Tabla 4.1: Resultado Pruebas

Nombre	Parte	Tipo de Prueba	Resultado
Check of compliance of new application	Servicios Invocados	Los servicios invocados no están presentes en la tarjeta	Aplicación Instalada
		Los servicios invocados están presentes y pueden ser usados	Aplicación Instalada
		Los servicios invocados están presentes pero no pueden ser usados	Aplicación Rechazada
	Servicios Necesarios	Los servicios necesarios están provistos por la tarjeta	Aplicación Instalada
		Los servicios necesarios no están provistos por la tarjeta	Aplicación Rechazada
	Servicios Provistos	Los servicios provistos no son invocados	Aplicación Instalada
		Los servicios provistos son invocados por una aplicación sin permiso	Aplicación Rechazada
		Los servicios provistos son invocados por una aplicación con permiso	Aplicación Instalada
Check of removal an application	Tiene dependencias	La aplicación a eliminar tiene servicios que otras aplicaciones necesitan	Aplicación No Eliminada
	No tiene dependencias	La aplicación a eliminar no tiene otras aplicaciones que la invoquen	Aplicación Eliminada
Check of compliance of AppPolicy update	Añadir servicios necesarios	Los servicios añadidos están provistos por la tarjeta	Actualización Aceptada
		Los servicios añadidos no están provistos por la tarjeta	Actualización Rechazada
	Eliminar permisos	La aplicación a la que se le quitan los permisos no está en la tarjeta	Actualización Aceptada
		La aplicación existe pero no usa el servicio del permiso	Actualización Aceptada
		La aplicación existe y usa ese servicio	Actualización Rechazada
Check of new Contract compliance with de Policy	Añadir invocaciones	El servicio añadido no está provisto por la tarjeta	Actualización Aceptada
		El servicio añadido está provisto y tiene permisos para usarse	Actualización Aceptada
		El servicio añadido está provisto pero no tiene permisos	Actualización Rechazada
	Servicio eliminado	El servicio es necesario por otra aplicación	Actualización Rechazada
		El servicio no es necesario para ninguna aplicación	Actualización Aceptada
	Servicio añadido	El servicio añadido no es invocado	Actualización Aceptada
		El servicio añadido es invocado pero no tienen permisos	Actualización Rechazada
		El servicio añadido es invocado y tienen permisos	Actualización Aceptada

4.3.1. Resultados

En esta sección se realiza un estudio sobre las necesidades del espacio necesario de cada estructura de datos implementada en el sistema. Estos datos son introducidos en varias tablas correspondientes a cada una de las estructuras las cuales pueden encontrarse en el Apéndice E. Tras esto, se realiza un estudio del espacio necesario de un sistema real.

4.3.1.1. Cadenas

Aunque las cadenas son internalizadas por Java, lo que significa que sólo una de las instancias de la misma cadena es guardada en memoria, es necesario saber el espacio que ocuparía cada una de ellas en el peor caso para poder maximizar el espacio requerido. Una cadena almacena:

- 8 Bytes para la clase
- 16 Bytes para la cadena de caracteres
- 4 Bytes para el Offset
- 4 Bytes para la longitud de la cadena
- 4 Bytes para el cómputo
- 4 Bytes para el Hash
- 2 Bytes para cada carácter en la cadena

Por lo tanto, una cadena de 6 caracteres ocupará 40 bytes para almacenar la información de la cadena y 12 bytes para la información en si. Para la medida de espacio de cada estructura las cadenas no han sido tenidas en cuenta hasta el cálculo final del sistema.

4.3.1.2. Permitidos

Permitidos es una estructura que almacena dos cadenas de caracteres. Los resultados pueden ser vistos en la Tabla E.1 y muestran que se necesitan 8 bytes para la clase y 8 bytes más para los punteros a las cadenas.

4.3.1.3. Claim

Claim es una estructura que almacena dos vectores de cadenas. Como se muestra en la Tabla E.2, el objeto usa 8 bytes para la instancia de la clase, 8 bytes más para los punteros de los vectores y:

- 40 bytes para el vector vacío inicializado con una posición vacía.
- 8 bytes más para cada dos elementos añadidos debido al padding.

4.3.1.4. Política de la Aplicación

Esta estructura está compuesta por una cadena y dos vectores, uno de cadenas y otro con Permitidos. Como en las clases anteriores, se necesitan 8 bytes para la clase y 4 más para el puntero de cada objeto con un total de 16 bytes debido al padding. Los vectores ocupan 40 bytes

vacíos y crecen de la siguiente manera: el vector de cadenas como en el Claim, 8 bytes cada dos elementos, y el vector de Permitidos crece 16 bytes por cada elemento añadido y 8 bytes cada dos elementos para los punteros.

4.3.1.5. Contrato

Como el contrato es básicamente una estructura que almacena las dos clases anteriores, su tamaño es la suma del tamaño de estas añadiéndole: 8 bytes para la clase, 16 bytes para los punteros de las clases y la cadena y el tamaño de la Política y el Claim (Tabla E.4).

4.3.1.6. Aplicaciones

Aunque esto no es una clase, es donde se almacena los contratos de las aplicaciones de la tarjeta. El espacio necesario para almacenarlo es, como en los vectores anteriores: 40 bytes para el vector vacío, 8 bytes por cada dos elementos añadidos y el tamaño de los contratos almacenados en el vector. En la Tabla E.5, la primera columna corresponde con el tamaño de los elementos añadidos al vector en orden. La segunda columna representa el numero de elementos en el vector, la tercera el tamaño del vector sin el tamaño de los elementos y la ultima el tamaño total.

4.3.1.7. PolAllows

Esta clase almacena únicamente un vector de Permitidos más las constantes de la clase. Estas son: 8 bytes para la clase, 8 bytes para el puntero y el padding y 40 bytes para el vector vacío. Por cada elemento añadido al vector el tamaño se incrementa en 16 bytes y ocho bytes más por cada dos elementos.

4.3.1.8. Dependientes

Este estructura sólo amacena un vector de cadenas por lo que el tamaño de la clase sólo depende del número de datos en su interior. Si el vector está vacío necesitará 8 bytes para la clase, 8 bytes para el puntero y padding y 40 bytes para el vector. Por cada dos elementos añadidos al vector, el espacio necesario crecerá en 8 bytes.

4.3.1.9. Política

Esta estructura consiste en dos vectores, uno de Dependientes y otro de PolAllows, lo que significa que el tamaño total será la suma del tamaño de los datos de estos más las constantes. Esto es: 8 bytes para la clase, 8 bytes para los punteros de los vectores y 40 bytes para los vectores vacíos. Para entender un poco mejor cómo crece el tamaño de estos vectores, se han separado tres casos. En el primero (Tabla E.8), el número de elementos en los vectores es constante e igual a uno, y lo que cambia el tamaño de estos elementos dependiendo del numero de servicios que almacenan. En el segundo (Tabla E.9), el número de elementos en los dos vectores cambian, pero el tamaño de los objetos añadidos es siempre el mismo e igual a 56 y 72 bytes respectivamente. El último caso (Tabla E.10) es una mezcla de los dos anteriores donde ambos el numero de elementos y el tamaño de cada uno varia.

4.3.2. Caso de Estudio

Una vez que cada clase a sido medida separadamente, es el momento de medir cuánto puede ocupar un sistema completo de la vida real. El sistema completo está compuesto por tres elementos: la política de la plataforma, las aplicaciones en la tarjeta y las constantes necesarias en el programa

Application	Policy	
	PolNeeds	PolAllows
EMV@BANK	ePurse@BANK	(transaction, ePurse@BANK) (fill_purse, ePurse@BANK)
ePurse@BANK	jTicket@Transport jTicket@Transport eTicket@SAS	(payment, jTicket@Transport) (account_balance, jTicket@Transport)
jTicket@Transport	-	-
weather@Sky	-	(weather_info, travel@Sky) (weather_RSS, travel@Sky)
eTicket@SAS	-	-

Tabla 4.3: Política

principal. Este sistema está compuesto por cinco aplicaciones con distintas relaciones entre ellas como sería en un sistema real. En la tabla 4.2 se muestra las cinco aplicaciones integradas ya en el sistema e insertadas en el orden de aparición en la tabla. Durante esta inserción, la política ha sido creada y queda como se muestra en la Tabla 4.3.

Application	Claim		Policy	
	Provides	Calls	Needs	Allows
EMV@BANK	transaction fill_purse	-	-	(transaction, ePurse@BANK) (fill_purse, ePurse@BANK)
ePurse@BANK	payment account_balance	fill_purse transaction	fill_purse	(payment, jTicket@Transport) (account_balance, jTicket@Transport)
jTickect@Transport	buy_ticket	payment payment	payment payment	-
Weather@Sky	weather_info weather_RSS	-	-	(weather_info, travel@Sky) (weather_RSS, travel@Sky)
eTicket@SASTravel	-	-	fill_purse payment	-

Tabla 4.2: Contrato

Con esta información, los datos son insertados en las diferentes tablas para obtener el tamaño total del sistema. Los resultados pueden verse en la Tabla 4.4 sin tener en cuenta las constantes necesarias para la ejecución del sistema, que ocupan unos 70 bytes. Con estos valores se obtiene que el espacio necesario para almacenar la información del sistema presentado anteriormente es de, aproximadamente, 3 kilobytes. A estos valores es necesario añadirles el espacio ocupado por las cadenas. Por ello, asumiendo que cada cadena es almacenada independientemente en memoria, el tamaño total necesario sería de unos 4,5 Kilobytes. Este valor es muy grande comparado con el espacio necesario para almacenar el resto del sistema, pero aun con este valor, el espacio total necesario es de unos 7 kilobytes. Este valor, comparado con el espacio que tiene una tarjeta inteligente que puede ser de 128 kilobytes, es una pequeña parte de lo que se puede almacenar, lo que indica que las tarjetas inteligentes actuales disponen de suficiente espacio de memoria no volátil para llevar el sistema desarrollado a un ambiente real. Además, esto contempla el caso en el que el espacio que ocupa cada cadena se mide de manera independiente. Si se mide tomando solo una copia por cada una de ellas el espacio necesario se reduce a 700 bytes para las cadenas y a 4

kilobytes para el sistema completo.

	Claim	AppPolicy	Contract	PolAllows	PolNeeds	System
EMV@BANK	104	144	272	96	64	
ePurse@BANK	112	144	280	96	64	
jTicket@Transport	104	122	240	56	56	
weather@Sky	104	144	272	96	56	
eTicket@SAS	96	112	232	56	56	
Total	1352			780		2140

Tabla 4.4: Resultado del Sistema Completo

Una vez calculado el resultado teórico del test se lleva este al sistema desarrollado en el simulador con una tarjeta de 128 kilobytes, donde se introducen las distintas aplicaciones y se comprueba tanto obteniendo los datos del simulador como del sistema con los sistemas de test que el espacio calculado por las aplicaciones es el calculado anteriormente.

Capítulo 5
Conclusiones y Trabajo Futuro

5.1. Resultados

Los resultados obtenidos del desarrollo de este proyecto fin de carrera son:

- El desarrollo de un protocolo seguro que permite la instalación, modificación y eliminación de aplicaciones dentro de una tarjeta inteligente multi-aplicación permitiendo la compartición de información entre las aplicaciones y evitando la fuga de información y el posible mal funcionamiento de las aplicaciones en la tarjeta.
- La implementación de un sistema que simula el funcionamiento de una tarjeta inteligente con el protocolo definido implementado en él.
- Una batería de pruebas que comprueban el funcionamiento del sistema para diversas entradas.
- Medición del espacio de memoria requerido por el sistema desarrollado y que demuestra que el sistema desarrollado encaja en las limitaciones de las tarjetas inteligentes.

5.2. Conclusiones

A lo largo de este proyecto fin de carrera se ha presentado la tecnología Java Card centrándose en las características de seguridad de las mismas. Tras esto ha sido presentado el actual problema con las tarjetas inteligentes multi-aplicación. Para la resolución de este problema se ha presentado el marco de SxC, describiéndolo en detalle y se ha aportado una implementación tanto teórica como práctica para solucionar los problemas de seguridad en las tarjetas inteligentes multi-aplicación. En este sistema, según la aproximación definida en él, cada aplicación que quiere ser instalada en la tarjeta va acompañada de la especificación de su comportamiento seguro, el cual debe ajustarse a la política de seguridad impuesta por la plataforma de la tarjeta anfitriona. En particular, se ha mostrado como la aproximación del SxC puede ser usada para resolver diferentes problemas de la seguridad de las tarjetas inteligentes, como los problemas del intercambio ilegal de información entre las aplicaciones de una misma tarjeta o el problema de preservar el estado seguro de la tarjeta después de haberse realizado cambios dinámicos en las aplicaciones o sus políticas. El marco ha sido definido en el primer nivel de la jerarquía llamada nivel 0 del modelo para tarjetas inteligentes.

Con esta aproximación se ha realizado la implementación del sistema que simula la tarjeta inteligente y del marco del SxC. La implementación del sistema ha sido testeada ejecutando diferentes test en el sistema usando el simulador del entorno de Java Card. Estos test muestran que el sistema implementado siguiendo la aproximación propuesta es válida y funcional y soluciona los problemas de la aproximación de primer nivel en sistemas reales.

También las medidas de espacio del sistema demuestran no sólo que el sistema implementado se puede almacenar en una tarjeta inteligente, si no que puede almacenar la información de varias aplicaciones sin llegar a preocuparse por el uso que la política y el contrato almacenados lleguen a necesitar usando la tecnología actual del mercado. Cogiendo estos resultados juntos, se observa que el sistema implementado esta en disposición de ser exportado a sistemas reales. Esto será un gran paso en las tarjetas inteligentes multi-aplicación las cuales todavía no están siendo usadas como tal, integrando este marco desarrollado en las Java Cards.

5.3. Trabajo Futuro

Como posibilidades de continuación con este trabajo, las opciones consistirían en la extensión de los contratos actuales con interacciones indirectas entre aplicaciones y con la ordenación de el árbol de invocaciones de servicios. Esta extensión requerirá modelos más complejos de representación de los contratos de las aplicaciones, como por ejemplos grafos de flujo. La parte problemática de esta nueva representación sería el encontrar un punto medio entre la posibilidad de ejecutar esta nueva representación de los modelos en la tarjeta con la expresividad de estos modelos. Otra solución sería proveer recursos computacionales externos para la resolución de la correspondencia entre la política y los contratos con este nuevo modelo, pero extendiendo el modelo actual flujo de trabajo y marco con métodos de cifrado y descifrado para asegurar que los resultados de la verificación externa permanecen intactos.

5.4. Problemas Encontrados

El primero de los problemas encontrados fue la búsqueda de información tanto sobre la base del proyecto (tarjetas inteligentes) como del marco del problema (tarjetas multi-aplicación) y sobre la nueva infraestructura a usar. En cuanto a la búsqueda de información de lo primero, el problema fue bastante grande porque se dispone de mucha información, sobre todo de tarjetas inteligentes, de la cual mucha de ella está anticuada o incluso mucha de la información disponible era contradictoria entre diferentes fuentes, teniendo que clasificarla y contrastarla para poder discernir cual de ellos tenía razón. En el segundo caso el problema fue justo el contrario ya que casi no existe información sobre tarjetas inteligentes multi-aplicación. Por último, el buscar información sobre las posibles infraestructuras para elegir la opción correcta y más tarde llegar a familiarizarse con esta. Aunque la elección fuera Java Card para su implementación, fue necesario el familiarizarse con ella pues está bastante limitada en cuanto a funciones con lo que estaba acostumbrado normalmente con Java además de ser necesario trabajar con estructuras de bajo nivel a las que no estaba acostumbrado.

Aunque el desarrollo del protocolo parecía en un principio bastante complicado, el trabajo en el que se basaba estaba bien fundamentado y la información recibida tanto por el supervisor como con los colaboradores me ayudo mucho. Además las constantes correcciones recibidas por parte de estos ayudaron a que, en caso de desviarme del camino correcto, no tardar mucho en darme cuenta.

El siguiente problema encontrado fue encontrar el simulador para Java Card. Según la información obtenida de los foros y páginas web, la mejor opción era usar Eclipse con IDE, pero este simulador solo permitía usar la versión 2.2.2 de Java Card. Como el uso de esta versión suponía el no poder crear estructuras de datos complejas, lo cual era una de las principales necesidades para la implementación del sistema. Por ello se eligió NetBeans como simulador, aunque este simulador no disponía de ninguna herramienta que permitiera la simulación del sistema externo de la tarjeta por lo que toda la información enviada a la tarjeta se tenía que hacer mediante scripts realizados a mano con el consiguiente consumo de tiempo.

5.5. Incidencias

La experiencia de la elaboración de este proyecto ha sido muy gratificante tanto debido a las condiciones de trabajo en las que se ha desarrollado como los conocimientos aportados tanto en punto de vista técnico como en el modo de llevar a cabo el trabajo (entrevistas con el tutor, búsqueda de información en solitario, colaboración con otras personas, etc.).

Primero el tener que familiarizarse con una tecnología de la que no sabía prácticamente nada como son las tarjetas inteligentes, además de la búsqueda de un simulador que permitiera el desarrollo de los objetivos que al principio todavía no estaban del todo claro en la fase de diseño fue bastante difícil. Además de las limitaciones de Java Card respecto a las capacidades que ofrece este sistema a las que no estaba acostumbrado me llevó al principio muchos problemas, aunque al final llegará a controlarla.

Además, el haber realizado esta clase de proyecto en un país extranjero ha hecho más difícil la realización de todo lo mencionado, lo que requería mucho más esfuerzo por mi parte así como la escritura de esta clase de proyectos en inglés, que requiere un gran dominio de vocabulario técnico como el desarrollo de estructuras complejas. Por último, el realizar una presentación en otro idioma durante treinta minutos, cuando ya es difícil realizarla español me parece una experiencia increíble.

Por último, la necesidad de organización, constancia, ser riguroso a la hora de obtener un buen resultado de lo buscado han sido necesarios para llevar este proyecto a buen puerto, más teniendo en cuenta que estando de Erasmus es muy fácil abstraerse y darse a otras tareas.

Como resumen, la realización de este proyecto podría ser la base para el desarrollo de futuros proyectos puesto que, desde el principio, se tenían que tener en cuenta detalles como tiempo necesario para realizar las tareas, posibles incidencias, etc. y que a la hora de llevar esto a la vida laboral serán muy positivas.

5.6. Gestión del Tiempo y del Esfuerzo

Esta sección da una idea de la manera en que el tiempo ha sido gestionado en el desarrollo del proyecto, así como del esfuerzo empleado en el mismo en las distintas fases. Para ello se han separado las diferentes fases de realización del proyecto y se ha detallado el tiempo necesario para la realización de dichas tareas. De estos resultados se obtiene el diagrama de Gant mostrado al final de esta sección.

5.6.1. División de tareas

La división de tareas se ha realizado de la siguiente manera:

- Estudio previo donde se engloba toda la consulta, lectura y búsqueda de bibliografía durante las distintas fases del proyecto.
- Diseño donde se incluye la definición del protocolo tanto en solitario como conjuntamente con el tutor así como el diseño en pseudocódigo de los algoritmos que forman parte del protocolo.
- Familiarización con el entorno donde se incluye tanto la búsqueda, prueba y posterior elección del simulador e IDE a utilizar así como el diseño de pequeñas aplicaciones y seguimiento de tutoriales para un aprendizaje básico.
- Implementación del sistema diseñado.

- Evaluación tanto de las corrección del diseño propuesto para el problema a solucionar como del sistema implementado. Además se incluye un análisis de memoria del sistema desarrollado.
- Documentación donde se incluye la generación de documentación del proyecto tanto para la memoria de Dinamarca como para la de España.

5.6.2. Diagrama de distribución de tiempos

En la figura 5.1 se muestra la distribución de tareas a lo largo del desarrollo del PFC mediante un diagrama de Gantt. En este se observan las distintas fases del proyecto y como se superponen unas a otras en algunos casos o la necesidad de terminar con una de las fases para poder continuar con las siguientes.

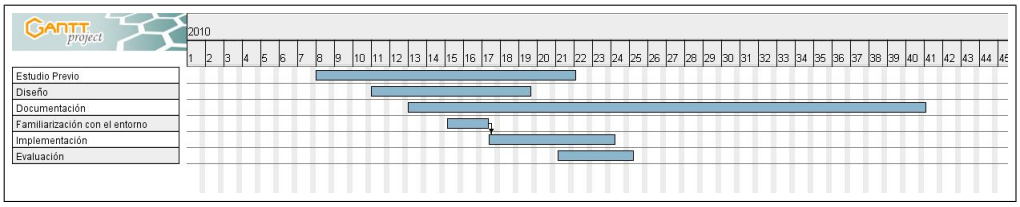

Figura 5.1: Distribución de tiempos

Apéndice A

Smart Card Technology

This chapter will state the current smart card technology and it is divided into three parts. The first part introduces the smart cards as a personal device and makes a general overview about how they work. The second part deals with the current architecture of the smart cards, explaining more in detail each element of the cards and which is its function in the smart card. The third part will introduce the multi-applications smart cards explaining in detail the actual standards for multi-applications smart cards and finally focusing in the specific case of Java Cards.

A.1. Background

The recently interest in smart cards along these last few years is due to those cards have the power and the speed of a computer and secure mechanisms to store data all included in a small card you can carry in your pocket wherever you.

Besides, magnetic cards, which have been used since the early 1960's until current days, cannot provide the same security features as smart cards do in order to avoid fraud or information leaks. Also these cards were not able to carry out with the pertinent functions and information on it. Instead of that, the smart cards are able to store this information allowing them performing transactions without accessing to remote databases.

These cards can be either contact or contactless cards depending on how they communicate with the off-card systems.

In the first one (contact smart cards), the chip establishes direct contact with the off-card system, which supplies energy and a communication channel to interchange information [19].

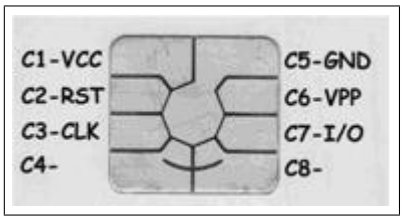

Figura A.1: Contacts of a typical contact smart card

As it can be seen in the Figure A.1, contact smart cards' chips have eight pads which, according to standards, are:

- Power supply voltage (VCC)

- Reset (RST)
- Clock signal (CLK)
- Ground (GND)
- Write Voltage (VPP)
- Input/output (I/O)
- Two more reserved to future uses

In the case of contactless cards A.2, these are activated due to a magnetic field created by the off-card system through which smart cards receive the energy supply to be powered on.

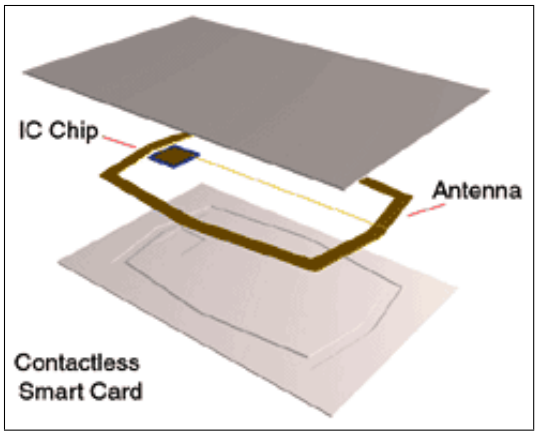

Figura A.2: Inside a contactless smart card

Contactless smart cards use an antenna, which work by Radio Frequency (RF) sending the data through the air, to communicate with the off-card system

A.2. Architecture

Smart cards are developed with an embedded Integrated Chip (IC) which could store data, carry out local processing and execute complex operations.

This IC has a microprocessor and a physically protected memory thanks to which the smart card is able to process and store several amount of data.

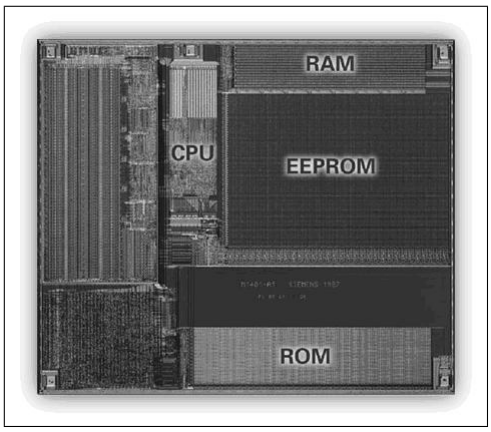

Figura A.3: Smart card chip

The different parts which can be seen in the Figure A.3 are explained below.

A.2.1. Microprocessors

Due to security reasons, the processors should be very reliable and secure in order to prevent failures [19]. So, the microprocessors used in the manufacture of smart cards are not the last model in the market but there are the models that has been used and proved for a long time. Moreover, if they develop their own microprocessor it will be much more expensive.

A.2.2. I/O Controller

The I/O controller is the responsible of manage the data interchange among the off-card entities and the processor.

A.2.3. Memory

It is the second most important part after the processor. Each card has three different kinds of memories:

- ROM: is the one which stores the Operating System instructions of the smart card since it only stores the permanent data which would not be erased.
- RAM: is where either the temporal data resulting of calculations done by the processor or the temporal data from the I/O are stored due to they are going to be erased when the card will be out of energy supply.
- EEPROM: is where the data related to the applications is stored because of the property of this kind of memory, which can store data permanently and erase this information when it becomes unnecessary or it changes. This memory should store the data during a long period of time without energy supply and write and erase the data many times.

A.2.4. Coprocessors

Owing to microprocessors must be very reliable, these cannot be the most powerful and faster. Hence, it is not possible to these microprocessors to do some complex calculations necessaries to

the cryptographic algorithms with the software they incorporate. So, some new smart cards add others processors especially developed to this purpose. The most important are [19]:

1. Coprocessors which develop cryptographic algorithms: Since long time ago coprocessors which calculate algorithms like DES and, since DES is not secure enough, AES have been developed. But sometimes some applications need to store certificates and to generate/validate signatures. So it is necessary to use public cryptographic algorithms like RSA or elliptic curves. To achieve this, processors which can calculate several basic operations necessaries to carry out these algorithms have also been developed.

2. Coprocessors which generate Random-numbers: since the use of random numbers is necessary when you want to generate a key or use some cryptographic algorithms, the random numbers should be genuine random numbers to avoid the breaking of the keys and not pseudo-random numbers. To achieve this, the algorithms which generate these numbers must use environmental conditions as temperature, but trying to make impossible predict these numbers if someone manipulates these conditions.

A.2.5. Standards

To assure the compatibility among the different classes of smart cards and all the different terminals which are nowadays in the market, all of them should follow the established standards defined in the ISO-7816 [1] developed as a extension of the standard ISO-7810 for magnetic cards. This standard covers the complete characteristics of smart-card technology where fifteen different parts like the physical characteristics, electric characteristics, cryptographic information application, commands for application management in a multi-application environment and so on are included.

A.3. Multi-application Smart Cards

In this section the Global Platform ([7]) standard for multi-application smart cards is explained and, after that, the specific case of Java Card is exposed.

A.3.1. What is a multi-application Smart Card?

Although nowadays the majority of smart cards are single applications cards like SIM cards for mobile phones, smart cards are basically a PC which has enough space and computational power to run several applications. They can also store data from different sources and work with this information. Due to that, years before card manufacturers started to develop smart cards which could be able to consolidate multiple applications itself. This idea started since both end users and issuers will be benefited with this. First, the end-users can replace all the cards they carry in only one which can be used for payments and other transactions. Secondly, the new multi-application smart cards will create a new market which will benefit to the issuers making possible to create new personalized applications. Besides, multi-application smart cards help optimizing costs.

With these multi-application smart cards now not only the card user and the issuer of the cards are present in the system, besides third parties and services providers are and could use the functions the card supplies.

But these new feature do not create only benefits, it create some problems, starting with the data the applications have in the smart cards. Where before the smart card only needs to keep safe the information when it went to the off-card systems encrypting the data, now they should

isolate this information from the other applications in the card. Due to that, an open standard was created and different implementations of this standard were developed.

A.3.2. Global Platform

Global Platform ([7]) is an organization boosted by public companies, industries and governments which was created to develop a global infrastructure for single and multi-application smart cards.

The principal target of this organization is to develop a standard considering neither the software nor the hardware the smart cards are going to use and interact with. The final point of this specification is to, in the worst case, be able to migrate from one card technology to another if any problems arise without significant problems.

Nowadays, Global Platform devotes their efforts to specify the terminals, cards and management systems.

From this standard, several technologies have been developed which, nowadays, the most important are Java Card and MULTOS.

A.3.2.1. Card Architecture

The smart card architecture is divided into several components with the purpose of reach neutral interfaces, to both hardware and software providers, with the applications and the off-card systems.

All the applications included in a smart card should have neutral Application Programming Interface (API) to allow the portability.

Global Platform does not bind to use a specific Runtime Environment (RTE) technology. Furthermore, the smart card includes several security domains which can guarantee absolute isolation among issuer, application provider and controlling authorities.

Finally, the card manager is the responsible of the central administration of the Global Platform.

The different components of the Global Platform card architecture (Figure A.4) are [7]:

- Security domains [27]: are the domains which support security services as key handling, encryption, decryption, digital signature generation, verification for their providers' applications and several functions download and execute an application. These domains are assigned from the card's issuer to applications or controlling authorities to have the tools necessaries to communicate with the off-card entities securely, to check the providers and to have the cryptographic keys completely isolated from the other security domains. When an application provider wants to use some of their keys, the application associated to this provider communicates with its secure domain through the Global Platform API to obtain it. Three different secure domains are defined:

  1. The issuer domain, which is compulsory in all the cards.
  2. Another one to the applications providers, which is optional.
  3. And another for the control authorities, which is the responsible to enforce secure measures to all the applications on the card.

- Global Services Applications: these are the applications which provide services to the other applications like methods to verify the cardholder identity and so on.

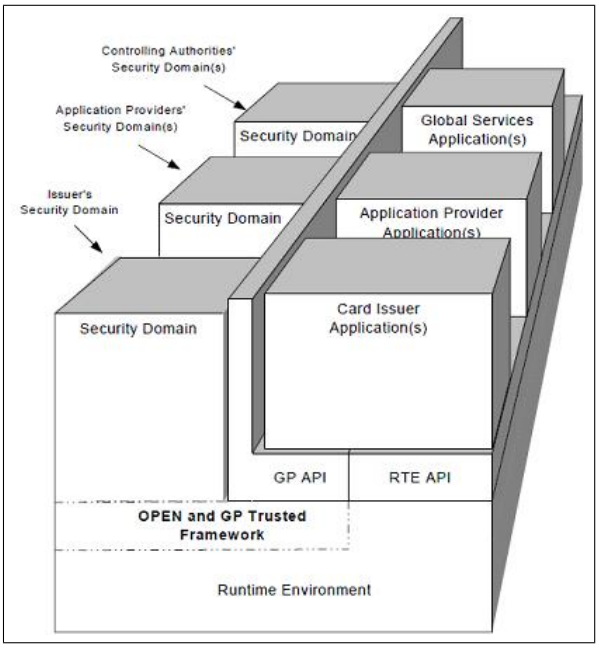

Figura A.4: Global Platform card architecture

- RunTime Environment (RTE): this is, as it has been aforementioned, a neutral API which provides, besides secure storage to the code and the data, a separated space (called context) where the application can execute its own code and services to have secure communication among the application and the entities off-card.
- Trusted Framework: it is the one which provide communication services among the applications in the card.
- Global Platform Environment (OPEN): it is the responsible to provide all the services of the RTE when it hadn't implemented or when the RTE had implemented it but not as the standard says. Additionally, when a new application wants to be installed in the card, OPEN is the manager of download the code and the installation of the application, assuring that the security principles imposed by the issuer are performed.
- Global Platform API: it is the one which provides services, like cardholder verification, and Card Content management services to applications.
- Card Manager: it is the central administrator of the card and it is represented by three entities:

  1. OPEN.
  2. Cardholder Verification Methods services.
  3. Issuer Security Domains.

A.3.2.2. Security Architecture

The goal of this architecture is to provide several secure mechanisms like data integrity, confidentiality, and authentication, which allows assuring the integrity and the security of all the

components to protect the card operation inside the system. The responsible to carry out with this security measures are [7]:

- Card Issuer: generating keys, enforcing standards and policies and so on.
- Application Providers: generating keys, providing applications, obtaining authorization to install new applications...
- Controlling Authorities: generating keys, controlling the applications to carry out with the standards and so on.
- On-card components as RTE, trusted framework, OPEN, etc.
- Back-end systems.

A.3.2.3. Cryptographic Support

The card must provide algorithms which allow: signature generation, symmetric cryptographic algorithms like DES and optionally asymmetric cryptographic algorithms like RSA which can assure data integrity, providers and other external entities authentication and secure messaging [7].

- Secure card content management: adds a new value to messages to verify the source and the message integrity.
- Secure communication: offers services with the aim that interchanged data between off-card and on-card is neither modified nor replaced for a third party. Provide three different services:

  1. Entity authentication: where the sender prove his identity to the receiver through a message interchange.
  2. Integrity and authentication: which allow the receiver to know if the message received is correct and if the message comes from the correct sender.
  3. Confidentiality: to assure that an external entity cannot understand (decrypt) the message.

A.3.3. Java Cards

A.3.3.1. Overview

Java card technology was invented in 1996 to allow applications written on Java language work in smart cards. Thanks to this technology, smart cards could incorporate the Java language advantages like Object Oriented programming, even though the most important advantage is to store and manage several applications in the same card (Figure A.5), including download, installation and so on.

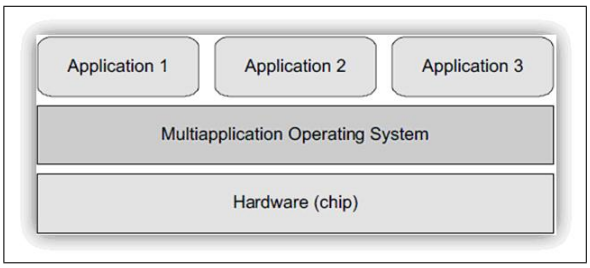

Figura A.5: Multiapplication card architecture

Java Card technology works via applets, which are applications written in Java language that you can download and run. Applets communicate among the off-card clients via the Application Protocol Data Unit (hereafter APDU).

Moreover, this new technology adds new security measures which do not exist before as on-card checker or a mechanism (firewall) which isolate the applications (data and keys) in their own context. Though the firewall isolates applications, it allows secure communication among applications through a public API.

As aforementioned, adding Java Card language to smart cards supply smart cards with new features as:

- Object-oriented programming
- Programming secure platform
- Operating System independent
- Multi-application support
- Secure applet loading
- Open standards

As well as another security features:

- Isolated applications through firewall
- Secure sharing and communication among applications
- Cryptographic support

A.3.3.2. General Features

- Java Card worked as a passive server until the new version 3.0 connected edition was developed. With this new version, applications can connect with off-card servers, losing the passive function. With this new version Java Card can also admit several communication protocols (both secure and no-secure).
- As aforementioned, Java Card uses APDU commands to communicate with the off-card system. The APDU [6] is a communication format based on question/response messages and

used between on-card and off-card applications (Figure A.6) following ISO-7816-3 ([1]) specification. APDU is defined as an entry point object in the Java Card, so every applet context can access to it. APDU commands were long restricted, so if the data sent is longer than this limited size, the information is written in portions and sent separately. After version 2.2.2 [16], the specification ISO 7816-4 defines an extended format, which is able to send until 65536 Bytes, but due to the use of shorts in the field which denotes the length of the messages LC and LE, the maximum data length to send is 32,768 [2].

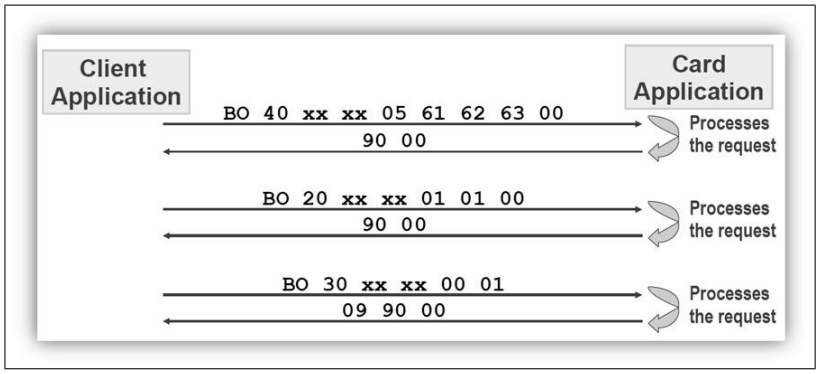

Figura A.6: APDU exchange

- Two heaps: volatile and no-volatile. All the objects in a Java card are created in the volatile memory. To make these objects persistent (be part of the no-volatile memory), they should be referenced by an object which is in the no-volatile memory. All the objects of the volatile memory are garbage collected (this function is automatic in Java Card 3.0.2 ([5] [3])). By way of illustration, the Figure A.7 shown the volatile and the non-volatile heaps. The non-volatile contains several objects and in the volatile is shown an object called "v"which point to two others: "s1."and "s2". In the next image (Figure A.8) "v"has been referenced by the blue object of the non-volatile memory. This means that both the object "v."and the objects referenced by has been promoted and stored in the non-volatile memory, while their old copies are garbage collected.

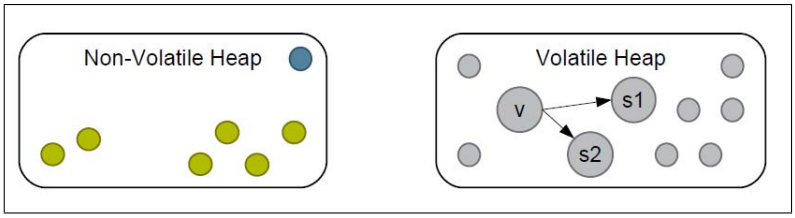

Figura A.7: Objects in volatile memory

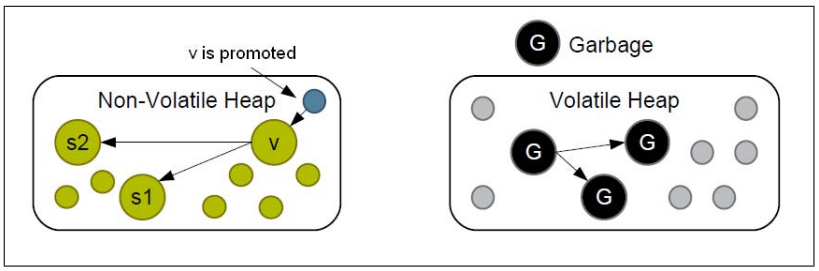

Figura A.8: Objects in non-volatil memory

- Firewall among applications: the firewall avoids applications to access to other applications objects unless these applications communicate through shared interfaces and they have access permission.
- Split Virtual Machine (VM): the VM (as it can be shown in the Figure A.9) is divided in two parts due to Java card technology features limitations. So, the off-card part (converter) receives the files .class, which contains the applications code, and does the following tasks: load, link and name resolution as well as bytecode verification, optimization and conversion, obtaining the .cap file, which is sent to the on-card part (interpreter). There, the interpreter manages the bytecode execution and the security enforcement.

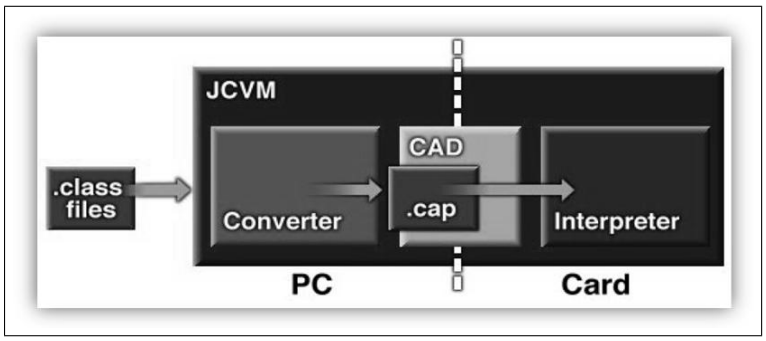

Figura A.9: Split virtual machine

- Transactions: owing to the card can be disconnected in any moment, Java card add the concept of transactions: all the operations done by an application should be atomic and consistence. If a transaction doesn't finish, all the things which happened will be ruled out to assure the integrity and consistency of both data and code [10].

A.3.3.3. New Features

With the new two versions (3.0 classic and connected edition) some of the old features have been upgraded and others have been added. The connected edition has the upgrades of the classic edition adding some more [3]:

- Applets:
  - A Java application can have several packages instead one.

  - More facilities and libraries
  - Concurrently execution over different I/O interfaces
  - Extended applets (similar to the old ones but using the new API features)
- Multithreading: With this new version, the web and the classic applets support multithreading. Since of the threads are no persistent, this editions add a new feature which saves the execution tasks to recover they automatically after a reset.
- Persistence: both machine code and applications are stored in the persistent memory, instead the objects should be referenced by a persistent object if you want to keep it. To make it and due to the objects are created automatically in the volatile memory, Java card use a strategy which consist on reference this object by a persistent object to be promoted. When an object in the no-volatile memory is no referenced anymore, this object and all the objects it references are garbage collected [10].
- Transactions
  - Multiple concurrent transactions
  - Nested transitions
  - Better control and programming of the transactions duration.
- Communication between applications: to improve and make easier the communication among applications the new version adds two new features:
  1. Publish a service that the other applications can use. They can be published by applications or be a predefined service.
  2. Informing other applications that something has happened through an event, achieving an asynchronous communication model. As with the services, predefined events are already defined.
- Network communication: with the new versions, applications are allowed to connect directly with off-card servers. This involve smart cards can work not only as passive servers. Besides, Java cards support several communication protocols both secure and no secure.
- File access: each application has its own storage space where the application is the only one which has access.

A.3.4. Applications Uses

Since the appearance of multi-application smart cards, multiples industries have developed application which could be in smart cards together with others. These industries can be classified into:

- Payment and banking
- Loyalty
- Insurance
- Governmental
- E-security

Apéndice B

Security of Smart Cards

In this chapter it is informally introduced the issue of security of smart cards, distinguishing between hardware and software security. This chapter describes both:

- How the attackers try to break in the smart card information and how it is prevented
- How the smart card prevents to be broken in and how the attackers try to skip these methods

B.1. Hardware Security

Smart card hardware security consists in avoiding visualizing how the smart card works and the data it has and controlling the environment and the inputs which could be dangerous. These hardware security issues try to prevent the attacker from obtain important information about how the card works or the data it has. There are different methods to protect smart cards depending on the attacks are going to be realized.

B.1.1. Anomaly Monitors

These monitors are the responsible for switch off the smart card when the environment conditions become extremes or the voltage or clock values become anomalous. There are different kinds of monitoring [19]:

- Voltage monitoring: when the voltage supplied to the card is lower or upper than an established limit, the smart card switch off automatically to prevent from possible attacks. Due to attackers try to deactivate the monitor before doing the attack, the monitor is especially protected.
- Frequency monitoring: similar to the voltage monitoring, if the value of the clock frequency supplied by the external entity is lower or upper than the limits established by the manufacturer, the monitor orders the card to switch off in order to avoid the microcontroller to be analyzed.
- Temperature monitoring

B.1.2. Reverse Engineering

Since the chip and its components could be observed in order to learn how the chip works and, with this information, try to find weaknesses on it, smart cards includes methods as putting the components randomly to make more difficult to understand how them works. The chips include also methods as obfuscated logic and buried buses to prevent reverse engineering.

B.1.3. Scrambling

This method consists in scrambling the buses and the EEPROM to avoid knowing the addresses of the memory or the function the bus realizes [19].

B.1.4. Ion Implanted ROM

The manufacturers use a kind of ROM where the data cannot be visualized using a microscope within visible or ultraviolet light spectrum [19].

B.1.5. Silicon Features

Adding metal shields prevent from exterior access to the components.

B.1.6. Side-channel Countermeasures

Side-channel attacks are a no-invasive attack which involve observing the supply consume, electromagnetic radiation, the time that need a task to execute and so on. The more normal methods to prevent from these attacks are:

- To execute always the instructions in the same order regardless of the data.
- To include random delays between instructions.
- To put shields to avoid the electromagnetic emissions.

B.1.7. Redundancy

This method consists in repeat parts of the algorithms which are too easy to attack in order to avoid an input specially manipulated to discover how the card works using fault analysis methods [18].

B.2. Software Security

Smart cards offer a lot of methods to protect both privacy and security of the data stored [9]. The methods are:

- Authentication: smart cards provide mechanisms to prove the identity of the user, the applications and the off-card entities.
- Secure data storage: the data stored in the card could only be accessed by the application which belongs to or by other applications through the API, but only if the application has the firewall authorization.
- Encryption: smart cards have several encryption methods to keep the system safely.
- Secure communication: smart cards have secure communication protocols which provide confidentiality, integrity and authentication.
- Biometrics: smart cards allow storing biometric templates to the systems which allow these methods.

- Certifications: smart cards have been certificated as carry out with the standards after strict tests.

Besides, contactless smart cards have more security features related to them [9]:

- Mutual authentication: not only the card should identify itself besides the reader should do before starting.
- Authenticated and authorized information access: the smart cards could recognize the authority of the requestor and it will give him only the requested information, not access for all the information stored.

In the other hand, smart cards data can be attacked using different methods [19]:

- Bug exploits: although in all the software are bugs, in the smart cards is very difficult to use it because they are very difficult to find and they should be loaded before exploit it. Usually, these bugs are used with physical attacks to be successful.
- Illegal bytecode: these attacks are very difficult to find in the smart cards because verify all the bytecode require a lot of CPU power which the cards don't have. If this attack reaches its objective, the attacker obtains the control of the card.

Besides these attacks, which affect to both contact and contactless smart cards, the contactless smart cards even have more attacks [19], which are:

- Altering data transmitted: an external attacker could do a man-in-the-middle attack, obtaining important information.
- Denial of service: an attacker could, with a strong radio tool, realize a lot of requests to the reader making impossible to the other cards to use it.
- Eavesdropping of the data transmitted: due to the information in a contactless communication goes through the air, this information could be stolen by a non-authorized user compromising the information sent.

B.2.1. Java Card Security

3.0 connected edition add new security mechanisms to the standards in smart cards [3]:

- Code isolation: the code of an application cannot interfere with other applications' code since each application has its own space referenced to the loaded application. Even so, it is allowed using public interfaces and shared libraries to communicate with other applications' objects.
- Package access control: prevents packages to be overwritten or extended once the package has been loaded.
- Context isolation: an object owned by an application in its own context cannot be acceded by applications in other contexts unless they access through the shareable interfaces. It is imposed by the firewall and it helps with application containment.

- Dedicated name-spaces: in this edition, each application has a unique name and it is represented by a URI, and its resources are referenced through this name. This mechanism helps with applications containment.
- Permission-based security: this powerful tool allows a security authority to avoid the access to resources, libraries and so on, depending on the characteristics the application which wants to access has. There are two different controls:
  1. Context-based: where an application in a context wants to access, e.g. to a shareable interface of another application in a different context.
  2. Programming permission check: the one which the system imposes initially to the system.
- Role-based security: adding a new characteristic to the applications (the role) it is possible to restrict the access to resources and applications according to the role value of the applications.
- User authentication and authorization: the authentication consists on prove that the user is who he says through a password, a PIN or other access mechanism. Once the user is authenticated, it is necessary to know if the user has authorization to carry out with the task he wants to do.
- On-card client application authentication and authorization: in this case, a client application should authenticate to the server application. With this identity, the server allows or denies the client accessing to protected resources.
- Network communication security: it consists in reach a secure communication among applications and off-card peers through secure communication protocols as Secure Sockets Layer (SSL), Hypertext Transfer Protocol Secure (HTTPS) and so on and cryptographic algorithms to reach confidentiality, integrity and peers authentication.
- Key and trust management: it is the responsible of managing the keys used to authentication tasks and trust content in the secure connections when the application wants to open a secure channel but it does not want to be worried about security.

Although smart cards provides several security methods and moreover Java Cards add more methods to increase security on communication, context and code isolation and so on, nowadays multi-application smart cards are rarely used unless all the applications come from the same provider because no methods which assure secure data sharing among the applications installed are implemented.

Apéndice C

Supporting Applications' Evolution in Multi-application Smart Cards

The aim of this chapter is to examine the problems on multi-application smart card when the security features of both smart cards and Java Cards are not enough to integrate them in a real system where several applications of different stakeholders which are placed in the card interact and share information.

The chapter begins explaining the current multi-application systems and how have them migrated to smart cards. It will then go on to explain the current multi-application smart cards, why they are not real multi-application and why it is necessary to develop a new framework which solve these problems. Finally the state of the art is presented.

C.1. Multi-application Interaction

Nowadays a lot of new environments where the applications cooperate and share information have arisen, as well as more applications could be installed or modified adding or removing some functionalities during the live of the system. That is possible since in these environments either the security of the data stored and shared is not important or the computational power is enough to use a powerful tools as runtime-monitors which control the security of the data stored and shared.

The problem starts when the data is the most important part of the environment and the system does not have enough computational power to use one of the tools which control the data security in the runtime so its security cannot be assured.

This is the problem of smart cards, the most widespread computers which nowadays are used like transport cards, banking-transaction cards and so on. Due to every person has more than one of these cards in his pocket, the solution goes through make an unique smart card with all the functionalities the person needs. Because of this solution was noticed long time ago and the solutions taken to solve this problem in the others environments were not possible due to the computational power limitations, some standards were developed to solve it in multi-applications smart cards. The most commons standards are Java Card, MULTOS and GlobalPlatform.

Although this standards were developed to cooperate among different stakeholders in the same smart card, the majority of the smart cards used as multi-application smart cards are because the applets come from the same provider. But the problem continues being the information sharing. The smart card platform assigns a separate security domain for each application provider and also provides secure methods to share information among the different applets installed in the platform. In the case of Java Card, which is the solution which has been chosen to develop this thesis since it is the most widespread, are the Shareable Interfaces.

Imagine a current multi-application smart card which has two applets, one for a transport company and other from some bank the owner uses. It is normal to think that each time the owner uses the card to pay the transport ticket it is made through a service of the bank has shared with the transport. But, in fact, nowadays this exchange of information has done in a system outside the card. This happens because the methods to share information are not secure or is not possible to trust in the others applications installed on the card.

But this becomes worse when the applications can be modified as in Java Card happens. The multi-applications cards are certified by the manufacturers with the card platform and the applications the providers included in the card. So, if an application is modified for an update, the certificate is broken and it is necessary to certificate again, which relays in expensive costs.

When the multi-application smart card has applets from different providers it becomes much more complicated. When one of the providers of the applets wants to modify an applet, he should negotiate every time among the others stakeholders since the protocol Java Card uses to share information. Although it is true that Java provides secure ways to share some information, these methods have some problems and limitations as have been explained in [8]. Imagine that two applications A and B are installed on the card and share information between them through the firewall using the Shareable interfaces. The application B wants to uses the service s from A and, to do that, it uses the shareable interfaces. The firewall detects the request and asks A if the application B is allowed to use its service. A has a list where the firewall checks if B is allowed to use it service comparing the AID (Application Identifier) of the applet with the allowed applets. After check it the request is sent. This verification is checked every time an applet wants to use a service provided by other application. The problem is that, if an applet is updated, the applets which provide services it uses will not be sure that the updated applet is still secure or some malicious application has impersonated the trusted one. Moreover, using this method to manage the data sharing, when a server application allows a client application to share information, the server allows to use all the shared interfaces it has, not only the interfaces the client needs, compromising the security of the card. So, when a provider modifies an application, he should paid the new certificate in order to continues using the agreement with the others providers.

So, the solution could be to add a certified repository where the applications store the information they share and the applications which can use it. Using this, when an application is modified, only this repository should be re-certified, making easier to change the access policies.

From this idea, the Security-by-Contract (hereafter SxC) approach taken from the initial idea for mobile devices is proposed for smart cards. This model is based in the Model-carrying code proposed by Sekar et al [**?**] and development-by-contract. In this approach, the applications arrive to the platform with a Contract, which is a description of the behavior of the application. This contract is compared with the platform policy, which is the union of the other installed applications contract and checked for compliance. If both are compliance, the installation or de update is accepted. Otherwise the user will be notified with the problem and asked to solve it.

To introduce this approach in a smart card, SxC should be adjusted to both the computational power and the memory space the card has.

C.1.1. Example

You start imaging a Java Card with a server application installed on it. If this servlet wants to share information with other applets using sharing interface (the method implemented in Java Cards to share information among applets), the applet should know a priory the AIDs of this

applications. This is a challenging requirement to a server applet, but, though it will know the AIDs of all the applets is going to interact with, what will happen with the applets written after and which needs access to this servlet? This involve that either these objects cannot obtain access to this servlet or it is necessary to rewrite the servlet with the new AIDs until the next time it will happen again, which will entail in a huge expense [23].

Now, suppose the previous problem has been solved and the servlet has all the AIDs of the applets allowed to use its services. The problem now is that some application could impersonate a legitime application because only the AID is checked. A malicious application could change its own AID and set it with the AID of a well-known applet. To solve that, the servlet and the client should have a secret shared between them and pass it as a parameter in the function "getAppletShareableInterfaceObject"[12].

By way of illustration, we suppose an server applet installed in a Java Card. This application provides several services for different client application. Not all the services are provided for all the applications allowed to use some of them, but the shareable interfaces do not difference from services. If the servlet wants to difference among the applications allowed it should implement one interface for each application or group of applications which use the same services. Later, with the function "getAppletShareableInterfaceObject", the client application will select which of the shareable interfaces it wants. But once the client gets access to one interface, the client could access to services of other interfaces ([23] and [8]).

C.2. State-of-the-Art

In [13] Ghindici et al. proposed a domain specific language for security policies describing the allowed information flow inside the card. Each class of an application is certified at loading time, having an information flow signature assigned to each method. Information flow in this framework is represented as a relation between two method variables with an annotation about the type of flow, for example, from secret to secret through a direct assignment.

In [21], [11] Huisman et al. presented a formal framework and a tool set for compositional verification of application interactions on a multi-application smart card. Their method is based on construction of maximal applets, w.r.t structural safety properties, simulating all the applets respecting these properties. To check that the composition of two applets A and B respects the behavioral safety property model checking techniques can be used.

In [17] Domingo i Ferrer proposed an approach to increment the capacity of the smart cards exporting some complex and heavy operations outside the card, when storage capacity or the computational power of the smart card is not enough. To do this, he proposed to use privacy homomorphisms. In this method, the card is the responsible of encrypt the private data stored on it. It sends this information to the off-card system, which will make the pertinent operations and the encrypted results will be returned to the card. These results are decrypted obtaining the clear data which will be used after.

In [15] Girard suggested to associate security levels (clearances) to application attributes and methods, using traditional Bell/La Padula model, and the security policies in this model define authorized flows between levels. This approach was further explored in [25] by Bieber et al. and the technique based on model checking for verification of actual information flows was presented there. Schellhorn et al. in [14] used the same approach for their formal security model for operating systems of multi-application smart cards. Avvenuti et al. in [20] proposed a tool for off-card verification of CAP files (Java Card applications are delivered on the platform as CAP files), that can be later

installed on the platform. The method again explored the multi-level security policy model and the theory of abstract interpretation of the operational semantics.

The SxC approach improves the current literature by addressing problems related to the dynamic evolution of the applets on the platform. Moreover, it is based on a hierarchy of models that allows having some benefits in terms of computational complexity or language expressivity. In particular, the first level of the hierarchy (presented in this thesis) proposes on card algorithms not more complicated than the usual smart card applications, while the mentioned approaches are usually based on complicated logic and model checking, all needing off-card activities.

Apéndice D

Security by Contract

As has been explained in the previous chapter, there are several ways to implement a protocol to interchange information among the applications installed in the smart card, but each solution has some problems to assure the security of the data shared. But the most important problem is that with these protocols is not possible to download new applications, delete old applications, update applications or change their policy (the applications which are allowed to share information with) without relay in security failures, compromising the well-working of the applications in the card or relaying in several costs. In the next sections of this chapter the SxC framework and the different options which are possible to develop will be explained and detailed, and also the option which has been selected.

D.1. SxC Framework

As in the previous chapter was mentioned, the SxC framework is based on the Model-Carrying-Code idea which is actually working for mobile systems and relays in extract the contract from the applications code and compare with the policy. Once both are compatibles, the Model-carrying Code assures that the policy will not be violated by the code where the contract has been extracted [**?**]. As in this protocol, in the SxC applications come with a contract which describes the security behavior.

Definition (Security Contract) a security contract is a formal specification of the behavior of an application for what concerns relevant security actions.

The example shown below helps to understand what a contract is:

Example 4.1: Suppose a company called Pharmacy which has an application Medicine@Pharmacy which establishes the next contract: "The application provides a global service called list_medicines.Medicine@Ph which could interact with the applications of the company Pharmacy and Hospital".

This contract could be extracted directly from the application bytecode. From this code, this information could be extracted in a first approximation [24]:

- Shareables Interfaces
- Needed calls to other security domains
- Allowed calls by other security domains
- Forbidden calls by other security domains

But after, in a second approximation [22] the information extracted consists in a set of services classified as:

- Services the application provides
- Services the application calls
- Services the application needs
- Services the application allows and the application which is allowed to use it

Due to this modification, each application should explicitly describe which application is allowed to use a service, not only the security domain.

Definition(Security Policy) a security policy is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions.

Continuing with the previous examples, a smart card policy example could be something like:

Example 4.2: "The service list_medicines.Medicine@Pharmacy of the application Medicine@Pharmacy is allowed to be used only by other applications of Pharmacy and Hospital companies".

Knowing how a contract and a policy are defined, now the contract-policy matching could be defined as:

Definition (Contract-Policy Matching) a contract of an application A matches a platform policy if there is no illegal information exchange between the application A and the applications already on the card.

Due to this definition, a contract matches with a policy if does not exist an information leak between this contract and the policy (which is the contracts already added in the platform) following the workflow shown in (Figure 2.1). So, when a new application tries to be installed on the platform, it is only necessary to check that forbidden communication will not exist. After this is checked, if the policy and the contract are compliant (no forbidden communication exists) the action supposed to be done (i.e. update of an application) will be executed, otherwise the action will be rejected. Continuing with the examples:

Example 4.3: (Successful match) Suppose the application Pharmacy is already installed on the card and the policy has been updated with its information. Now, a new application Manage@Hospital from the company Hospital which uses the service list_medicines.Medicine@Pharmacy tries installing itself. Due to the policy has stored the contract of Medicine@Pharmacy and its contract allows the Hospital applications to use its services, hence no information leak exists and the application installation is accepted.

Example 4.4: (Unsuccessful match) On the other hand, if another company (not Hospital neither Pharmacy) want to use the service list_medicines.Medicine @Pharmacy, the platform will reject the installation caused by the illegal information exchange this new application wants.

Once the policies and the contract has been explained, it is necessary to look for a way to develop this in a smart card where the computational power is not too much. Besides, process this information outside the card required secure communication methods and cryptography to make sure that the result has not been altered. In order to solve this limitations, in [24] a hieratical model has been presented where the higher level should be more expressive but more costly, and in the lower levels the opposite.

D.1.1. A Hierarchy of Contract/Policy Models

The first challenge we have to address is to find an appropriate language for specifying contracts (policies) describing possible (allowed) information exchange among applications. To address computational limitations we propose a hierarchy of contracts/policies models for Global Platform-based smart cards. The rationale is that each level of the hierarchy can be used to specify contracts and policies, but with different computational efforts and expressivity limitations. For instance, using the lower level (L0) one can get computational benefits, but lose in contract/policy expressivity.

- L0: Application as Services. This level models applications as a list of required and available services. Essentially it is the current set-up of the GP.
- L1: Allowed Control Flow. This level provides a call graph G1(A) of the application, where vertices are the states of the application and edges represent the invocation of different services. Then we can do a bit of history based access control and more fine grained information exchange control.
- L2: Allowed and Desired Control Flow. This level adds to the previous one the notions of correct and error states. It can be necessary if we want to test that the removal of an application (or a change in a policy) does not break other applications.
- L3: Full Information Flow. This level extends the previous one considering also the information flow among variables. Practically speaking, moving from some level to a higher level (for instance, from L0 to L1) means to add more details in the contract/policy specifications, modeling more precisely the behavior of the applications running on the card.

In this thesis, the level 0 is going to be used to implement the SxC framework.

D.2. Supporting Applications' Evolution by SxC

In this section, the information extracted from the code and the platform policy, which should be represented in the same way to make possible to compare it and check if they are compliant, is explained. Also the algorithms and why they are used will be exposed.

D.2.1. Information Structure

First of all, to address the problem is necessary to specify the changes in the contract of the applications which are going to be dangerous to keep the platform in a secure state. These changes could be:

- Add a new application to the card
- Remove an old application from the card
- Modify an application already on the card adding or removing services

To assure the secure state two properties have been defined in [22]. These properties describe how should be the state of the platform after occurs a change in order to keep the secure state of the platform. These are:

- P1 (Stable Security) After a change there should not be illegal information exchange between applications

- P2 (Stable Functionality) After a change every application on the platform should be able to work correctly

In order to keep these statements, the code is checked before an application modification. If both are compliant (the new application characteristics and the platform policy), then the change is allowed and the platform policy is updated with the new changes. Otherwise, the change is rejected. In both cases, the properties will be still kept.

To achieve this problem, the representation of both the contract and the policy is made listing the services which the applications provides, the services which the application needs, the services the application calls and which of this services are allowed to others applications, in sets. These sets are directly extracted from the application using either proof-carrying-code techniques or other technique which allows extracting the contract directly from the application code as has been mentioned in [22]. With these techniques is possible to extract the application contract which has been divided in two parts, the claim and the application policy. This information is stored in the card to use it afterwards.

The information extracted from the code has been divided in two parts because the four sets can be classified by two different criterions. The first one is called claim and is described as how an application is supposed to behave with the applications which are or will be on the smart card. On the other hand, the application policy is referred to how others applications, which are supposed to share services, should interact with it. The application providers, the security domain owner or the controlling authority will establish this behavior (see [24]).

For each application on the card, the Claim is a pair (Calls, Provides). Both are subsets of services the applications which are or will be installed on the platform have. The Calls set represents the services this application code is going to call during its execution on the card. On the other hand, Provides is a subset of services this application has implemented and supplies to the platform so other applications can use.

The application policy (hereafter AppPolicy) is also defined by a pair (Allows, Needs). In this case the Needs is a subset of the services which are called by this application and are included in its Calls set. They will be necessaries during the invocation of the application. In the other hand, the Allows is a set of pairs, where one of it has a service of the Provides set and the other element could be an installed or a non installed application ID.

With this new classification the previous contract example will be represented as:

Application	Claim		Policy	
	Provides	Calls	Needs	Allows
Medicine@Pharmacy	list_medicines	-	-	(list_medicines, app1@Pharmacy) (list_medicines, app2@Hospital)

Tabla D.1: Contract example

Once this has been explained, the information exchange can be defined using these concrete sets of services. A information exchange between two applications A and B exist when the service "serviceAïs provided by A and is invoked by B. That means that in the Provides set of the application A exists an entry called "serviceA.ªnd in the Calls set of the application B exists an entry called "serviceA". But, though an information exchange between the two applications exists, this does not mean that this exchange is legal. A legal information exchange is defined as:

Definition(Legal Information Exchange) An information exchange is legal only if the application which provides the service allows the applications which calls the service to use it.

Following with the examples, the information exchange between application A and B is legal if the pair (serviceA, B) exists in the Allows set of the application A. Besides this, it is also necessary to satisfy the application policy. To achieve this, all the services in the Needs list of an application should be provided for other applications in the platform.

With this new data structure, the changes in the contract can be explained more in detail. These changes can also be separated in two different types depending in the problem they cause: functional or security. The functional is referred to the AppPolicy and the security failures to the Claim changes. The possible changes and the possible failures, which have been explained in [22], are:

- Changes in the Claim:
  - Add a service to Calls set could provoke a security failure due to this application is not allowed to use it.
  - Remove a service from Calls set cannot provoke any security failure.
  - Add a service to Provides set can provoke a security failure because this service could be called by an application already installed in the platform but not allowed to use it.
  - Remove a service from Provides set could provoke a security failure because it is possible that some application needs this provided service.
- Changes in the AppPolicy:
  - Add a service to Needs set can provoke a functionality failure because it is possible that the service requested is not provided.
  - Remove a service from Needs set cannot provoke any functionality failure.
  - Add a pair (service, application) to Allows set cannot provoke any functionality failure.
  - Remove a pair (service, application) from Allows set can provoke a functionality failure because the application in the pair could be still using the service allowed.

In all these cases, the change could be accepted or rejected. The changes which cannot lead in any security or failure will be always accepted. But in the case the changes can lead in any failure, it is necessary to check the smart card issuer and the stakeholder to solve it. In this thesis, updates which can lead in security or functionality failures are always rejected.

Finally the policy of the platform should be defined. Because the policy should satisfy all the requirements of the stakeholders, it is compositional and provided by them, and when a new stakeholder arrives to the car, her policy is checked to be compliance with the policy already in the card and. If this new policy is compliance, her new rules are added to the card policy. Therefore, the policy is defined as a set of the AppPolicy from all the applications stored in the platform. This is necessary because some authorities want to choose how the other applications on the card can interact with its application's services. Due to that, some external rules control the interaction among the applications in the platform. Which is more, this policy could be updated without modify the working application. This has been defined before as AppPolicy, where each application authority set the relations allowed. The combination of all these application policies makes the platform security policy. Due to that, if a platform has "n.ªpplications, the Policy will be= $\{AppPolicy_1, ..., AppPolicy_n\}$. An example with three applications could be as shown below:

Application	Policy	
	Needs	Allows
Medicine@Pharmacy	-	(list_medicines, app1@Pharmacy) (list_medicines, app2@Hospital)
app1@Pharmacy	-	-
app2@Hospital	-	-

Tabla D.2: Policy example

But this information can also be reordered in two set of relations, PolNeeds and PolAllows, depending on the target of the security requirements. The PolNeeds set is represented as $\{Dependentes_1,..., Dependents_n\}$ where each element Dependents is a list of the applications which needs some of the services $Application_1$ provides. For the PolAllows set, it is represented as $\{Allows_1,..., Allows_n\}$ where each element is the Allows set of the corresponding application.

Finally, the applications installed in the platform will be stored as the sum of the applications' contracts and it will be represented by Λ.

D.2.2. Algorithms for Applications' Evolution

Once the problem has been exposed, the algorithm necessaries to address the problem will be explained. These algorithms could be divided in two parts: one with the algorithms which check the compliance of the actual policy with the changes are try to do and the algorithms which update the policy if the change is compliance with the actual policy.

In the first group, the algorithms are:

- Check of compliance of new application
- Check of removal of application
- Check of compliance of AppPolicy update
- Check of new Contract compliance with the Policy

For the second group are:

- Policy update for approved new application
- Policy update for approved application removal
- Update of the Policy after approved AppPolicy change

D.2.2.1. Check of Compliance of New Application

When a new application wants to be installed in the smart card it is necessary to check if the application contract is compliance with the actual policy and the others applications contracts installed on the platform. To make sure this contract is compliance, the algorithm check three different parts of the Contract which may lead in any failure.

The first part checks, for each service the new application has in its Calls set, if there is an application on the platform which provides this service and if this application allows new applications

to use it. If an application which provides this service doesn't exist, this application cannot lead in a security failure. If some application provides this service but it does not allow to the incoming one to use it, then the new application is not compliant with the platform security policy and the installation is canceled. Otherwise, if the service is provided and the application which provides this service also allows the incoming application to use it, this step is passed and the algorithm continues checking the compliance.

In the second part the Needs set is checked. In this case is enough checking if all the services the application needs are provided for one of the application on the platform. If they are not, the installation will be canceled.

Finally, in the third case, the services which the application provides are checked. If the services provided are not called for any application on the platform, the application will be installed. On the other hand, if the services are called by some application on the platform, these applications should be also allowed to use it, otherwise the installation will be rejected.

If the application contract passes all these three steps, then the algorithm will return true and the application will be installed on the platform.

D.2.2.2. Check of Removal of Application

This algorithm is the responsible checking if no applications relays in the application which is wanted to be removed. To achieve that it is only necessary to check the Dependents list of this application. If the Dependents list is empty because none applications needs its services, the application will be removed. Otherwise, the removal will be canceled.

D.2.2.3. Check of Compliance of AppPolicy Update

When any provider wants to modify its AppPolicy adding or removing some services, first it is necessary to check the compliance of the new AppPolicy with the platform security policy. To make sure of this it is necessary to check some special cases as it is shown in [22]:

The first case deals with when a new service is added to the Needs list. This should be checked because this new service should be supplied by an application in the platform. To achieve this, first it is necessary to store the new services added to the Needs list and after, make sure that all of them are provided by the platform applications.

Secondly, when a service is removed from the Allows list. Because maybe some application calls this service, this services which will be removed from the Allows list are stored in a new list and this list is checked. If none of the services in this list are called by any application on the platform, this step will be passed. Otherwise the update will be rejected.

Finally, if the previous steps have been passed, the algorithm returns true and the update could be applied.

D.2.2.4. Check of New Contract Compliance with the Policy

Instead of modify only the AppPolicy, all the contract could be modified, so it is necessary a new algorithm which checks this modification. So, in this algorithm is not only necessary to check if the AppPolicy is correct as in the previous algorithm, but also if the Claim of the contract is. Because of this, there are some parts of the algorithm (the parts which check the Needs and the Allows differences) which will be similar to the previous one (Algorithms 3 parts 1 and 2) and which are shown below.

In the different ones, first is checked the Calls modifications. For each new service added to this list, it is necessary to confirm if the service is supplied for an application in the platform. In the case the service will be supplied, it is also necessary to check if the application is allowed to use it.

Finally, for the Provides differences it is necessary to check both the added ones and the removed ones (Algorithm 4 part 3). In the first case (services removed from the list), the algorithm checks if the services which are going to be removed are needed for some application installed on the platform. If some service is still needed, the update will be rejected. Otherwise this step will be passed and the algorithm will continue with the checking. In the second case the services added to the Provide list are checked. Only if one of these services is called by some application in the platform but this application does not allow the callee to use it, the update will be rejected.

D.2.2.5. Policy Update for Approved New Application

Once the application compliance has been accepted, it is necessary to update the platform policy and to add the new application contract. To do that, the algorithm developed has three distinguished parts; the first one which modifies the data stored in the old policy, the second one which adds the new application contract to the application list and the third which modifies the policy adding the Dependents and Allows of the new application.

First of all, the services needed by the new application are checked. Each element in the list is compared with the services provided in the platform. If this service is provided then the ID of the incoming application is added to the Dependents list of the application which supplies the service.

Next, an empty vector is added to the PolNeeds list with the ID of the new application and the Allows list of the new application is stored in the policy list PolAllows.

Finally, the application list is updated with the contract of the new application.

D.2.2.6. Policy Update for Approved Application Removal

Once the application removal compliance has confirmed, the application can be removed from the platform. To remove it from the platform will be necessary to update the policy of the platform and remove the application contract.

As can be seen from the Algorithm 6, the Dependents and the Allows lists are removed from PolNeeds and PolAllows respectively. After that, any reference to the application is removed from the Dependents list and later, the application is removed from the applications list.

D.2.2.7. Update of the Policy after Approved AppPolicy Change

After the compliance of the modification of an application (either the application policy or the application contract) is accepted, the platform policy should be updated. Since policy only depends in the Needs and the Allows information, the algorithm only needs the new and the old AppPolicy of the modified application.

Like in the algorithm which checks the compliance of a new contract, first the differences between the new and the old Needs list are extracted storing the new services added in a list. For each element in this list, the services provided by the applications on the platform are compared to them. If the service is provided, the ID of the application which is being modified is added to the Dependents list of the application which provides it.

After that, the services which have been removed from the application Needs list are stored. In

this case, if some application has been providing this service, the ID of the application which needs this service is removed from the Dependents list of the application which provides the service.

Finally it is necessary to store the new Allows list, so if the new and the old Allows list are different, the new one is stored in the policy.

Apéndice E

Implementation of SxC Level 0

This chapter will examine:

- The information about the technology used to develop the program which solves the problem
- How this program has been implemented which includes searching of information not only about the technology used either the way this information should be used to solve the problem.
- The problems which arose during the implementation of the system with both the technical and the theoretical problems.
- Also the implementation of the system will be explained in detail emphasizing the most important points as the algorithms which represent the framework itself. Also the test did to check the correction of the implementation will be explained.
- The last section assesses memory footprints, which measure the memory the system uses and references during the execution and when the card is switched off.

E.1. Technology adopted

To address the problem presented, Java Card Technology has been selected due to the fact it is the most important standard already in use and also the easiest, common and documented one. Nowadays, the most common and moderns version developed for Java Card are: Java Card v2.2.2 and Java Card v3.0.1 (connected and classis edition). Since it is true that the first version is the most used in the current systems, the new one (version 3.0.1) adds more features and it will be the one which will be used in a close future. Although it was possible to develop the system in both versions, the second one was chosen because it makes easier the development of the system with more modern facilities and libraries.

E.1.1. Installation of the Environment

For the installation of the system, the Development Kit User's Guide [4] has been followed. As a summary, the steps necessaries to follow are, in order:

1. First it is necessary to download and install several programs before install the development kit. Each program should be installed following the instructions available in the his web page. The programs are:
    - Apache ANT, necessary to run the samples.

- Firefox browser, necessary for running the reference implementation (RI) which simulates a dual concurrent interface implementation.
- Internet Explorer 7 browser to be used as a remote client.
- GCC compiler with MinWG required to build the cjcre.exe.
- Java Development Kit version 6 update 10 or higher.
- NeatBeans IDE 6.8 including the Java Card platform plugins to develop and run examples.

2. Secondly, install and set up the Development kit. Download the appropriate development kit from the Sun web page and install it double clicking in the JAR file or executing it from the command line. The installation wizard will display a window which will guide you during the installation.

3. After that, the System Variables should be set up. The easier way is setting it permanently, but if due to some other programs requirements it cannot be done, it is likewise possible to set up only temporarily. If you want to keep it permanently, you have to go to Windows Control Panel > System > Advanced > Environment Variables dialog and create variables for Java, Java Development Kit, MinGW and ANT with the value of the directory where the programs have been installed. Once the variables have been created, they should be added to the Path variable of the system in the same window.

4. Once the variables have been added to the path, it is necessary to configure NetBeans. Initialize it and upgrade to the last version. Next, go to the plugins list and add the ones related with Java Card. After that, go to Tools > Java Platforms and add a new platform. Chose the Java Card Platform, select the directory where the Java Card Development Kit has been installed and click on finish button. After this, restart NetBeans and start using it.

E.2. Implementation

This section has been divided in four parts. The first deals with the data structures selected to be used in the implementation of the system, why they have been chosen in this way not in other. The second part explains in detail the implementation of the algorithms discussed in the chapter four. The next one explains the complete system developed. The last part evaluates the size of each data structure presented in the first part of this section. A real example is finally presented and evaluated.

E.2.1. Data Structures

This first section describe the design of the three main data structures implemented.

First of all, it was decided that the best method to implement the sets defined in the theoretical part was using vectors. Vectors were selected because Java Card v3.0 was just added it in this version and secondly because vector class allows to implement nested structures whereas buffers do not, and this is an important part of the data structures implementation. Also vector class allows the user to create very short sets and after increasing automatically their capacity when the vector initial size becomes insufficient to accommodate new elements. This increase can be set by the programmer according to the program needs.

For this system, all the vectors were initialized with the initial size value equal to one and the capacity increment equals to one too. These values were selected to decrease the initial size of the

classes. Depending on the use the smart card is going to have or the applications which are going to be installed, the values of the different classes can be modified to save both computational power and size.

Lastly, three possibilities were considered to save the information about services, application IDs and so on: buffers, strings and stringbuffers. First of all, between strings and stringbuffers the first option was selected because the stringbuffers are implemented as mutable sequence of characters in order to use it when changes in the strings could happen, but for this case they were only necessary to store permanent data. In the other hand, strings are similar to buffers in the way they store the information, but buffers offer some disadvantages. First, buffers need to set the initial size value before known the data is going to store while string size can be set in runtime. Also strings offer several methods which helps in the comparison and in the search of similarities.

E.2.1.1. Contract

The design of the Contract (Listing E.1) was based in the theoretical part where, as it was explained, contract is divided in two parts: the Claim and the AppPolicy. Due to that, the application contract has been developed as a register with two fields, one of them with the Claim of the application and the other with the AppPolicy. An extra field with the application ID has been added in order to identify the application once the contract has been extracted from the application code and it has been stored into the card for future uses. The Contract constructor initializes the ID with the application ID and fills the Claim and AppPolicy with the input data received.

```
static public class Contract{

    public String ID;
    public Claim claim;
    public AppPolicy app_policy;

    /*****Contract Constructor****/
    public Contract(String ID, Claim claim, AppPolicy app_policy){
        this.ID = ID;
        this.claim = new Claim(claim.calls, claim.provides);
        this.app_policy = new AppPolicy(ID, app_policy.allows,
            app_policy.needs);
    }
}
```

Listing E.1: Contract Data Structure

For each application on the card, the Claim is defined as a pair (Calls, Provides). Both are subsets of applications' services which are present on the platform. The Calls set represents the services this application code is going to call during its execution on the card. On the other hand, Provides is a subset of services this application has implemented and supplies to the platform so other applications could use. This data structure (Listing E.2) has been developed as a register which has also two fields, one for the Calls set and other for the Provides set. Each field has been implemented as a vector which stores the applications IDs the application calls and provides respectively. This data structure has two constructors, one which creates the instances of the structure when no data is available and a second one which fills the calls and provides vectors with the input data as it receives.

```
static public class Claim{

    public Vector<String> calls = new Vector(1,1);
```

```
    public Vector<String> provides = new Vector(1,1);

    /*****Claim Constructor 1*****/
    public Claim(){
    }

    /*****Claim Constructor 2*****/
    public Claim(Vector<String> calls, Vector<String> provides){
        for(int i = 0; i < calls.size(); i++){
            this.calls.addElement(calls.elementAt(i));
        }
        for(int i = 0; i < provides.size(); i++){
            this.provides.addElement(provides.elementAt(i));
        }
    }
}
```

Listing E.2: Claim Data Structure

The application policy AppPolicy is also defined by a pair (Allows, Needs). Needs is a subset of the services which are called by the application and they should be included in it Calls set. These services will be necessaries during the invocation of the application. In the other hand, Allows (Listing E.3) is a set of pairs where each element has a service of the Provides set and the other element could be an installed or a non installed application ID. Each pair represents a service the application provides and the application which can use it. The Allows constructor receives the service and the application ID and copies these values into its own variables.

```
static public class Allows{

    public String service;
    public String application_id;

    /*******Allows Constructor********/
    public Allows(String service, String ID){
        this.service = service;
        this.application_id = ID;
    }
}
```

Listing E.3: Allows Data Structure

Similar as the Claim structure, the AppPolicy data structure (Listing E.4) has been developed as a register with two vectors where each one stores the Allows and Needs information. The Needs set is a vector which stores the services the applications needs like in the Claim structure. Contrary to that, the Allows vector stores in each field both the service provided and the application which is allowed to use it. This register has also a field with the application ID of the application which represents because, sometimes, only the AppPolicy of the application is modified, contrary to the Claim which is modified when the entire contract will be modified. Due to that, it is necessary to store this information allowing the algorithms to know which application policy has been modified. The constructor of this structure copies the input information in the two local vectors.

```
static public class AppPolicy{

    public String ID;
    public Vector<Allows> allows = new Vector(1,1);
    public Vector<String> needs = new Vector(1,1);
```

```
    /*****AppPolicy Constructor****/
    public AppPolicy(String ID, Vector<Allows> allows,
                     Vector<String> needs){
        this.ID = ID;
        for(int i = 0; i < needs.size(); i++)
            this.needs.addElement(needs.elementAt(i));
        for(int i = 0; i < allows.size(); i++){
            this.allows.addElement(allows.elementAt(i));
        }
    }
}
```

Listing E.4: AppPolicy Data Structure

E.2.1.2. Policy

As it has been explained in the previous chapter, the policy represented as a set of all the applications policies on the platform is modified and restructured in groups depending on the security requirements: (PolNeeds, PolAllows). PolNeeds stores for each application the applications which need some service of it. Each field of this vector has been implemented (Listing E.5) as a register with one string which stores the dependent application ID and a vector with the applications which need some of the services offered. An example of the vector could be: PolNeeds = ($Dependents_{A1}$,..., $Dependents_{AN}$), where the $Dependents_{A1}$ is a register with application ID = $Application_{A1}$ and a vector = ($Application_{A2}$,..., $Application_{AM}$), where all this applications need at least one of the services of the application $Application_{A1}$ and being $m < n$ & $n = applications\ on\ the\ platform$. On the other hand, PolAllows (Listing E.6) stores for each application the Allows list of this application and has been implemented in the same way than PolNeeds but in this case the vector is an Allows vector, which stores also the application ID and a vector with the list of Allows (Application ID, service) of this application. Each structure has a constructor which receives a vector (a vector string for the Dependents and an Allows one for the polAllows) with the information, and fills its own vector with this information.

```
static public class Dependents{

        public String ID;
        public Vector<String> app = new Vector(1,1);

        /*****Dependents Constructor****/
        public Dependents(String ID, Vector<String> app){
            this.ID = ID;
            for(int i = 0; i < app.size(); i++){
                this.app.addElement(app.elementAt(i));
            }
        }
    }
}
```

Listing E.5: Dependents Data Structure

```
static public class polAllows{

        public String ID;
        public Vector<Allows> app = new Vector(1,1);
```

```
        /*****Dependents Constructor****/
        public polAllows(String ID, Vector<Allows> allows) {
            this.ID = ID;
            for(int i = 0; i < allows.size(); i++){
                this.app.addElement(new Allows(allows.elementAt(i).
                    service, allows.elementAt(i).application_id));
            }
        }
    }
```

Listing E.6: PolAllows Data Structure

Finally, Policy (Listing E.7) is a register which stores the PolNeeds and PolAllows information with a constructor which initialize the variables.

```
static public class Policy{

        public Vector<Dependents> polNeeds = new Vector(1,1);
        public Vector<polAllows> polAllows = new Vector(1,1);

        /*****Policy Constructor 1****/
        public Policy(){

        }
    }
```

Listing E.7: Policy Data Structure

E.2.1.3. Applications Contracts

Because it is necessary to store all the information about the applications (both the applications ID and the contract) other data structure has been created to keep this info. To achieve this, at the beginning of the Multi-application class, a vector which will stored the contracts is created. In this case, the vector could be initialized with an initial capacity similar to the number of applications will be in the card.

```
private Vector<Contract> applications = new Vector(1,n);
```

Listing E.8: Applications Contracts

E.2.2. Implementation of the Algorithms

In this subsection how the algorithms have been developed and why have developed in this way will be explained. Each algorithm has two implementations, the first which is the first version and only makes the system works, and the second version, which optimizes the first version using vector functions, which is explained.

E.2.2.1. Check of Compliance of New Application

As has been explained in the description of the algorithm, it has been divided in three different parts where each one checks a different part of the application contract.

At the beginning, a boolean variable called .[ex]istsïs declared to use it lately.

In the first part the calls set is checked. Due to that, a loop which goes down the Calls vector is implemented. For each service in this vector, .[ex]istsïs set to false at the beginning of the iteration.

Now, the platform application list is gone through. For each application in the list and using the function çontains.ºf the vector class, the services provided by the application selected in the iteration are compared with the service selected from the Calls vector. If this application doesn't provide this service, the next application will be checked. If no applications are installed or the installed applications don't provide the services the new application calls, the loop will finish and the next step will be checked. Otherwise it will be necessary to check if the application which wants to call this service is allowed. To check this, the Allows vector of the application which provides the services is gone down. Each element in the vector is compared with the pair (service called, incoming Application) and the result is stored in the variable .exists". If the value obtained is true before the loop ends, this service is provided and allowed and the next services in the Calls vector are checked until finish. Otherwise, the service is provided but not allowed, violating the contract-policy matching and the algorithm returns false canceling the installation.

In the second part, all the services the incoming application needs are checked to assure that they are present on the platform. As before, a loop which goes through the vector is implemented, but this time the loop looks into the needs vector. .Existsïs again set to false, and other loop, which goes down the application list of the platform, is implemented inside the first one. Lately and using again the vector function çontains", the service selected in the first loop is searched in the Provides list of the application selected in the second loop. If the vector contains this service, .existsïs set to true finishing this iteration and the next service is checked. If it is not in the vector, the service is searched in the next application. If no application in the platform provides this service, the application is not compliance and the installation is rejected. Otherwise, the algorithm continues.

Finally, the services the incoming application provides are checked. Again, .existsïs set to false and the applications list is scanned. Using the çontains"function, the service provided is looked for in the Calls list of the application selected. If the element owns to the vector, then it is necessary to check if the application which calls the service is also allowed to use it. To achieve this, the Allows vector of the application which provides the service is gone through and the pair (service, Application) is compared with the elements in the vector, storing the result in .exists". If .existsïs set to true (the application is allowed to use it), the checking continues but if one service is not allowed, the application installation is rejected.

If the three parts of the algorithm are correctly passed, the installation is accepted returning true to the main program.

E.2.2.2. Check of Removal of Application

Firstly, the PolNeeds list is gone through using a loop until find the element which stores the information of the application is going to be removed. The index of this loop is a global variable called "position"which after the loop will store the position of the removal application to futures uses helping in the removal algorithm. Once the Dependents list of the application has been found, it is only necessary to check the vector size. If the size is bigger than zero, at least one application needs some of the services the application provides and it cannot be removed. Otherwise, none application needs the services it provides and the application could be removed from the platform.

E.2.2.3. Check of Compliance of AppPolicy Update

First of all, for this function and for the function in section E.2.2.4 will be necessary a piece of code (Listing E.9) which extracts the differences between two services lists. Because of that, this part of code is going to be explained here and referenced after. Let's suppose the system has two vectors (v1 and v2) with several services inside and it is necessary to extract the services from

v1 which don't have the vector v2. To achieve this, a loop which goes through the vector v1 is implemented. Inside this loop, using the çontains"function, it is checked if the vector v2 doesn't have the element selected in the loop and, if it has, the element will be added to an auxiliary vector.

```
for(int i = 0; i < v1.size(); i++){
    if(!v2.contains(v1.elementAt(i)))
        vAux.addElement(v1.elementAt(i));
}
```

Listing E.9: Extract function

Now, for the first part of the algorithm and using the function above explained the new services added to the Needs set are extracted. Once the new services have been added to the vector they should be checked. First, the vector filled before is scanned and "belongs", a boolean variable created at the beginning of the algorithm, is set to false inside the loop. Later, for each application in the platform, the service selected in the loop is searched in the Provides list of the application. If the service is provided, "belongsïs set to true and the loop continues with the next service. If all the services are provided, the algorithm continues with the next part of the algorithm, otherwise (some service is not provided) the algorithm returns false and the update is rejected.

Once the Needs services modification has been checked and passed, it is the turn for the Allows list. As in the first part of the algorithm, the differences between the old and the new policy should be extracted. In this case, instead of the Needs set, the differences are extracted from the Allows set, and, instead of the services added, the selected services are the removed. Because the Allows structure is a register, it is impossible to use the function çontains", so it is necessary to do in other way. To obtain that, it is necessary to make two nested loops: the outer one for the old services and the inner one for the news. Between both, the "belongs"variable is set to false. In the body of the inner, the services selected from the lists by the indexes are compared and, if they are equals, "belongsïs set to true breaking the iteration and continuing with the other services. If the inner loop finish its data and "belongs"value is still false, the service is added to an auxiliary vector. These iterations continue until all the services of the old Allows set have been checked.

```
Vector<Allows> vAuxAllows = new Vector(1,1);
for(int i = 0; i < app_policy_a_old.allows.size(); i++){
    belongs = false;
    for(int j = 0; j < app_policy_a_new.allows.size(); j++){
        if(app_policy_a_new.allows.elementAt(i).service.equals(
                app_policy_a_old.allows.elementAt(j).service)){
            belongs = true;
            break;
        }
    }
    if(!belongs){
        vAuxAllows.addElement(new Allows(app_policy_a_old.allows.
                elementAt(i).service, app_policy_a_old.allows.
                elementAt(i).application_id));
    }
}
```

Listing E.10: Extract function for allows structure

Now it is time to check if the removed services stored in the auxiliary vector are called by any application on the platform, therefore all the services on the list are compared with the applications IDs of the applications in the platform. If these values are equals, it is necessary to check if the application which is allowed to use the called service really uses it. Using the çontains"function the

service is searched in the calls vector. If the vector contains a reference to this service the update is canceled. Otherwise, the next service is checked.

When all the previous steps have been passed, the algorithm returns true and the update starts.

E.2.2.4. Check of New Contract Compliance with the Policy

Using the function to extract the differences between two vectors defined before (Listing E.9), the new services added to the Calls list are extracted.

In the first part, the vector with the services before extracted is scanned. Later, each application on the platform is selected and the service is looked for in their Provides list. If the service is provided by some application, its Allows list is checked in order to know if the callee application is allowed to use it. If it is, a boolean variable called "belongsïs set to true and thanks to the escape sentences the next service is checked. Otherwise, the algorithm returns false and the update is canceled.

The second part is identical to the Check of Compliance of AppPolicy (section E.2.2.3) update part one explained above.

The third part extracts the old Provides' services which are supposed to be removed using again the extraction method. Now, the applications list is gone through and if some of these applications still need one of the old services provided, the update is rejected.

In the forth part the new added Provides' services are checked. If the service is called by some application on the platform, it will be necessary to know if the application is allowed to use it in the new contract. If is allowed the algorithm continues until all the services has been checked. Otherwise, the new contract is not compliance and the update is refused.

For the fifth part, the algorithm does the same as in the Check of Compliance of AppPolicy Update (section E.2.2.3) explained before.

E.2.2.5. Policy Update for Approved New Application

As in the previous algorithm, a boolean variable called .exist"will be necessary, so it is declared in the beginning of the algorithm.

The first part, which adds the needed services to the Dependents lists, starts going through the Needs vector. Hereafter, .existïs set to false. For each application in the platform, the Provides vector is checked to know if it contains the service selected before. If this application contains it, the ID is stored in the PolNeeds vector. Once the application which provides this service is found, the .exist"variable is set to true in order to continue with the next services until finish with all the services.

Next, the application is added to the platform adding its contract to the application list.

In the final part a new empty vector is created which, together with the application ID, is added to the PolNeeds structure as the new Dependents information of the application. In the PolAllows case, the ID and the Allows vector of the application are stored.

E.2.2.6. Policy Update for Approved Application Removal

Before starting with the algorithm implementation, an integer variable called ïndexïs created and initialized to zero for future uses.

Now, first of all, the Dependents and the Allows lists of the application to be removed are erased

from the platform policy directly. This can be done because the position of this information has been stored previously in the check of removal of application algorithm (section E.2.2.2). After that, it is necessary to remove all the references the other applications have in their Dependents' list. To achieve this, the PolNeeds vector is gone down. For each element in the vector, a loop which does not finish until the boolean variable ïndexïs equal to minus one is implemented. In the body of the loop, ïndexïs assigned with the value returned by the function "vector.indexOf(element, index).ºf the vector class. This function returns the position of the element in the vector or minus one if this vector does not contain the element searched starting the search in the index position. In this case, the vector is the Dependents list in each PolNeeds element and the element searched is the ID of the application to be removed. The first search will start in zero, and the next ones in the position of the last element removed. When all the references have been removed from one application, the index is set to zero again and the loop continues with the next element.

After all the references to the application have been removed, the application is removed from the platform.

E.2.2.7. Update of the Policy after Approved AppPolicy

Firstly, using the .ªextract function"(Listing E.9), the new services added to the Needs list are extracted and stored.

Once the services have been extracted, the vector filled before is gone through. Then, for each application in the platform, the service selected in the loop is searched in its Provides list. If the service is provided, the ID of the application is added to the PolNeeds list in the position of the application which provides this service.

Secondly, the old services removed from the list are extracted. Next, the vector filled before is gone down and, for each application in the platform, the service selected in the loop is searched in the Provides list of the application. If the service is provided, the ID of the application is removed from the PolNeeds list in the position of the application which provides this service.

Finally, the Allows information should be actualized. To make this easy, the new one is always inserted in the same position replacing the old one. This is made in this way because is less costly to replace always the old information for the new one without check if this information has been modified. This is possible because both the applications and the policy are stored in the same position, so once one of them has been found, the other can be accessed directly.

E.2.3. Implementation of the system

Although the important part of the system is the algorithms and the data structures, it is necessary to test the correct functionally of these in a system inside of a Java Card simulator to make sure the system fits with it.

The previous part of the system has been packed in two different classes: Data Structures and Multi_Application_Framework. The first has the data structures aforementioned and the framework algorithms and more algorithms necessaries to simulate the system which are:

- One which checks if the platform has an application comparing the incoming ID with the applications in the card. This function is necessary because when an update arrives to the system, it is necessary to check if this application is already in the system.
- Two functions more, which check the correct functionality of the system, have been developed.

These functions show the applications information stored in the platform and the policy.

- Finally, two functions which store the changes in the contracts after an approved update have been implemented.

But the principal part of the system is the Input_applet class, which receives the APDU commands from the off-card system, checks if the commands are correct and with the information received invokes the different framework functions. In the first part of the algorithm, the constants and the variables necessaries to the execution of the system are declared like "policy"which will store the platform policy or "SxC"which is a reference to the class Multi_Application_Framework. After that, the functions necessaries to manage the smart card and the APDU commands are implemented. Besides the normal functions in an applet for smart card as the constructor, the install function, which calls the constructor and register the applet in the card, and the *process* function, other functions have been developed. Besides, the process function, which receives the APDU command from the off-card system has been modified. This function receives the APDU command from the off-card system which is checked to assure the APDU command class byte (CLA) is correct. In this case could be:

- 0x20 to add a new application
- 0x30 to remove an application from the card
- 0x40 to modify the AppPolicy of an application on the card
- 0x50 to modify the Contract of an application on the card

If the instruction is not correct, an error command is sent back to the off-card system. Otherwise the corresponding function is called. There are two different functions:

1. *manage_check_of_compliance* is the function which manages when a new application is wanted to be added or when an installed one is going to be updated. These functionalities have been put together because the similarities in their code. First of all, the information received from the APDU command is extracted and stored in a buffer. Now, depending on which action is going to be executed the information will change. If the action is an AppPolicy update, the application policy is stored; otherwise the entire contract is stored (for a contract update and for a new application). Once this information is stored, the applications in the system are compared with the ID of the incoming application. In the case of a new application, if this ID already exists in the platform, the update will be rejected because each application ID in the platform should be unique. If it is not in the platform, first the compliance is checked with the actual policy in the platform, and if it is compliance, the application is added to the application list and the policy is updated. For the other cases (AppPolicy and Contract update), if the application ID doesn't exist the update is rejected. If the application already exists, the compliance of the update will be compared with the old policy and, if it is still compliance, the policy and the application will be updated.
2. *manage_policy_update_remove_app* extracts the ID from the APDU buffer of the application is going to be deleted and, if the removal is accepted (no applications relay on this application) the update of the removal is executed. In this case is not necessary to check first if the application is already in the platform because if it's not, the function returns false and nothing is removed.

E.2.3.1. Problems

The main problems have arisen with the limitation of the Java Card in the use of the Vector structure, because the Vector class has a lot of useful functions which could have been used to make easier the implementation of the algorithms. The first problem is that the Vector class, in Java has a function called *clone()* which make a copy of the vector but the copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array. If this function had been available in the Java Card, the constructors of the structures which use vectors would have been easier. With this function had not been necessary to copy each element from the input vector to the vector the structure. Only assigning to the class attribute the input vector using the clone function will simplify the entire loop which have been used to do that.

Another problem is that, in the second version of the algorithms, sometimes is not possible to use the vector functions because the structures used for the Allows. Also if you want to look for a concrete application using the ID this function will help, but it is not possible to compare only the application ID.

E.2.4. Battery Tests

For each algorithm implemented, a battery test was performed in order to make sure the algorithm works as theoretically it should work following the SxC specification.

In each test, the APDU commands necessaries to test the algorithm functionality are presented and the results are explained.

These tests can been found in the Appendix G.

E.3. Performance

Due to one of the principal problems on the smart cards is the size the applications use to store the information, this should be taken in account to assure that the applications implemented in the simulator will fit with the real environment.

To test the space the applet which has been developed uses, several algorithms has been implemented with the objective of see how much space each data structure and the entire system need.

The memory tester has been implemented in the class *MemoryTestBench* which is composed of two algorithms. The first is called *calculateMemoryUsage* and it is the one which calculate the memory footprint. This algorithm starts constructing the object which is going to be tested and which is received as argument. Once the object has been created two variables called "mem0.ªnd "mem1.ªre created and initialized and the object which points to the class to be measured is assigned to null. Next the garbage collector is called several times to free the memory which is not in use and the memory used is now calculated. After that, the object is created and the garbage collector invoked again. Now the memory used is counted again and subtracted from the previous value. The result is returned to the second algorithm, which will show the results obtained.

To make possible testing the different objects instantiated in the framework, a public interface is created. This class is called *Object_test* and has one abstract function called *makeObject* which returns the object is going to be measured. Starting from this class, new classes are created for each data structure used. Each class implements the *Object_test* and the data structure to be measured. The abstract algorithm *makeObject* is implemented in these classes with the code where the data

structures are filled with different number of elements to known how the data increase its size when more information is added. Finally, this algorithm returns the object created.

Lastly, the *Main* class, where the instance of the *memoryTestBench* is initialized, is created. With this instance, the function *showMemoryUsage* is called and all the instances of the different date structures used in the implementation of the system are presented.

E.3.1. Results

For each different data structure used in the implementation has been measured different sizes as showed and explained below.

E.3.1.1. Strings

Due to memory usage is crucial to the application, so is understanding the memory usage of strings.

Although strings are actually ïnternalized", which means that only one instance of the same string is kept, it is also necessary to know the size which all the strings will size in the worst case (each string is unique) to maximize the space required to store the smart card information.

A String will have:

- 8 Bytes for the String class
- 16 Bytes for the character array
- 4 Bytes for the offset
- 4 Bytes for the String length
- 4 Bytes for the count
- 4 Bytes for the hash
- 2 Bytes for each character in the String

So, an string with 6 characters will size 40 Bytes for the information plus 12 bytes for the data.

For the performance of the structures no strings will be taken in account until the system performance.

E.3.1.2. Allows

Allows is a simple data structure which consists in two strings. The result of the test is shown in the Table E.1.

Object Size	Pointers	Total Size
8	8	16

Tabla E.1: Measure in Bytes of Allows data structure

In this result could be seen that an object needs 8 bytes to it instance and only 8 bytes to store the pointer to the strings.

Object Size	Pointers	String Vector 1		String Vector 2		Total Size
		Elements	Size	Elements	Size	
8	8	0	40	0	40	96
8	8	1	40	1	40	96
8	8	0	40	1	40	96
8	8	2	48	0	40	104
8	8	2	48	2	48	112
8	8	2	48	3	48	112
8	8	3	48	3	48	112
8	8	4	56	3	48	120
8	8	4	56	4	56	128
8	8	4	56	5	56	128
8	8	5	56	5	56	128
8	8	5	56	6	64	136
8	8	6	64	6	64	144
8	8	6	64	7	64	144
8	8	7	64	7	64	144

Tabla E.2: Measure in Bytes of Claim data structure

E.3.1.3. Claim

Claim, as has been explained in the previous chapter is a structure with two string vectors. As can see in the Table E.2, the object uses 8 bytes to it instance, 8 bytes more for the two pointers to the vectors, and:

- In the first case 40 bytes for an empty vector, but initialized with an empty position.
- In the next cases, the size of the vector increases when a even data is added due to the padding. For example, in the fourth row of data could be seen that the first vector sizes 48 bytes with two elements, but the second vector sizes 40 bytes with one element.

E.3.1.4. AppPolicy

Here, the AppPolicy structure is measured. This structure is composed by one string and two vectors, one with Strings and other with Allows elements inside. As in the previous cases, the object instance sizes 8 bytes and the other sixteen bytes for the vectors' and the String's pointers (4 for the String and 4 more for padding). The vectors size forty bytes when they are empty and ground in the next way. The string vector increases the space eight bytes for each two strings added as in the Claim as it can be seen in the second and in the fourth lines. In the case of the Allows vector, for each element added the size increases sixteen bytes and eight bytes more in each even data added as in the normal vectors. For instance, when the first element is added (line 2 in Table E.3), the size of the Allows vector increases in sixteen bytes, but in the line 5 of (Table E.3), when the second element has been added, the size has increased in twenty four bytes.

E.3.1.5. Contract

For the contract, because this structure is made up of two previous structures (Claim and AppPolicy), the size it uses is directly calculated as the addition of the sizes of the elements which

Object Size	Vector Pointer	String Pointer	String Vector Elements	String Vector Size	Allows Vector Elements	Allows Vector Size	Total Size
8	8	8	0	40	0	40	104
8	8	8	1	40	1	56	120
8	8	8	0	40	1	56	120
8	8	8	2	48	1	56	128
8	8	8	2	48	2	80	152
8	8	8	2	48	3	96	168
8	8	8	3	48	3	96	168
8	8	8	4	56	3	96	176
8	8	8	4	56	4	120	200
8	8	8	4	56	5	136	216
8	8	8	5	56	5	136	216
8	8	8	5	56	6	160	240
8	8	8	6	64	6	160	248
8	8	8	6	64	7	176	264
8	8	8	7	64	7	176	264

Tabla E.3: Measure in Bytes of AppPolicy data structure

made up the structure. This is: eight bytes of the object, eight bytes of the vectors pointers, eight bytes of the string pointer and the size of the AppPolicy and the Claim. In the Table E.4, the result of the previous tables has been used as the input data to calculate the examples of size of a Contract.

E.3.1.6. Applications

This is not exactly a data structure, but is the way which has been used to store the applications contracts in the application. This structure represents the size of all the applications stored in the smart cards. The size necessary is, as in all the previous vectors: forty when it is empty, forty plus the size of the elements in the other cases and eight bytes for each even element added. In the Table E.5, the first column corresponds with the size of the elements added to the vector in order: the first no elements, the second one element sized in two hundred and fifty six bytes, the third two elements, one with two hundred and fifty six (the previous one) and other with two hundred and eighty eight bytes, and so on. The second column represents the number the elements in the vector, the third is the size of the vector without the elements size and the last one is the total size of the applications in the platform.

E.3.1.7. PolAllows

This structure is which stores the PolAllows information, which make up the policy of the platform after the transformation depending on the security requirements. This structure has only an Allows vector, so the size it needs depends only the data stored in this vector and the constants. If the vector is empty, the size is 8 bytes for the object, 8 bytes for the pointer and forty for the empty vector. For each item added to the vector the size increases in 16 bytes and in 8 more if the item added is assigned to an even position in the vector.

Object Size	V. Pointer	S. Pointer	Claim	AppPolicy	Total Size
8	8	8	96	104	224
8	8	8	96	120	240
8	8	8	96	120	240
8	8	8	104	128	256
8	8	8	112	152	288
8	8	8	112	168	304
8	8	8	112	168	304
8	8	8	120	176	320
8	8	8	128	200	352
8	8	8	128	216	368
8	8	8	128	216	368
8	8	8	136	240	400
8	8	8	144	248	416
8	8	8	144	264	432
8	8	8	144	264	432

Tabla E.4: Measure in Bytes of Contract data structure

New Element Size	String Vector		Total Size
	Elements	Vector	
0	0	40	40
256	1	40	296
288	2	48	592
304	3	48	896
320	4	56	1224
352	5	56	1576
368	6	64	1952
400	7	64	2352
416	8	72	2232
432	9	72	3208

Tabla E.5: Measure in Bytes of Applications data structure

Object Size	Pointers	Allows Vector		Total Size
		Elements	Size	
8	8	0	40	56
8	8	1	56	72
8	8	2	80	96
8	8	3	96	112
8	8	4	120	136
8	8	5	136	152
8	8	6	160	176
8	8	7	176	192

Tabla E.6: Measure in Bytes of PolAllows data structure

Object Size	Pointers	String Vector		Total Size
		Elements	Size	
8	8	0	40	56
8	8	1	40	56
8	8	2	48	64
8	8	3	48	64
8	8	4	56	72
8	8	5	56	72
8	8	6	64	80
8	8	7	64	80

Tabla E.7: Measure in Bytes of Dependents data structure

E.3.1.8. Dependents

In this structure is where the Dependents of PolNeeds is stored. This structure has only a String vector, so the size it needs depends only the data stored in this vector and the constants. If the vector is empty, the size is 8 bytes for the object, 8 bytes for the pointer and forty for the empty vector. For each two items added to the vector the size increases in 8 bytes starting in the second element added.

E.3.1.9. Policy

Finally, this structure of the Policy is measured. This structure consists in two vectors, one Dependents and one PolAllows, what means that the size will be the sum of the size of the two vectors plus the constants. This means 8 bytes for the object, 8 for the pointers and the sum of the vectors sizes. Due to exists different ways to increase the size of this structure, three different tables has been attached. In the first one (Table E.8), the number of elements in the two vectors is constant and equal to one, but not the size of this element, because the number of services varies.

In the second table (Table E.9), the number of elements in the two vectors vary (Dependents and PolAllows), but the size of this elements is always the same and equal to fifty six and seventy two bytes respectively.

The last case (Table E.10) is a mixture of the two previous examples, and both the size of the elements and the number of elements varies.

E.3.2. Case of Study

Once each structure has been measured separately, is the moment to measure an example of the complete system as it could be in the real life. The real system is composed by three elements, the platform policy, the applications in the card and the integers and constants. The system is made up of five applications with different relations among them as could be in a real system.

In the table 4.2 could be seen the five applications which are in the system and which were inserted in the order which appears in the table. During the insertion of these applications in the platform, the policy was created and it can be shown in the Table 4.3.

Now, the quantity of the data is inserted in the different tables to obtain the total size of the system. To this values obtained from the tables is necessary to add all the constants and integer

Object Size	Pointers	Dependents Vector			PolAllows Vector			Total Size
		Elements	Strings	Size	Elements	Allows	Size	
8	8	0	0	40	0	0	40	96
8	8	1	1	96	1	1	112	224
8	8	1	2	104	1	2	136	256
8	8	1	2	104	1	3	152	272
8	8	1	3	104	1	3	152	272
8	8	1	4	112	1	3	152	280
8	8	1	4	112	1	4	176	304
8	8	1	4	112	1	5	192	320
8	8	1	5	112	1	5	192	320
8	8	1	5	112	1	6	216	344
8	8	1	6	120	1	6	216	352
8	8	1	6	120	1	7	232	368
8	8	1	7	120	1	7	232	368

Tabla E.8: Measure in Bytes of Policy data structure with only one copy in each Vector, but the size of the element in the vector changes

Object Size	Pointers	Dependents Vector			String Vector			Total Size
		Elements	Strings	Size	Elements	Allows	Size	
8	8	0	0	40	0	0	40	96
8	8	1	1	96	1	1	112	224
8	8	2	1	160	1	1	112	288
8	8	2	1	160	2	1	192	368
8	8	3	1	216	2	1	192	424
8	8	3	1	216	3	1	264	496
8	8	4	1	280	3	1	264	560
8	8	4	1	280	4	1	344	640
8	8	5	1	336	4	1	344	696
8	8	5	1	336	5	1	416	768
8	8	6	1	400	5	1	416	832
8	8	6	1	400	6	1	496	912
8	8	7	1	456	6	1	496	968
8	8	7	1	456	7	1	568	1040

Tabla E.9: Measure of Policy data structure with several copies in each Vector, but the size of each element is constant

Object Size	Pointers	Dependents Vector			Allows Vector			Total Size
		Elements	Strings	Size	Elements	Allows	Size	
8	8	0	0	40	0	0	40	96
8	8	1	1	96	1	1	112	224
8	8	1	2	104	1	1	112	232
8	8	1	4	112	2	2	216	344
8	8	2	3	168	2	1	192	376
8	8	2	2	168	2	4	256	440
8	8	2	2	168	3	2	288	472
8	8	3	4	232	3	1	264	512
8	8	3	2	224	4	1	344	584
8	8	3	2	224	4	4	408	648
8	8	4	3	288	4	5	424	728
8	8	4	6	304	5	1	416	736
8	8	4	3	288	5	2	440	744
8	8	5	5	352	6	1	496	864
8	8	5	3	344	6	1	496	856
8	8	5	2	344	6	4	560	920
8	8	6	2	408	7	4	632	1056
8	8	6	3	408	7	5	648	1072
8	8	7	2	464	7	1	568	1048

Tabla E.10: Measure of Policy data structure with a combination of both, increasing the number of elements in the vector and also the size of each element

used to the execution of the system. Also is necessary to add to this information the Strings' memory aforementioned. Assuming each String is stored independently the memory usage should be: forty times the number of strings plus two times the number of characters in the system which is roughly 4.5 kilobytes. This value is very high compared to the system structure, but it is necessary to understand that this value is not real because in a system where the services provided by one application are called by other, the strings are duplicated. In this instance, the true value counting only one copy of each string is 700 bytes which is a sixth part of the value calculated before. Furthermore, if the number of service the applications offers and needs is higher than the number of applications on the platform, this value will decrease.

The result can be seen in the table 4.4 without the constants aforementioned, which weight almost seventy bytes. Adding these two values, the total size of the system is roughly three Kilobytes. This means that and smart card with these five applets installed on it with a few interactions among they, in a Java Card platform smart card devices which typically have one hundred and twenty eight kilobytes of non-volatile read-write memory, should use only one of fifteen part of its memory to store the applications contracts and the policy of the platform, which means the card has enough memory to make possible the framework proposed in this project. It is also true that if the smart card has more and more applications, the space necessary to store it will be bigger because all the interaction between they, but it will be still a small part of the total amount of memory.

E.3.2.1. How to save memory

Optimizing Java for Size: these methods help saving memory in Java applications an in our particular case of Java Cards:

- Use the compiler optimization.
- Separate common code: if a specific function is used only in a few parts of the program copy the code of this function in each invocation to the function. This method has been used in some part of the framework's functions like in the function which separates the differences between two lists.
- Use short names for the functions and classes.

Also it is possible to save space with the strings which, in fact, are the data which uses more memory in this implementation. The idea is using the function "toHash", available in the String class which converts strings into integer values which sizes four bytes. This method was not implemented because it is more important to understand the outputs in order to test the correctness of the system.

Apéndice F

Conclusion

In this thesis Java Card has been presented focusing in the security features of them. After that, the current multi-application smart card problem has been explained and the SxC framework have been presented, described and implemented as a security framework for multi-application smart cards. In this approach, each application comes with a specification of its security behavior which must be compliant with the security policy of the hosting smart card platform. In particular, it have been shown how the SxC approach can be used to address different problems of smart card security, as the problem of illegal information exchange among applications on a single smart card, or the problem of preserving security states after dynamic changes in applications. The framework has been defined at the first level of the hierarchy called level 0 of applications model for smart cards.

With this theoretical approach, the implementation of the systems has been developed. The implementation has also been tested executing different tests in the system using the Java Card environment simulator. These tests have shown that the system implemented following the proposed approach is valid and functional and will solve the first level approach in real systems.

Also the system measures proves no only the system implemented is able to be stored, either it could stores lot of information of the incoming applications without get worried about the space they will use to store the policy and the contract using the current market technology.

Taken together, these results suggest that the system implemented is able to be exported to the real systems. This will be an important step in multi-application smart cards which are still not used as they should be used integrating this framework into the Java Cards.

Apéndice G

Battery tests

G.1. Check of compliance of new application

Because this algorithm could return false in three different cases, the test was divided in three parts also.

1. The services called by the new application are checked. For this test, three variant of the test were checked:

   a) In the first one, the test checks if the services called for the incoming applications are not provided then the application should be accepted. The first part represents the application is going to be inserted and the second represents the APDU command. In the first insertion, there is no problems (APDU responds with "01"which means correct) because no applications was installed before, and in the second neither because the service called "theïs not provided by the applications already installed.
   EMV@BANK:

   - Provides = (trans)
   - Calls = ()
   - Needs = ()
   - Allows = (trans, ePurse)

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1B 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x06 .ePurse"0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   ePurse@BANK:

   - Provides = ()
   - Calls = (the)
   - Needs = ()
   - Allows = ()

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x0F 0x06 .ePurse"0x00 0x01 0x03 "the"0x00 0x00 0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   b) For the second case, the test checks if one of the services called is provided and it is also allowed to this applications. In this case, the installation will be accepted. Like in the previous example, the first part represent the application incoming and the second

the APDU command. In this example the output is also correct because the service "transçalled by ePurse is provided by the application EMV already installed in the card.
EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, ePurse)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1B 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x06 .ePurse"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = ()
- Calls = (trans)
- Needs = ()
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x11 0x06 .ePurse"0x00 0x01 0x05 "trans"0x00 0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

c) Finally, the case where the service called for the incoming application is not allowed is presented. In this case, the application ePurse wants to call the service "trans"provided by the application EMV, but this application does not allow ePurse to use it and the installation is rejected (see APDU response = 00).
EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, LOA)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x03 "LOA"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = ()
- Calls = (trans)
- Needs = ()
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x11 0x06 .ePurse"0x00 0x01 0x05 "trans"0x00 0x00 0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

2. The services needed by the incoming application are checked. Two cases are check for this case.

a) First, if the services needed are provided for the applications already installed in the card. First, two applications which provides services are added. After that, the tested application is sent using the APDU command. Because the services needed by the incoming application are provided by the applications installed in the card, the card responses that the contract matches with the policy.

EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1B 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

jTicket@Transport:

- Provides = ()
- Calls = (trans, pay)
- Needs = (trans, pay)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x20 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x02 0x05 "trans"0x03 "pay"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

b) Lastly, the case when one of the services is not provided is checked. First, two auxiliary applications are installed. After that, the application to test is sent to the card. Because the service .oïs not provided by the card, the installation is rejected.

EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;

Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1B 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

jTicket@Transport:

- Provides = ()
- Calls = (trans, pay)
- Needs = (trans, pay, o)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x22 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x03 0x05 "trans"0x03 "pay"0x01 .o"0x00 0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

3. Finally, the services provided by the new application are checked. Three different cases are analyzed.

   a) First, when none of the services are called. Because none of the services provided by the incoming application are called by the applications already in the card, the installation is accepted.
   EMV@BANK:

   - Provides = (trans)
   - Calls = ()
   - Needs = ()
   - Allows = (trans, jTicket)

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   ePurse@BANK:

   - Provides = (the)
   - Calls = ()
   - Needs = ()
   - Allows = (the, jTicket)

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1B 0x06 .ePurse"0x01 0x03 "the"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

b) Secondly, the case when the incoming application provides a service which is called but the application that calls it is not allowed to use it is checked. In this case, because the application installed in the card called jTicket calls the service "trans.ªnd the incoming application provides this service but the jTicket is not allowed to use it, the installation is rejected.
EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .ᴱᴹV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

jTicket@Transport:

- Provides = ()
- Calls = (trans, pay)
- Needs = (trans)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, other)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x19 0x06 .ᵉᴾurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 .ºther"0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

c) Finally, the case when all the services the incoming application provides are called and allowed. In this case, because all the services provided and called are allowed, the installation is accepted.
EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .ᴱᴹV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

jTicket@Transport:

- Provides = ()
- Calls = (trans, pay)
- Needs = (trans)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

G.2. Check of removal an application

1. The first possibility is when an application is tried to be removed and the application Dependents list is not empty. In the example shown below two applications are installed in the card: EMV and ePurse. When the application EMV is tried to be removed from the card, its Dependent list is checked. Because jTicket needs "trans", the service EMV provides, the list is not empty and the removal is canceled.

   EMV@BANK:

   - Provides = (trans)
   - Calls = ()
   - Needs = ()
   - Allows = (trans, jTicket)

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   jTicket@Transport:

   - Provides = ()
   - Calls = (trans, pay)
   - Needs = (trans)
   - Allows = ()

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x01 0x05 "trans"0x00 0x7F;

Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to remove the application EMV: canceled

Sent APDU command (Delete command): CLA: A0, INS: 30, P1: 00, P2: 00, Lc: 0x04 0x03 .EMV"0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

2. The other possibility is when the Dependents list is empty. As it can be seen in the example listed below, the card installs first two applications. After that the EMV is tried to be removed but as it was explained in the previous example, it Dependent list is not empty and the removal is canceled. After this, the card receives the instruction to remove the application jTicket. Because its Dependent list is empty jTicket is removed from the card. After that, EMV is again tried to be removed. Because now its Dependent list is empty the application is removed from the card.

   EMV@BANK:

   - Provides = (trans)
   - Calls = ()
   - Needs = ()
   - Allows = (trans, jTicket)

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   jTicket@Transport:

   - Provides = ()
   - Calls = (trans, pay)
   - Needs = (trans)
   - Allows = ()

   Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x01 0x05 "trans"0x00 0x7F;
   Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   Try to remove the application EMV: canceled

   Sent APDU command (Delete command): CLA: A0, INS: 30, P1: 00, P2: 00, Lc: 0x04 0x03 .EMV"0x7F;
   Response: Le: 02, 00, 00, SW1: 90, SW2: 00

   Try to remove the application EMV: accepted

   Sent APDU command (Delete command): CLA: A0, INS: 30, P1: 00, P2: 00, Lc: 0x08 0x07 "jTicket"0x7F;

Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to remove the application EMV: accepted

Sent APDU command (Delete command): CLA: A0, INS: 30, P1: 00, P2: 00, Lc: 0x04 0x03 .EMV"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

G.3. Check of compliance of AppPolicy update

In this algorithm, two possibilities are checked: when the Needs list is modified adding new services and when the Allows list is modified removing old services.

1. For the first case two possibilities are considered:

   - First when the new services added to the list are provided. In this example two applications are added to the card. After that, the application ePurse is modified, adding to its Needs list a new service called "trans". Because the service is provided by the applications already installed in the card, the modification is accepted and performed.
     EMV@BANK:
     - Provides = (trans)
     - Calls = ()
     - Needs = ()
     - Allows = (trans, jTicket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     ePurse@Bank:
     - Provides = (pay)
     - Calls = ()
     - Needs = ()
     - Allows = (pay, jTicket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     Try to modify the application ePurse: accepted
     Sent APDU command (Update command): CLA: A0, INS: 40, P1: 00, P2: 00, Lc: 0x1B 0x06 .ePurse"0x01 0x05 "trans"0x01 0x03 "pay"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   - The other case is when the modification adds a service to the Needs list but this service is not provided by the applications in the card. In the example, EMV adds the service .another"to its Needs list, which is not provided and the update is canceled.
     EMV@BANK:

- Provides = (trans)
- Calls = ()
- Needs = ()
- Allows = (trans, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application EMV: canceled

Sent APDU command (Update command): CLA: A0, INS: 40, P1: 00, P2: 00, Lc: 00x1C 0x03 .EMV"0x01 0x07 .another"0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

2. For the second case three different cases are discussed:

   - The first one describes the case when the application referenced by the permission removed from the Allows list is not in the application collection. In the example, the Allows pair (pay, jTicket) is removed from the ePurse Allows list. Because the application referenced is not in the card the modification is accepted.

     ePurse@Bank:

     - Provides = (pay)
     - Calls = ()
     - Needs = ()
     - Allows = (pay, jTicket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     Try to modify the application ePurse: accepted

     Sent APDU command (Update command): CLA: A0, INS: 40, P1: 00, P2: 00, Lc: 0x09 0x06 .ePurse"0x00 0x00 0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   - The second example cover the case when the application referenced by the permission is in the application collection but the service is not called. In this example, the pair removed is (pay, jTicket) from the application ePurse. This pair makes reference to the application jTicket which is in the card, but the service "payïs not in its Calls list, so the modification is accepted.

     jTicket@Transport:

     - Provides = ()
     - Calls = (trans)
     - Needs = ()
     - Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 00x12 0x07 "jTicket"0x00 0x01 0x05 "trans"0x00 0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application ePurse: accepted
Sent APDU command (Update command): CLA: A0, INS: 40, P1: 00, P2: 00, Lc: 0x09 0x06 .ePurse"0x00 0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

- Finally, the case when the application referenced by the permission is in the application collection and the service is called. In this case, the application update is rejected due to the new contract does not match with the policy in the card. In the example shown below, the pair (trans, jTicket) is tried to be removed from the Allows list of EMV. Because the application jTicket is in the card and it calls "trans", the update cannot be accepted.
  EMV@BANK:
  - Provides = (trans)
  - Calls = ()
  - Needs = ()
  - Allows = (trans, jTicket)

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
  Response: Le: 02, 00, 01, SW1: 90, SW2: 00

  jTicket@Transport:
  - Provides = ()
  - Calls = (trans)
  - Needs = (trans)
  - Allows = ()

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
  Response: Le: 02, 00, 01, SW1: 90, SW2: 00

  Try to modify the application EMV: rejected
  Sent APDU command (Update command): CLA: A0, INS: 40, P1: 00, P2: 00, Lc: 0x06 0x03 .EMV"0x00 0x00 0x7F;

Response: Le: 02, 00, 00, SW1: 90, SW2: 00

G.4. Check of new contract compliance with the policy

In this algorithm is necessary to check five different cases, depending the sets which have been modified. Because two of them test the same cases like in G.3, only the new ones will be explained. These are:

1. A new service is added to the Calls list of the application. Three different cases will be discussed:

   - The first example explains when the service added is not provided in the smart card. As can be seen below, first the application is added to the smart card, and afterwards, the modification adds a new service to the Calls list called "transp". Because the service added is not provided the update is accepted and performed.
     ePurse@BANK:
     - Provides = (pay)
     - Calls = ()
     - Needs = ()
     - Allows = (pay, jTIcket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     Try to modify the application ePurse: accepted
     Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x22 0x06 .ePurse"0x01 0x03 "pay"0x01 0x06 "transp"0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   - The second example cover the case when the service added is both provided and allowed. In the example, one of the applications installed in the card called jTicket is going to be modified adding a new service to its Calls list named "pay". Due to this service is provided by the application ePurse and this application also allows jTicket to use it, the update is executed.
     EMV@BANK:
     - Provides = (trans)
     - Calls = ()
     - Needs = ()
     - Allows = (trans, jTIcket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     jTicket@Transport:

- Provides = ()
- Calls = (trans)
- Needs = (trans)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTicket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application jTicket: accepted
Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x00 0x02 0x05 "trans"0x03 "pay"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

- And finally, when the service is provided but it is not allowed for this application. In the example, the application EMV provides the service called "trans". The application ePurse is installed in the card and after that, it is tried to be modified adding a new service to its Calls list. Although the application is provided, ePurse is not allowed to use it and the modification is rejected.
  EMV@BANK:
  - Provides = (trans)
  - Calls = ()
  - Needs = ()
  - Allows = (trans, jTIcket)

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
  Response: Le: 02, 00, 01, SW1: 90, SW2: 00

  ePurse@BANK:
  - Provides = (pay)
  - Calls = ()
  - Needs = ()
  - Allows = (pay, jTicket)

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;

Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application jTicket: rejected
Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x21 0x06 .ePurse"0x01 0x03 "pay"0x01 0x05 "trans"0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

2. This case corresponds with the first one explained in the previous section G.3. The only difference is the format of the APDU commands because in this algorithm all the contract could be modified contrary to in the previous one which only could modify the AppPolicy.

3. This time the problem starts when a service has been removed from provides list. Two possibilities exist:

   - When the service removed is needed. In this example the application EMV provides a service called "trans"which is needed by the application jTicket installed in the smart card before the update. During the update, the services removed from the Provides list are checked and, due to jTicket needs this service, the update is rejected.
     EMV@BANK:
     - Provides = (trans)
     - Calls = ()
     - Needs = ()
     - Allows = (trans, jTIcket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     jTicket@Transport:
     - Provides = ()
     - Calls = (trans)
     - Needs = (trans)
     - Allows = ()

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     Try to modify the application EMV: rejected
     Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x16 0x03 .EMV"0x00 0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 00, SW1: 90, SW2: 00

   - The second case checks the case when the service is not needed and the update is performed. In the example, ePurse removes from its Provides list the service "pay". Because none applications in the smart card use it, the update is executed.
     ePurse@BANK:

- Provides = (pay)
- Calls = ()
- Needs = ()
- Allows = (pay, jTIcket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1b 0x06 .ePurse"0x01 0x03 "pay"0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application ePurse: accepted
Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x17 0x06 .ePurse"0x00 0x00 0x00 0x01 0x03 "pay"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

4. Contrary to the previous case, this checks the services added to the provides list. Three cases should be checked:

   - When the service added is not called. In this case, if the service added, in the example "new", is not called for the applications installed in the card and the update is performed.
     EMV@BANK:
     - Provides = (trans)
     - Calls = ()
     - Needs = ()
     - Allows = (trans, jTIcket)

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x1C 0x03 .EMV"0x01 0x05 "trans"0x00 0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     jTicket@Transport:
     - Provides = ()
     - Calls = (trans)
     - Needs = (trans)
     - Allows = ()

     Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

     Try to modify the application jTicket: accepted
     Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x01 0x03 "new"0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
     Response: Le: 02, 00, 01, SW1: 90, SW2: 00

   - When the service added to the Provides list is called but the applications which provides the service does not allow the callee to use it. In the example, the update tries to add the service "the"to the jTicket list. But this service is called by the application EMV but not allowed to use it, so the update is rejected.
     EMV@BANK:

- Provides = (trans)
- Calls = (the)
- Needs = ()
- Allows = (trans, jTIcket)

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x20 0x03 .EMV"0x01 0x05 "trans"0x01 0x03 "the"0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

jTicket@Transport:

- Provides = ()
- Calls = (trans)
- Needs = (trans)
- Allows = ()

Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application jTicket: rejected
Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x1C 0x07 "jTicket"0x01 0x03 "the"0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;
Response: Le: 02, 00, 00, SW1: 90, SW2: 00

- In the final one, the service added to the provide list is both called and allowed. In the example the update tries to add the service "the"to the jTicket list and a new pair (the, EMV) to the Allows list. Because the service is called, it is necessary to check if the application which calls this service is allowed to use it. Contrary to the previous case, now EMV is allowed to use the service and the update is executed.
  EMV@BANK:
  - Provides = (trans)
  - Calls = (the)
  - Needs = ()
  - Allows = (trans, jTIcket)

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x20 0x03 .EMV"0x01 0x05 "trans"0x01 0x03 "the"0x00 0x01 0x05 "trans"0x07 "jTicket"0x7F;
  Response: Le: 02, 00, 01, SW1: 90, SW2: 00

  jTicket@Transport:
  - Provides = ()
  - Calls = (trans)
  - Needs = (trans)
  - Allows = ()

  Sent APDU command (Insert command): CLA: A0, INS: 20, P1: 00, P2: 00, Lc: 0x18 0x07 "jTicket"0x00 0x01 0x05 "trans"0x01 0x05 "trans"0x00 0x7F;

Response: Le: 02, 00, 01, SW1: 90, SW2: 00

Try to modify the application jTicket: accepted
Sent APDU command (Update command): CLA: A0, INS: 50, P1: 00, P2: 00, Lc: 0x24 0x07 "jTicket"0x01 0x03 "the"0x01 0x05 "trans"0x01 0x05 "trans"0x01 0x03 "the"0x03 .EMV"0x7F;
Response: Le: 02, 00, 01, SW1: 90, SW2: 00

5. Like in the second part of this section, this case corresponds with one of the cases explained in the previous section G.3, in this case, with the second one.

Bibliografía

[1] Iso-7816. Technical report, ISO/IEC.

[2] *Java Card 2 Platform, Version 2.2.2*, 2005.

[3] *The Java Card 3 Platform*, 2008.

[4] *Development Kit User's Guide: Java Card 3 Platform, Version 3.0.2 Connected Edition*, 2009.

[5] *Java Card 3 Platform, Version 3.0.1*, 2009.

[6] *Java Card API, Connected Edition*, 2009. This is the Java CardTM application programming interface (API), Version 3.0.1 Connected Edition, which is a subset of the JavaTM programming language.

[7] *Global Platform: Card Specification. Version 2.2*, March 2006.

[8] A.S. Lee A. Abid, A. Mueen and H.Y. Tan. Object sharing in multi-function java smart card. *UNITAR E-Journal*, 3, 2007.

[9] Smart Card Aliance. About smart cards: Frequently asked questions, March 2010.

[10] P. Allenbach. Java card 3 platform. In *Java Mobile, Media & eMbedded Developer Days*, 2009.

[11] D. Gurov C. Sprenger and M. Huisman. Simulation logic, applets and compositional verificarion. *INRIA*, RR-4890, 2003.

[12] Z. Chen. *Technology for Smart Cards: Architecture and Programmer's Guide*, chapter 9. Addison-Wesley, 2000.

[13] D. Chindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. *LNCS*, 5189:32–47, 2008.

[14] A. Schairer P. Karger V. Austel G. Schellhorn, W. Reif and D. Toll. Verification of a formal security model for multiapplicatiove smart cards. *LNCS*, 1895, 2000.

[15] P. Girard. Which security policy for multiapplication smart cards? *USENIX Workshop on Smartcard Technology*, 1999.

[16] S. Hans. Java card platform overview, 2008.

[17] J. Domingo i Ferrer. Multi-application smart cards and encrypted data processing.

[18] G. Hancke I. Askoxylakis K. Markantonakis, M. Tunstall and K. Mayes. Attacking smart card systems: Theory and practice. *ScienceDirect*, Report 14:46–56, 2009.

[19] X. Leng. Smart card applications and security. *ScienceDirect*, Report 14:36–45, 2009.

[20] C. Bernardeschi M. Avvenuti and N. De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN n.12*, pages 20–27–28, 2003.

[21] C. Sprenger M. Huisman, D. Gurov and G. Chugunov. Checking absence of illicit applet interactions: a case of study. *LNCS*, 2984:84–98, 2004.

[22] F. Massacci and O. Gadyatskaya. Loading time policy certification for open multi-application smart cards.

[23] M. Montgomery and K. Krishna. Secure object sharing in java card. *USENIX Workshop on Smartcard Technology*, 1999.

[24] O. Gadyastskaya N. Dragoni and F. Massacci. Supporting applications' evolution for smart cards by security-by-contract.

[25] V. Wiles G. Zanon P. Girard P. Bieber, J. Cazin and J-L. Lanet. Checking secure interactions of smart card applets: Extended version. *J. of Comp. Sec*, 10:396–398, 2002.

[26] S. Basu S. Bahatkar R. Sekar, V. N. Venkatakrishnan and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *ACM Press*, pages 15–28, 2003.

[27] D. Sauveron. Multiapplication smart card: Towards an open smart card? *ScienceDirect*, Report 14:70–78, 2009.