ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

ESPECIALIDAD CIBERSEGURIDAD

SISTEMA PROACTIVO FORENSE BASADO EN TESTIGOS DIGITALES PARA LA GESTIÓN DE EVIDENCIAS DIGITALES EN ANDROID

PROACTIVE FORENSICS SYSTEM BASED ON DIGITAL WITNESS FOR THE MANAGEMENT OF DIGITAL EVIDENCE IN ANDROID

Realizado por
Silvia Cuenca Ramos
Tutorizado por
Ana Nieto Jiménez
Fco. Javier López Muñoz
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, septiembre de 2019

Resumen

Los dispositivos móviles recaban a diario gran cantidad de información personal (imágenes, ubicaciones, etc.) que desde la perspectiva de la informática forense resultan de gran utilidad, pudiendo ser empleadas como evidencias en un proceso judicial. Sin embargo, este proceso es delicado, pues hay que evitar comprometer la integridad de las evidencias electrónicas para que estas no puedan ser repudiadas. A día de hoy los dispositivos personales son contenedores de evidencias digitales, pero no participan activamente en la gestión de dichas evidencias, luego la adquisición y primer tratamiento de los datos depende mucho del terminal y de la pericia de los expertos en el uso de herramientas forenses, muchas de ellas comerciales. El objetivo principal de este Trabajo Fin de Master (TFM) es proponer una solución que permita gestionar las evidencias electrónicas desde un dispositivo Android haciendo uso del concepto de Testigo Digital. El Testigo Digital es una solución para la gestión de evidencias electrónicas que aprovecha los nuevos adelantos tecnológicos en seguridad embebida para que los dispositivos personales formen parte de la cadena de confianza. Durante el TFM se describe este concepto, así como el trabajo relacionado, a fin de entender mejor los requisitos que requiere para su desarrollo. Así mismo, se analizan los mecanismos de seguridad en Android que permiten implementar esta idea en un dispositivo de última generación. Como se comprobará, esta solución depende del hardware del dispositivo por las propias características del testigo digital, por lo que se aclaran los requisitos que debe cumplir la plataforma escogida si quieren usarse los resultados de este trabajo sobre otro dispositivo. Este trabajo presenta el diseño y desarrollo de un prototipo de testigo digital para Android que ha sido probado empleando el dispositivo Google Pixel 3, escogido por contar con el chip de seguridad Titán M, que cumpliría los requisitos para la testificación digital. La solución actual permite el almacenamiento y la transmisión de evidencias electrónicas de forma segura y verifica la identidad de los usuarios empleando la biometría.

Palabras clave

Testigo Digital, Secure Element, gestión de evidencias electrónicas, Informática Forense, OpenMobileAPI, *Android Keystore System*

Abstract

Mobile devices collect daily a large amount of personal information (images, locations, etc.) that from the perspective of forensic computing are very useful, and can be used as evidence in a judicial process. However, this process is delicate, because we must avoid compromising the integrity of electronic evidence so that they cannot be repudiated. Today personal devices are containers of digital evidence, but do not actively participate in the management of such evidence, then the acquisition and first processing of the data depends a lot on the terminal and the expertise of the experts in the use of forensic tools , many of them commercial. The main objective of this Master's Thesis (TFM) is to propose a solution that allows electronic evidence to be managed from an Android device using the Digital Witness concept. The Digital

Witness is a solution for the management of electronic evidence that takes advantage of new technological advances in embedded security so that personal devices are part of the chain of trust. This concept, as well as related work, is described during the TFM in order to better understand the requirements it requires for its development. Likewise, the security mechanisms in Android that allow to implement this idea in a last generation device are analyzed. As will be verified, this solution depends on the hardware of the device due to the characteristics of the digital token, so the requirements that the chosen platform must meet are clarified if the results of this work on another device are to be used. This work presents the design and development of a digital witness prototype for Android that has been tested using the Google Pixel 3 device, chosen for having the Titan M security chip, which would meet the requirements for digital witnessing. The current solution allows the storage and transmission of electronic evidence in a secure way and verifies the identity of users using biometrics.

Keywords

Digital Witness, Secure Element, electronic evidence management, computer forensics, *Android Keystore System*

Índice general

Índice de figuras

Índice de Códigos

Índice de tablas

1. Introducción

Los dispositivos personales se encuentran muy extendidos en la actualidad. Su uso por la sociedad es muy amplio, siendo algunos de sus principales usos acceder a redes sociales y a información de Internet en general y almacenar imágenes, vídeos y todo tipo de archivos que de esta forma podemos llevar siempre con nosotros.

Desde la perspectiva de la informática forense, esto plantea ciertas ventajas, a la vez que algunos inconvenientes. Por un lado, estos dispositivos son preciadas fuentes de datos acerca de un individuo, cuya información podría usarse como evidencia electrónica en un proceso judicial. Por otro lado, la manipulación de estas evidencias electrónicas plantea serios inconvenientes, en especial en el contexto de los dispositivos personales como móviles, tablets, etc., pues es un proceso muy delicado, existiendo incluso procedimientos concretos (p.ej. los mencionados en [1]) para evitar comprometer la integridad de las evidencias electrónicas. Un error durante el proceso de adquisición y análisis que ponga en cuestión la integridad de las evidencias podría conducir al repudio de dichas evidencias. Así, la admisibilidad de la evidencia electrónica como prueba en un litigio es de vital importancia.

Los recientes adelantos tecnológicos podrían aprovecharse para mejorar la gestión de evidencias electrónicas en nuevos escenarios como la Internet de las Cosas (IoT, por sus siglas en Inglés, *Internet of Things*). Por ejemplo, las nuevas características de seguridad de los dispositivos más recientes podrían emplearse para dar garantías sobre la integridad de la evidencia electrónica. Además, los múltiples sensores que incorporan muchos dispositivos y que permiten realizar mediciones y compararlas con otros dispositivos de alrededor pueden ayudar a aportar evidencias que permitan contrastar datos sobre diversos hechos.

Precisamente, el objetivo de este trabajo de Fin de Máster es el de diseñar y desarrollar una herramienta móvil proactiva que permita el almacenamiento y la transmisión de las evidencias electrónicas de forma segura, basándose en los trabajos previos de Testigo Digital (En inglés, *Digital Witness* [2]). Mientras que el Testigo Digital se centra en el diseño general de este tipo de soluciones y sus requisitos para la aceptación de la evidencia electrónica como prueba en un litigio, el objetivo de este TFM es ofrecer una solución para plataformas Android, sirviendo como una primera prueba tangible del concepto. Con este fin se analizarán las opciones disponibles de implementación de los requisitos para dicha plataforma, aprovechando los elementos *anti-tampering* presentes en las nuevas arquitecturas de seguridad de los dispositivos, en concreto el *Secure Element*. Para ello, será necesario realizar un análisis detallado del estado del arte y la viabilidad de las diferentes alternativas y tecnologías disponibles para satisfacer los requisitos del testigo digital, que están fuertemente ligados a la parte hardware y el cumpli-

miento de las normas para la gestión de evidencias electrónicas (p.ej. las normas UNE/71505 [3], UNE/71506 [4] o UNE/197010 [5]). Además, como parte del análisis identificaremos un conjunto de evidencias relevantes para realizar las pruebas del estudio.

El objetivo es permitir almacenar de forma segura las evidencias digitales para después transmitirlas a un dispositivo de características similares y continuar así la cadena de custodia de las evidencias electrónicas. Para el desarrollo de la herramienta se ha escogido el sistema operativo Android, por ser el más extendido a nivel mundial (Figura 1.1) y disponer de arquitecturas de seguridad embebida para el desarrollo que queremos realizar. No obstante, una parte importante de este TFM es determinar cuáles son las mejores opciones para realizar la prueba de concepto del testigo, dado que no todas las arquitecturas de seguridad son igualmente válidas para el fin perseguido de gestionar las evidencias digitales (p.ej. debido a limitaciones de recursos o falta de flexibilidad).

1.1. Motivación y Objetivos

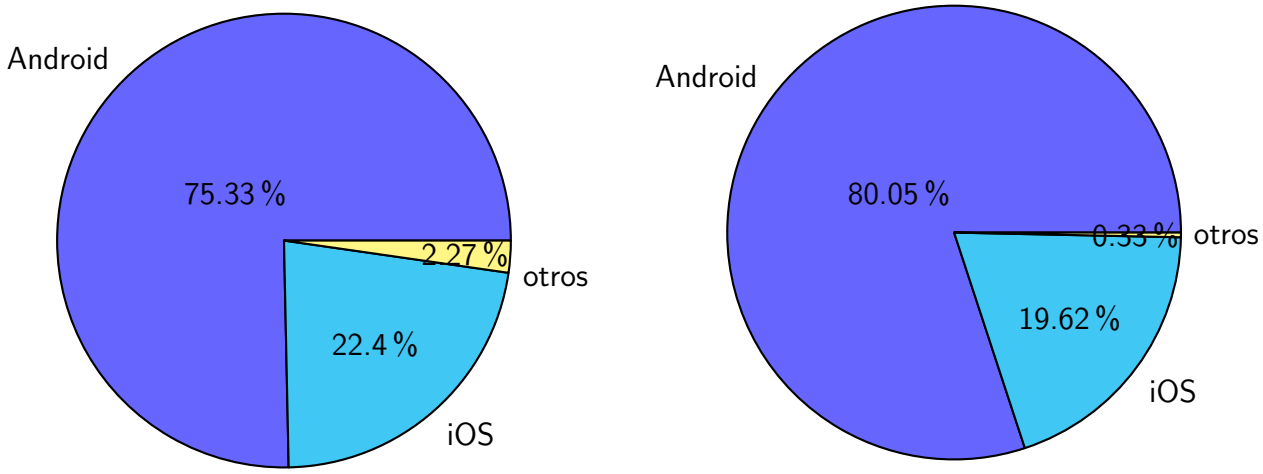
Los dispositivos personales forman parte intrínseca de nuestra vida diaria, siendo usados por más usuarios que los ordenadores. Como para numerosas tareas sustituyen a estos últimos, son preciados recursos de información que puede ser muy relevante en procesos judiciales. Sin embargo, el proceso de manipulación de esta información es compleja y debe asegurar la integridad y el no repudio de las evidencias.

El auge de estos dispositivos también ha llevado a que comiencen a incorporar nuevas características de seguridad, como el *Secure Element*, opciones de seguridad hardware que permite en algunos casos no sólo el cifrado, sino también almacenar de forma segura las evidencias electrónicas mencionadas anteriormente. Aprovechando estas nuevas características de seguridad de los dispositivos Android, se desarrollará una herramienta móvil para gestionar la adquisición y el almacenamiento de las evidencias electrónicas, de manera que aseguremos su integridad y admisibilidad como pruebas durante un litigio. Como podemos observar en la Figura 1.1 Android es el sistema operativo con el que cuentan más del 75 % de los dispositivos del mercado a fecha de marzo del 2019. Por tanto, se espera que esta solución o posibles variantes puedan ser usadas por un conjunto de dispositivos significativo.

Los objetivos de este proyecto de Fin de Máster son los siguientes:

O1. Analizar la viabilidad del concepto de testigo digital de forma práctica. Actualmente existen diversos trabajos teóricos sobre el testigo digital pero una prueba de concepto sobre la plataforma Android permitiría analizar de forma práctica la viabilidad del enfoque. No obstante, esto implica un análisis adicional sobre las arquitecturas de seguridad de los dispositivos Android para ver si son compatibles.

O2. Realizar un análisis detallado sobre las opciones actuales de salvaguarda de información de forma segura. En concreto se hará énfasis en las opciones actuales en las plataformas Android, con el objetivo de determinar qué soluciones pueden adecuarse mejor al desarrollo y puesta en práctica del testigo. Cabe destacar que al ser un enfoque proactivo



(a) Distribución mundial[1] (b) Distribución en España[2]

Figura 1.1: Sistemas operativos por dispositivos (Marzo de 2019)

se requiere capacidad para desarrollar software que la plataforma integre junto con los mecanismos de seguridad nativos.

O3. Desarrollar una herramienta móvil proactiva que sirva de prueba de concepto para demostrar la funcionalidad del testigo digital en Android. En particular, se espera contribuir en los siguientes aspectos como parte de la funcionalidad esperada:

O3.1. Permitir el almacenamiento seguro de la evidencia en el dispositivo y la transmisión de la evidencia entre dispositivos similares.

O3.2. Asegurar la integridad, confiabilidad y no repudio de la evidencia electrónica.

O3.3. Utilizar el hardware seguro del dispositivo para el desarrollo de dicha herramienta, siendo esta uno de los requisitos principales de la testificación digital.

1.2. Metodología

Para la realización de este trabajo se ha utilizado una metodología ágil, y diversas herramientas para su desarrollo. En concreto:

- Se ha utilizado una metodología ágil basada en iteraciones para una mejor división del trabajo a realizar.
- Se ha empleado la herramienta web *Trello* para una mejor planificación y control de las iteraciones.
- Se han marcado fechas de inicio y finalización de cada iteración y entregas internas de prototipos. Las fases de cada iteración del proyecto se muestran en la Figura 1.2.

[1]Fuente: *http://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201803-201903*
[2]Fuente: *http://gs.statcounter.com/os-market-share/mobile/spain/#monthly-201803-201903*

- Se han realizado distintas reuniones con los coordinadores del trabajo para mejorar el seguimiento del progreso del proyecto, utilizando la plataforma *ownCloud* como medio de intercambio de ficheros.

Figura 1.2: Fases del proyecto

1.3. Estructura de la memoria

La memoria se estructura en capítulos que abordan los diferentes aspectos estudiados en este proyecto teniendo en cuenta las fases descritas anteriormente.

Durante este primer capítulo se abordan la motivación y los objetivos del TFM, además de la metodología seguida para la coordinación, las fases y estructura. El segundo capítulo introduce los trabajos relacionados y las soluciones existentes para la adquisición de evidencias electrónicas que tienen especial relevancia por su relación con el TFM. Se introducirá también de forma más detallada del concepto de Testigo Digital, su implicación en la cadena de custodia digital y sus requisitos de diseño, muchos de los cuales son muy restrictivos.

Dichos primeros capítulos permiten sentar las bases de las necesidades y requisitos sobre los cuales se desarrollará el testigo digital para Android, que es el núcleo de este trabajo. Se analizarán las soluciones de la plataforma Android que permiten abordar los requisitos fundamentales del testigo digital y se planteará el diseño de la solución propuesta, comparándola con los trabajos relacionados. Los componentes principales en los que se divide la solución se analizan en el cuarto capítulo, en el que se destacan sus características principales y su implicación en este proyecto.

Atendiendo a estos componentes se desarrolla un prototipo de aplicación Android y el correspondiente servidor para la recepción y tratamiento de evidencias digitales. Las pruebas

de validación sobre el enfoque empleando el prototipo desarrollado se especifican en el sexto capítulo. Se define así la prueba de concepto, en la que se describe el entorno de pruebas escogido, además de las diferentes pruebas realizadas.

Posteriormente, en el capítulo siete se desarrolla una comparativa con soluciones anteriores, indicando lo aportado por este proyecto.

Finalmente, en el octavo y último capítulo se sintetizan las experiencias y resultados obtenidos durante este proyecto a través de las conclusiones. Además, se describirán posibles vías de trabajo futuro que podrán realizarse tomando como base este proyecto.

2. Análisis del trabajo relacionado

Debido a la relación de este trabajo (i) con la adquisición de evidencias digitales desde el dispositivo personal, en este caso teléfonos Android, y (ii) con el enfoque de testigo digital, como solución que se pretende aplicar en plataformas Android, este capítulo analiza los trabajos relacionados en ambas áreas.

Con el objetivo de ayudar a contextualizar el trabajo, se realiza una introducción a conceptos estrechamente vinculados con la gestión de evidencias electrónicas, identificando las fases donde se centra la solución propuests y haciendo hincapié en la obtención de evidencias.

En el caso del testigo digital, se introducirá también el concepto en sí mismo y sus requisitos específicos. Al ser un enfoque general, su adaptación a una plataforma Android no es trivial. Uno de los puntos clave para dicha adaptación será la de los componentes de seguridad hardware disponibles, por lo que se dedica una sección especial a este respecto.

2.1. Ciclo de Vida de una Evidencia Electrónica

Se entiende por *evidencia electrónica* cualquier dato digital almacenado o transmitido por un equipo informático [6]. Como se observa en la Figura 2.1, el ciclo de vida de una evidencia electrónica incluye seis fases generales, contempladas en la norma ISO/IEC 27037:2012. Así, este Trabajo de Fin de Máster se centra en las fases de adquisición o generación, almacenamiento y transmisión, haciendo especial hincapié en la fase de adquisición o generación de la evidencia. Cabe destacar que en este contexto *generación* hace referencia a considerar un dato como evidencia electrónica. A fin de evitar confusiones se emplearán en adelante los términos obtención o adquisición.

Figura 2.1: Ciclo de vida de una evidencia electrónica[1]

Así, se entiende que la evidencia electrónica se adquiere mediante técnicas forenses. Estas técnicas de adquisición de evidencias electrónicas se detallan en el Capítulo 2.2. Al igual que con una evidencia no electrónica, su adquisición da lugar al almacenamiento de esta. Posteriormente, se encuentra la fase de transmisión de una evidencia a una entidad autorizada. Estas fases nombradas anteriormente serán en las que se centrará la solución propuesta en este Trabajo de Fin de Máster, dado que, como se ha comentado, es uno de los problemas abiertos actualmente, y también uno de los puntos más críticos. Esta criticidad se debe a que si las técnicas para la gestión de las evidencias electrónicas en sus primeras fases se pone en cuestión toda la investigación podría ponerse en entredicho y afectar irremediablemente a un proceso judicial. Por lo tanto ofrecer soluciones que permitan verificar la procedencia de dichos datos en entornos dinámicos como IoT es uno de los desafíos actuales en informática forense.

Entre las técnicas que se están empleando en mayor medida destacan: (i) centrar los problemas de adquisición en plataformas intermediarias, tal y como se ha hecho con equipos de comunicaciones como routers, pero a mayor escala (e.g. Cloud), (ii) emplear *client-side forensics*, que consistirá en establecer el punto de control en los dispositivos, con lo que los usuarios pueden tener conocimento de este comportamiento, (iii) soluciones híbridas que combinan las dos anteriores, empleadas por los últimos trabajos que destacan la imposibilidad de obtener todos los datos del Cloud y otras plataformas y la pérdida de información por no tener acceso a los datos de los dispositivos [8].

Además de las fases nombradas anteriormente, estaría el segundo grupo de fases, pudiendo ocurrir estas un tiempo después de la transmisión de la evidencia a una entidad autorizada. Aquí se encuentran la recuperación de la evidencia para su procesamiento, el tratamiento de esta para la extracción de hechos, y la comunicación de dichos hechos, realizados siempre desde un punto de vista objetivo.

Todas estas fases anteriores conllevan sus riesgos, y deberán ser tratados de forma especial para poder garantizar la integridad, confiabilidad de las evidencias y evitar accesos no autorizado a dicha evidencia, para así mantener su integrar y evitar su repudio en un proceso judicial. Por tanto, durante todas estas fases anteriores se deberán seguir los principios forenses digitales [7]:

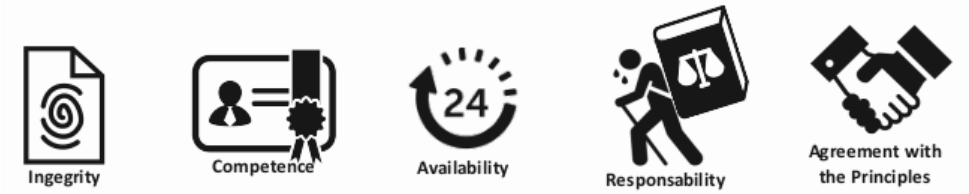


Figura 2.2: Principios forenses digitales conforme a la norma ISO/IEC 27037:2012[2]

- **Integridad:** Toda acción realizada sobre la evidencia (adquisición, almacenamiento, etc.)

[1][7]: Apuntes de la asignatura *Informática Forense* 2018, Ana Nieto, Máster Universitario en Ingeniería Informática

[2][7]: Apuntes de la asignatura *Informática Forense* 2018, Ana Nieto, Máster Universitario en Ingeniería Informática

deberá mantener esta inalterable. Si la integridad de una evidencia es puesta en duda, toda la investigación puede haber sido inútil.

- **Competencia o experiencia:** Toda persona que tenga acceso a la evidencia deberá tener experiencia en procedimientos forenses.
- **Disponibilidad:** Toda la actividad relacionada con la adquisición, acceso, almacenamiento, transmisión o tratamiento de la evidencia debe quedar documentado, preservado y estar disponible para su revisión en cualquier momento.
- **Responsabilidad:** Un individuo es responsable de todas las acciones tomadas respecto a la evidencia digital mientras esta se encuentre en su posesión.
- **Acuerdo con los Principios:** Toda empresa responsable de la adquisición, acceso, almacenamiento o transmisión de la evidencia digital debe de estar conforme con estos principios.

Junto a estos principios anteriores se encuentra implícito un último principio, la *repetibilidad*, que expresa que cualquier tercera parte independiente debe de ser capaz de repetir todo el proceso aplicado a la evidencia electrónica y obtener el mismo resultado.

Por tanto, se debe definir un procedimiento que siga los principios anteriores para que la evidencia sea admitida en un proceso judicial. Se define así el concepto de *cadena de custodia* como el sistema de control y registro de la secuencia de acciones llevadas a cabo por personal autorizado mediante el cual las evidencias son adquiridas, almacenadas, transmitidas y analizadas para que esta no se altere, destruya o pierda, para así ser válida durante un proceso judicial. Dicha definición queda recogida en estándar *ISO/PC 308*.

Se extiende la definición de la cadena de custodia a *cadena de custodia digital* cuando esta se refiere a la adquisición, almacenamiento, transmisión y tratamiento de evidencias digitales [2].

2.2. Obtención de evidencias en dispositivos móviles

La adquisición de evidencias se encuentra definida en la norma ISO/IEC 27037:2012 como el proceso documentado, reproducible y repetible mediante el cual se realiza la obtención de cualquier tipo de dato de carácter binario. Esta adquisición debe asegurar la cadena de custodia mencionada en los apartados anteriores, además de realizarse unicamente por personal autorizado, manteniendo siempre el histórico de todas las acciones realizadas.

Esta obtención de evidencias puede ser realizada sobre el *dispositivo origen*, actuando este como contenedor de evidencias, o sobre la *red*, realizando un análisis del tráfico. Además a su vez esta adquisición puede ser realizada en sin interrumpir el servicio (online, live), o una vez interrumpido el servicio, apagando el dispositivo (post-mortem, dead). La obtención puede estar sujeta a restricciones de tiempo en ambos casos, pero la obtención de evidencias en un sistema activo entraña el inconveniente adicional de que cualquier acción que se hace sobre el sistema deja trazas [7]. La solución desarrollada se centrará en la adquisición de evidencias en

el dispositivo, como entidad activa que etiquetará sus propias evidencias. En este sentido para la prueba de concepto se ha optado por la obtención de imágenes, verificando su integridad por el sistema destino.

Existen multitud de herramientas que obtienen evidencias de los dispositivos considerando éstos como contenedores, no gestores de evidencias electrónicas. Por ejemplo, aplicaciones de escritorio como *Oxygen Forensic Suite*, *MOBILedit Forensic Expres*, o incluso sistemas operativos orientados a ello, como *Santoku*, sistema operativo especializado en el análisis forense en dispositivos móviles basado en Debian [9].

Además de las herramientas mencionadas en el párrafo anterior, se encuentran otras herramientas que permiten el análisis de archivo en entidades externas, como *VirusTotal*, que mediante una *API Rest* permite el envío de ficheros para su escaneo en los servidores de este [10], además de aplicaciones propias que implementan dicha API [3].

Un enfoque novedoso lo plantea el Testigo Digital [2], en el que la obtención de evidencias se hace desde el propio dispositivo como elemento involucrado directamente en la gestión de la evidencia digital. Debido a su relación directa con este TFM pasamos a detallarlo.

2.3. Testigo Digital

Se define el concepto de *testigo digital* como el dispositivo que cuenta con un núcleo de confianza capaz de proteger la evidencia electrónica de modificaciones y accesos no autorizados, permitiendo la transmisión de esta a un dispositivo autorizado, pudiendo ser este dispositivo otro testigo digital [2]. Se entiende que el dispositivo que actúe como testigo debe de vincular la identidad del usuario al dispositivo.

Los requisitos del Testigo Digital descritos pueden enumerarse como sigue [2]:

REQ1. Núcleo de confianza: el dispositivo utilizado como testigo digital debe traer un núcleo de confianza de fábrica, aportando así confiabilidad a la acción que realiza. Dicho núcleo debe de evitar modificaciones y accesos no autorizados sobre la evidencia electrónica.

REQ2. Responsable humano: el testigo digital debe de asociar de alguna forma la identidad del usuario al dispositivo. Así, el usuario responsable del dispositivo será también responsable de las evidencias digitales que contenga y de las acciones que se realicen sobre ellas.

REQ3. Políticas de configuración: el testigo digital debe ser configurado en función del perfil del usuario vinculado. Por ejemplo, según si el usuario asociado al testigo digital pertenece a los cuerpos de seguridad del estado o no, este podrá almacenar información más o menos sensible.

REQ4. Registro de evidencias: el testigo digital debe de almacenar las evidencias, y todas las acciones realizadas sobre ellas, de forma segura en el dispositivo. Este almacenamiento necesita de un formato de evidencia electrónica utilizado tanto durante la adquisición como la transmisión de la evidencia.

[3] https://play.google.com/store/apps/details?id=com.funnycat.virustotal

REQ5. Transmisión de evidencias: el testigo digital debe de permitir la transmisión segura de evidencias entre dispositivos del mismo nivel o nivel superior. Además, siempre debe de almacenar un registro de esta transmisión para permitir el trazado de la cadena de custodia de la evidencia.

Se considera que un testigo digital es *testigo digital fuerte* si cumple con los requisitos **REQ1.** y **REQ2.**. De esta forma se protege la evidencia electrónica a a vez que se tiene potestad para actuar en nombre del usuario dependiendo de cómo haya sido configurado el dispositivo (permisos). Además, cuando el requisito **REQ2.** se cumple por parte de una persona representante del sistema legal se tiene la figura del *custodio digital*. La solución diseñada contempla el caso de testigo digital fuerte, en adelante testigo digital, y no considera las características específicas del custodio, que quedan fuera del ámbito de este trabajo. De hecho, la implementación de un custodio digital puede requerir hardware mucho más específico y preparado para almacenar gran volumen de evidencias.

El núcleo de confianza **REQ1.** según se define en el testigo digital se basa en la existencia de elementos de seguridad hardware embebidos en los dispositivos. Implica que un testigo digital no podría implementarse, a priori, en cualquier plataforma. Al ser una pieza fundamental para la solución propuesta se realiza a continuación un análisis de las opciones de seguridad hardware disponibles y su viabilidad para los intereses de este trabajo.

2.4. Seguridad Hardware para Dispositivos

Como se ha comprobado uno de los requisitos fundamentales para la implementación del testigo digital es el uso de seguridad hardware embebida. En esta sección se analizan las soluciones que podrían encajar con las características necesarias para el testigo digital.

Si bien solo algunas de estas soluciones serían viables para la solución específica en Android, sí es cierto que deben tenerse en cuenta porque otras implementaciones del testigo digital podrían usarlas.

Los diferentes módulos de seguridad hardware que almacenan información crítica como contraseñas han avanzado desde la aparición de los primeros tokens criptográficos, permitiendo en versiones posteriores proporcionar un núcleo de confianza a la plataforma en la que se implantaban. De esta forma surgen los distontos módulos seguros explicados a continuación.

2.4.1. Trusted Platform Module

El término *Trusted Platform Module* (TPM) se refiere a los criptoprocesadores seguros que pueden almacenar claves de cifrado en su interior. Esta especificación se encuentra disponible como la norma ISO/IEC 11889. Los módulos TPM son una solución hardware para el almacenamiento seguro. Estos chips aportan memoria (tanto volátil como no volátil), generador de números aleatorio seguro, algoritmo de generación de claves, funciones criptográficas para cifrado y descifrado y funciones hash, además de funcionalidades para permitir que la plataforma sea confiable.

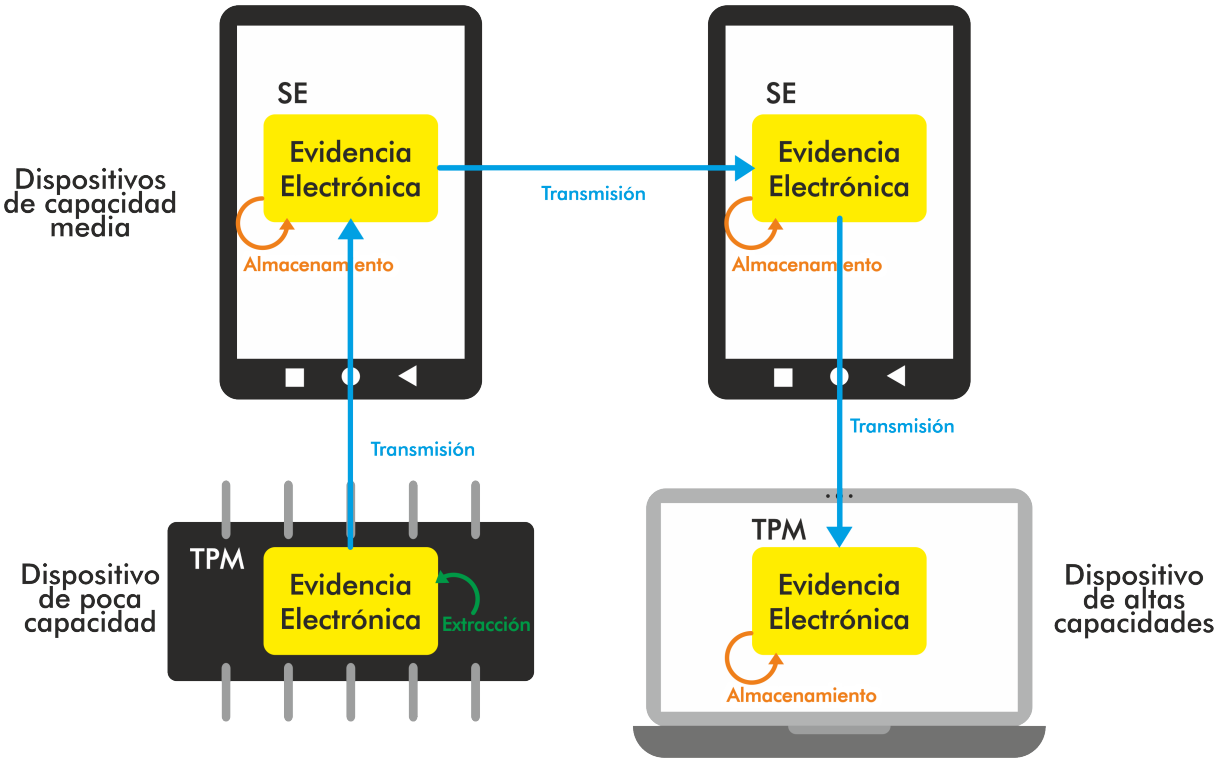


Figura 2.3: Testigos Digitales, ejemplo combinando TPMs y SE

Este módulo se encuentra separado físicamente del procesador principal, pero unido al circuito principal del dispositivo. Cuando se genera una clave o certificado, esta se sella dentro del TPM, resistiendo ataques lógicos y físicos. Además, estos integran una clave maestra de fábrica que nunca abandona el chip.

Este módulo cuenta con múltitud de aplicaciones, como podrían ser la gestión de derechos digitales (DRM) para evitar la piratería, cifrados de archivos y carpetas en distintas plataformas, correo electrónico seguro con características de firma digital, además de redes VPN entre otros.

Otra de las ventajas de estos módulos es que su utilización no se restringe unicamente a una plataforma, sino que puede ser utilizado tanto en ordenadores, smartphones o vehículos, entre otros. Por ejemplo, a partir de 2016 es obligatoria la implementación de módulos TPM en los nuevos equipos compatibles con Windows 10, ya sea ordenadores, tablets o smartphones [11]. También se pueden encontrar estos módulos en los vehículos, como el chip *OPTIGA TPM 2.0*, diseñado específicamente para proteger las comunicaciones entre los fabricantes de vehículos y los coches conectados [12].

2.4.2. Secure Element

Los *Secure Elements* (SE) son microchips que pueden almacenar datos sensibles y ejecutar aplicaciones seguras. Actúa como un almacén seguro, protegiendo lo almacenado en su interior contra ataques tanto físicos como software. Surgen como una evolución del *TPM* como solución anti-tampering para la gestión de claves, pero empleados en dispositivos personales. Su amplio despliegue se debe en gran parte a que se usan para flexibilizar las operaciones de pago.

El SE puede encontrarte en multitud de dispositivos electrónicos como móviles y *wereables*. En estos casos su uso suele venir ligado a la tecnología NFC, tecnología de comunicación de rango corto utilizada para realizar pagos desde el dispositivo móvil sin contacto con el dispositivo de pago. Además, este SE integrado puede ser utilizado para almacenar claves o hashes de datos biométricos del usuario del dispositivo.

De acuerdo a su uso, estos pueden ser clasificados en tres formas básicas [2]: SIMCards/UICC, microSD Cards o embebidos en el dispositivo.

Pese a que las tarjetas identificativas y los tokens son dispositivos anti-tampering por si mismos, estos tienen un propósito mucho más limitado que el SE. Es importante destacar que una vertiente importante de la comunidad de expertos considera que el SE es demasiado restrictivo al depender de componentes hardware e incrementar el coste de los dispositivos. Por ello surje como alternativa la tecnología *Host Card Emulation* (HCE) que utiliza el *Cloud* para realizar las consultas que haría al SE. Sin embargo, de esta forma se pierde el núcleo físico de confianza integrado en el dispositivo, pues en este caso se situaría en el *Cloud*, teniendo el control de estos una entidad externa. Mientras que esta solución puede ser efectiva de cara a la autenticación, contar con el Cloud como intermediario plantea conocidos problemas de cara a la gestión de evidencias electrónicas, dado que hay datos en los dispositivos que podrían perderse por ejemplo por una mala sincronización [13].

De forma más general, hay otras soluciones que se centran en aportar solución al concepto más general de *Trusted Execution Environment* (TEE). Algunos trabajos definen un TEE como un área segura dentro del procesador principal, que garantiza que el código y los datos de su interior están protegidos en cuanto a confidencialidad e integridad se refiere. Se encuentra aislado del código no verificado. Este TEE es utilizado actualmente por los operadores de telefonía para, junto a la tarjeta SIM, distribuir los puntos de acceso a la identidad del dispositivo y por tanto, del propietario, evitando así que si el terminal queda comprometido, el atacante tenga acceso a los datos de este [14]. Además, este método podría llevar en breve a la utilización del terminal para documento de identidad, como si de un DNI se tratase [15]. No obstante, cabe destacar que el concepto de TEE suele recaer en un núcleo de confianza, que suele basarse en el uso de chips como los mencionados anteriormente que permitan verificar la integridad del sistema desde su inicio.

2.4.3. Síntesis de soluciones

De acuerdo a lo anterior, la siguiente tabla muestra una comparación entre las soluciones, en la que también se destaca su idoneidad para el enfoque de testigo digital.

Así, en la Tabla 2.1 se observa que dado que la principal ventaja del SE frente a al HCE y el TEE es que este núcleo de confianza se encuentra en el propio dispositivo, pero aislado del procesador principal, por lo que lo hace la alternativa más adecuada para la implementación del Testigo Digital.

Por otra parte, a modo de ejemplo la Figura 2.3 muestra un escenario con cuatro testigos digitales, aunque el primero se obvia para mostrar únicamente el chip de seguridad anti-tampering que podría emplearse, en este caso el TPM, como se muestra en la Sección 3. Dicha

Tabla 2.1: Síntesis de soluciones

	TPM	**SE**	**HCE**	**TEE**
Ventajas	- Separado físicamente del procesador principal - Resiste ataques lógicos y físicos - No restringido a una única plataforma	- Separado del procesador principal - Si se encuentra embebido, aporta núcleo de confianza - Permite almacenar hashes, no solo claves	- Supera las limitaciones del SE - Más fácil de implementar - No depende del dispositivo	- Presente en dispositivos que no soportan SE. - Mayor nivel de rendimiento que un SE
Inconvenientes	- Solo almacena claves	- Depende del dispositivo	- Núcleo de confianza situado en el cloud - El control lo tiene una entidad externa	- Se encuentra en el procesador principal
Idoneidad para el testigo digital	Aporta núcleo de confianza, necesario para el testigo digita, pero no permite almacenar los hashes	Aporta núcleo de confianza y permite almacenar hashes, pero está muy ligado al dispositivo final	No aporta núcleo de confianza en el dispositivo	Al no encontrarse separado del procesador principal no es tan resistente ante ataques

transmisión combina dispositivos de distinto tipo, con soporte hardware. Cabe destacar que aunque la solución propuesta se centra en el uso de *secure element* (SE), conocer las opciones que pueden emplearse en otras plataformas es muy importante de cara a la extensibilidad.

La solución desarrollada se basa en el uso de SE. Cada dispositivo puede implementar de forma distinta el acceso a estos dispositivos TPM o SE. De hecho, en el caso de Android existen diversas soluciones que se analizarán a continuación. Para portar la solución propuesta a otras plataformas deberá realizarse un análsisi análogo para las plataformas específicas.

3. Mecanismos de seguridad en Android

En esta sección se describen las características de seguridad proporcionadas por la plataforma Android con especial énfasis en aquellas que son esenciales para el desarrollo de este TFM. En concreto, se describen de forma general los mecanismos de seguridad en la arquitectura del sistema Android, y posteriormente se detallan los mecanismos de conexión con el hardware seguro.

3.1. Arquitectura de Seguridad

La arquitectura de la plataforma Android se basa en una pila de 5 niveles, siendo desde el nivel más bajo hasta el más alto los siguientes: kernel de Linux, capa de abstracción de hardware (*HAL*), Android Runtime y bibliotecas nativas en C y C++, *framework* de Java API, y aplicacioness del sistema, como podemos observar en [16].

Este proyecto de fin de máster se centra en las capas del kernel de Linux y del *HAL*. La capa del kernel de Linux permite aprovechar funciones de seguridad claves de linux, además de utilizar otros métodos de seguridad propios de Android para proveer comunicaciones seguras entre los procesos. Como se puede comprobar en [17], estos métodos son principalmente el aislamiento en la ejecución de aplicaciones en un sandbox, una partición para el sistema y ejecución en modo seguro, sistema de ficheros basados en permisos, criptografía, arranque seguro, etc.

En cuanto a la capa *HAL*, esta ofrece interfaces para utilizar las capacidades del hardware en el *framework* de la API de Java. Esta capa consiste en módulos de biblioteca, cada uno de los cuales implementa una interfaz para un componente hardware específico. Cuando el framework de una API, por ejemplo la API de Java, intenta acceder al hardware del dispositivo, el sistema carga el módulo de la biblioteca para ese componente hardware.

En concreto, parte del trabajo desarrollado durante este TFM se centra en analizar el hardware *secure element* (SE) que empiezan a presentar los nuevos modelos. Como se observa en [18], el *SE* es un chip hardware *anti-tampering* independiente para almacenar información criptográfica. En dicho chip se pueden almacenar claves y asociarles alias para poder utilizarlas, pero todas las operaciones criptográficas relacionadas con dicha clave se realizaran en el interior del chip, por lo que estas claves nunca abandonarán el interior del chip. Además, el sistema operativo asegura que solo la aplicación que ha creado una clave puede utilizarla. Por otro lado, si se intentara abrir físicamente este chip para obtener sus claves, durante este proceso el chip se rompería y las claves se perderían, propiedad que se conoce con el nombre de *anti-tampering*.

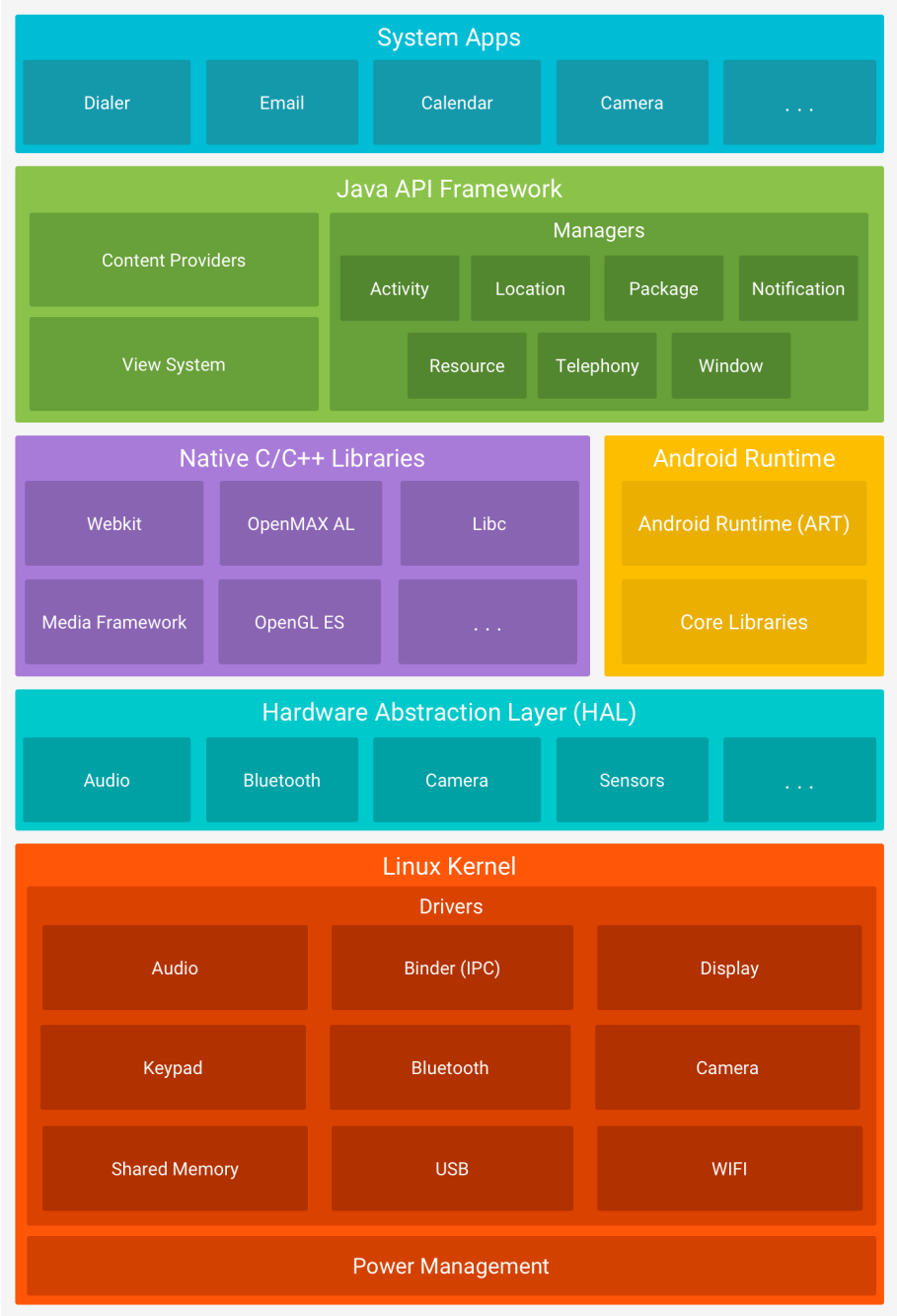


Figura 3.1: Arquitectura de la plataforma Android[1]

Se trata por tanto de un hardware que permite asegurar que nuestras claves se encuentran almacenadas de forma segura. Podría considerarse una evolución de otras soluciones previas como el TPM (Capítulo 2.4.1), pero adaptado a dispositivos personales.

Otra importante característica de seguridad del sistema relevante para este proyecto es el *Trusted Execution Environment* (*TEE*). Esta característica esta presente en los dispositivos a partir de *Android 7.0 Nougat*. Tal y como se vió en el Capítulo 2.4.2, se trata de un segundo entorno de ejecución, aislado de cualquier código no verificado. Al igual que el *SE*, el *TEE* asegura que la información y el código cargado aquí están protegidos en cuanto a integridad y

[1]https://developer.android.com/guide/platform/images/android-stack_2x.png

confidencialidad se refiere, pero en este caso, utilizando el procesador principal del dispositivo en lugar de un chip completamente aparte. El *TEE* se implementa utilizando la tecnología de *ARM Trust Zone* [19].

Existen por tanto, dos características bien diferenciadas entre sí, pudiendo existir ambas en el dispositivo: el *secure element*, que se trata de almacenamiento seguro en un chip hardware aparte, y el *trusted execution environment*, que se trata de un entorno de ejecución seguro, pero dentro del procesador principal.

3.2. Tecnologías para utilizar Secure Element en Android

La utilización directa de este chip por parte de los desarrolladores se encuentra muy limitada, unicamente *Google* y desarrolladores permitidos por este tienen esta opción. Sin embargo, el sistema aporta al resto de desarrolladores ciertos mecanismos para permitir la interacción con dicho chip. Estos son principalmente *Android Keystore System* y *Open Mobile API*.

A continuación se detallarán las tecnologías con las que acceder y utilizar el *Secure Element* desde Android, además del caso concreto del chip utilizado en el dispositivo de pruebas.

3.2.1. Android Keystore System

Android Keystore System fue introducido en Android de forma nativa en la versión API 18 (Android 4.3). Este sistema permite almacenar claves en un *almacén de claves*, restringiendo de esta manera su uso y acceso. El almacén de claves en este contexto se refiere a la ubicación dónde se almacenarán las claves criptográficas de forma segura, por tanto tenemos que el sistema *Keystore* es la interfaz proporcionada por Android para comunicarse con el almacén de claves del dispositivo, en este caso el *secure element*.

Como se observa en [20], las características de seguridad principales de este sistema son: i) los mecanismos para la prevención de extracción de claves, ii) el módulo de seguridad de hardware y iii) la autorización de uso de las claves. Para almacenar las claves mediante *Keystore*, se asocian alias a estas, de forma que sólo el alias sale del almacén de claves.

Dichas caracteríasticas se describen a continuación.

Prevención de extracción de claves

En cuanto a la prevención de extracción de claves, se dispone de las siguientes características para las claves almacenadas mediante este sistema:

- Las claves generadas mediante este sistema pueden estar vinculadas a hardware seguro, siempre que el dispositivo cuente con esta característica (*Trusted Execution Environment* o *Secure Element*). En este caso las claves nunca abandonarán el hardware seguro, pues todas las operaciones criptográficas que impliquen dichas claves se realizarán dentro de estos entornos. De esta forma, si el sistema operativo quedara comprometido, el atacante podrá utilizar las claves almacenadas en *Keystore* por cualquier aplicación, pero no podrá extraerlas. Es decir, en caso de que el dispositivo quedara comprometido, están claves no

podrían extraerse y utilizarse por ejemplo, para suplantar la identidad del usuario fuera del dispositivo, pero si podrían ser utilizadas desde otra aplicación.

- Las claves almacenadas nunca se utilizan en el proceso de la aplicación. Unicamente el proceso del sistema tendrá acceso a estas claves almacenadas en el hardware seguro. Así, si el proceso de la aplicación se encuentra comprometido, el atacante podrá utilizar las claves pertenecientes a la aplicación, pero al igual que antes, no podrá extraerlas y utilizarlas fuera del dispositivo.

Módulo de seguridad hardware

A partir de la version de la API 28 (Android 9) el sistema puede contar con una nueva implementación del *HAL* (llamado *Strongbox*) que contiene su propia CPU, almacenamiento seguro, generador de números aleatorios reales y mecanismos adicionales para evitar alteraciones e instalaciones no legítimas en el sistema. La principal diferencia entre utilizar esta versión del sistema o versiones inferiores radica en la situación del almacen de claves: la creación de claves en Android 9 mediante `KeyGenParameterSpec.Builder.setIsStrongBoxBacked(true)` asegura que las claves se encuentran en el *secure element*; mientras que las claves almacenadas en versiones inferiores cuyo `KeyInfo.isInsideSecureHardware()` indica que están almacenadas en hardware seguro no asegura que se encuentren en *secure element*, unicamente asegura que al menos se encuentran almacenadas en *Trusted Execution Environment* [21].

Autorización de uso de las claves

Durante la creación de una nueva clave se puede especificar si esta será vinculada unicamente en una aplicación, o por el contrario podrá ser compartida. Una vez creada esta clave con su autorización, no se permiten cambios en los permisos. Además, estas autorizaciones podrán estar tener un tiempo de expiración y requerir de la autenticación del usuario para su uso. La vinculación entre claves y aplicaciones se realiza mediante el paquete de la aplicación, es decir, una clave quedará asociada a la cadena correspondiente al paquete completo de la aplicación.

Tras pequeñas pruebas de concepto con *Keystore* durante la ejecución de este TFM, se puede destacar lo siguiente:

- Dos aplicaciones diferentes no tienen acceso a las mismas claves.

- Dos aplicaciones diferentes, pero con el mismo nombre de paquete, sí tienen acceso a las mismas claves. Cabe mencionar en este punto que ninguna de estas aplicaciones se encontraba firmada.

Esto implica que se podría suplantar una aplicación legítima simplemente creando una nueva aplicación y asignándole el mismo nombre de paquete, pues este nombre es fácilmente accesible desde el sistema de archivos del dispositivo, sin necesidad de que este se encuentre comprometido ni *rooteado*. De esta forma, la nueva aplicación tendría acceso a la utilización de todas las claves almacenadas en hardware seguro creadas desde la aplicación legítima. Sin

embargo, no hay que olvidar que estas claves podrían utilizarse, pero no extraerse para su uso fuera del dispositivo.

Además de este sistema *Keystore*, Android cuenta con un otro sistema de almacenamiento de credenciales llamado *Keychain*, cuya principal diferencia respecto al primero se encuentra en la privacidad de las claves, siendo *Keychain* recomendado cuando se desean utilizar credenciales en todo el sistema, y *Keystore* cuando se desea un control más exhaustivo de las aplicaciones que tienen acceso a estas. Como en el caso del testigo digital se desea un control más explícito sobre las aplicaciones no se ha considerado *Keychain* para este proyecto.

De forma resumida se destacan las siguientes características de *Android Keystore System* relevantes para el desarrollo del TFM:

- Se almacenan claves, no cadenas de texto.
- Las claves nunca abandonan el hardware seguro (siempre que el dispositivo cuente con este hardware).
- Cada clave tiene asociado un alias, que es lo único que abandona el hardware seguro.
- Las claves pueden ser accesibles unicamente por la aplicación que las ha creado, o compartidas entre aplicaciones.

3.2.2. Open Mobile API

Otra de las tecnologías que Android aporta para utilizar el *secure element* es *Open Mobile API* (*OMAPI*). Esta característica se trata de un mecanismo que permite a las aplicaciones autorizadas a comunicarse con los *applets* que se encuentran dentro del *secure element* [22], en concreto se trata de la especificación de una API para permitir la utilización de estos por las aplicaciones móviles. Esta librería fue introducida de forma nativa en el sistema a partir de la API 28 (Android 9), siendo necesaria su inclusión a mano en versiones anteriores si querían utilizarse sus características.

Utilizando esta API, las comunicaciones con el SE se realizan mediante mensajes *APDU* cumpliendo con el estándar *ISO/IEC 7816-4* [23]. Estos mensajes *APDU* son la unidad de comunicación entre una *smart card* y su lector [24]. Consisten en una secuencia de bytes con una estructura concreta, incluyendo cabeceras además de los mensajes. Estos mensajes estarían divididos en dos categorías (comando y respuesta), teniendo cada una de ellas su propia estructura.

La especificación oficial de esta librería [25] fue publicada en julio de 2018, al igual que la especificación concreta para la plataforma Android [26]. Esta publicación y su reciente inclusión en Android hace que haya sido muy complicado encontrar información o ejemplos concretos de su uso para este proyecto, más allá de estas especificaciones oficiales.

La forma concreta de utilización de esta librería en Android se verá en el Capítulo 4.

En la siguiente tabla se observa un resumen con las características de ambas opciones:

[2]Android Studio, enero 2019

[3]Android Studio, enero 2019

Tabla 3.1: Comparación de tipos de opciones para acceder al secure element

	Android Keystore System	**OMAPI**
Ventajas	- Gestión de claves - Las claves no abandonan el hardware seguro, solo el alias - Cada clave tiene un alias - Versión de Android >= API 18 (95.9 % de los dispositivos[2])	- Indica expresamente que *secure element* se utiliza
Inconvenientes	- Solo se almacenan claves (no cadenas de texto) - Aplicaciones diferentes con mismo nombre de paquete comparten claves	- Versión de Android >= API 28 (<1.0 % de los dispositivos[3]) - Release de julio de 2018, muy poca información

Finalmente, se optó finalmente por utilizar *OMAPI* para acceder al *secure element* en el caso del guardado de hashes, pues como se observa en la Tabla 3.1, los inconvenientes de *Android Keystore System* son muy importantes para este proyecto, debido a que en este caso es imprescindible que sólo la aplicación desarrollada en este Trabajo de Fin de Máster tenga acceso a las claves generadas.

Sin embargo, para el almacenaje de las claves de cifrado se optó por el sistema *Keystore*, pues es el que mejor aporta esta funcionalidad.

A continuación se analizará el caso concreto del *secure element* del dispositivo de pruebas utilizado. Un análisis sobre dicho chip de seguridad es necesario dado que sus características varían dependiendo de la versión, y son precisamente dichas características las que determinan su idoneidad o no para el desarrollo del testigo digital.

Titan M

Titán M es el chip de seguridad hardware más avanzado hasta ahora de los que se incorporan en teléfonos Android. Este chip está presente en el dispositivo de pruebas *Google Pixel 3*, del que se detallarán las características en los capítulos siguientes. Este chip fue creado especialmente para el dispositivo *Google Pixel 3* para asegurar los datos y el sistema operativo. Se trata de tomar las mejores funciones de los chips *Titan* utilizados en *Google Cloud* y adaptarlos para dispositivos móviles [27].

En el caso de la versión *Titán M* se protege el dispositivo de diferentes formas. Por un lado se encuentra integrado en el arranque verificado del dispositivo, por otro, se utiliza para verificar el código de desbloqueo del dispositivo y para encriptar el disco, produciéndose todas las operaciones criptográficas en el interior de dicho chip. Además, este chip es utilizado no sólo para proteger Android, sino también las aplicaciones de terceros y las transacciones, pues utilizando *Strongbox Keystore* las claves quedan almacenas en *Titán M*. Por último, este chip

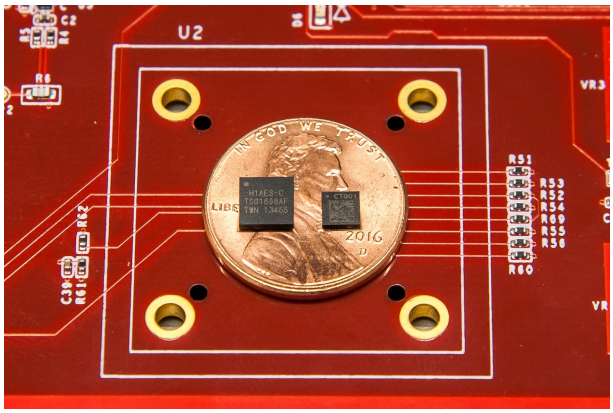


Figura 3.2: Comparativa de tamaño entre chip *Titán* (a la izquierda) y *Titán M* (a la derecha)[4]

evita manipulaciones, propiedad anti-tampering, tal y como se destacó anteriormente.

En cuanto a su ubicación en el dispositivo, este chip se encuentra aislado físicamente para evitar *exploits* a nivel hardware. Tanto el procesador, como la caché, la memoria y el almacenamiento persistente no son compartidos con el resto del sistema para evitar ataques *side channel*. Además, destaca también por su firmware open-source [28] (Figura 3.3).

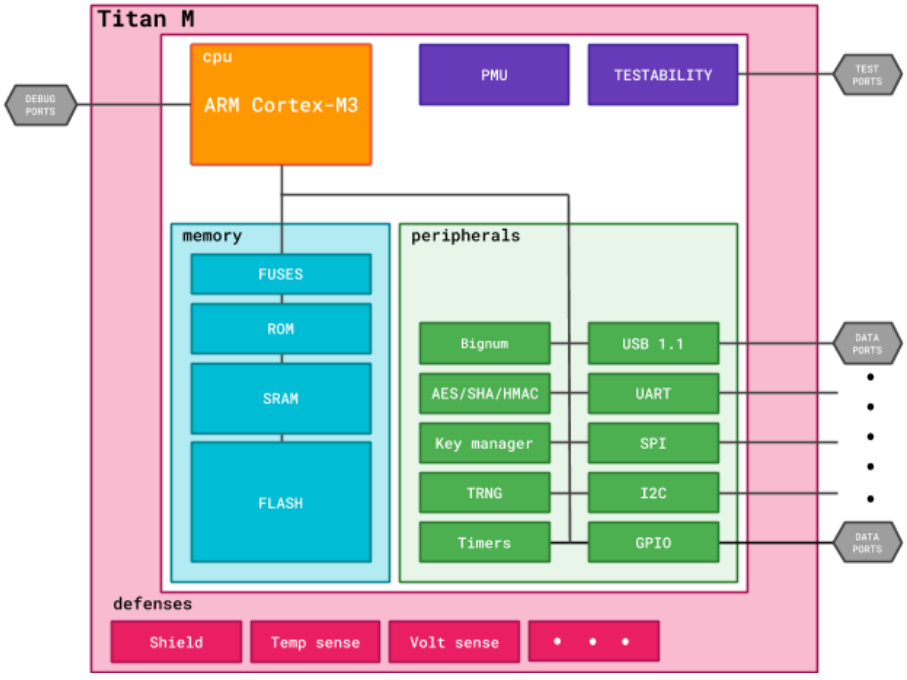


Figura 3.3: Diagrama de componentes de *Titán M*[5]

Por último, la serie de chips *Titan* se rigen por el protocolo FIDO [29] que utiliza técnicas estándar de criptografía de clave simétrica para proveer autenticación. Como se observa en la Figura 3.4, este protocolo se basa en que durante el registro con un nuevo sitio online, el dispositivo genera un nuevo par de claves, guardando la clave privada en el dispositivo y

[4]https://android-developers.googleblog.com/2018/10/building-titan-better-security-through.html
[5]https://android-developers.googleblog.com/2018/10/building-titan-better-security-through.html

enviando la clave pública al sitio online. La utilización de la clave privada del usuario únicamente se llevará a cabo una vez que este se ha autenticado en el dispositivo [30].

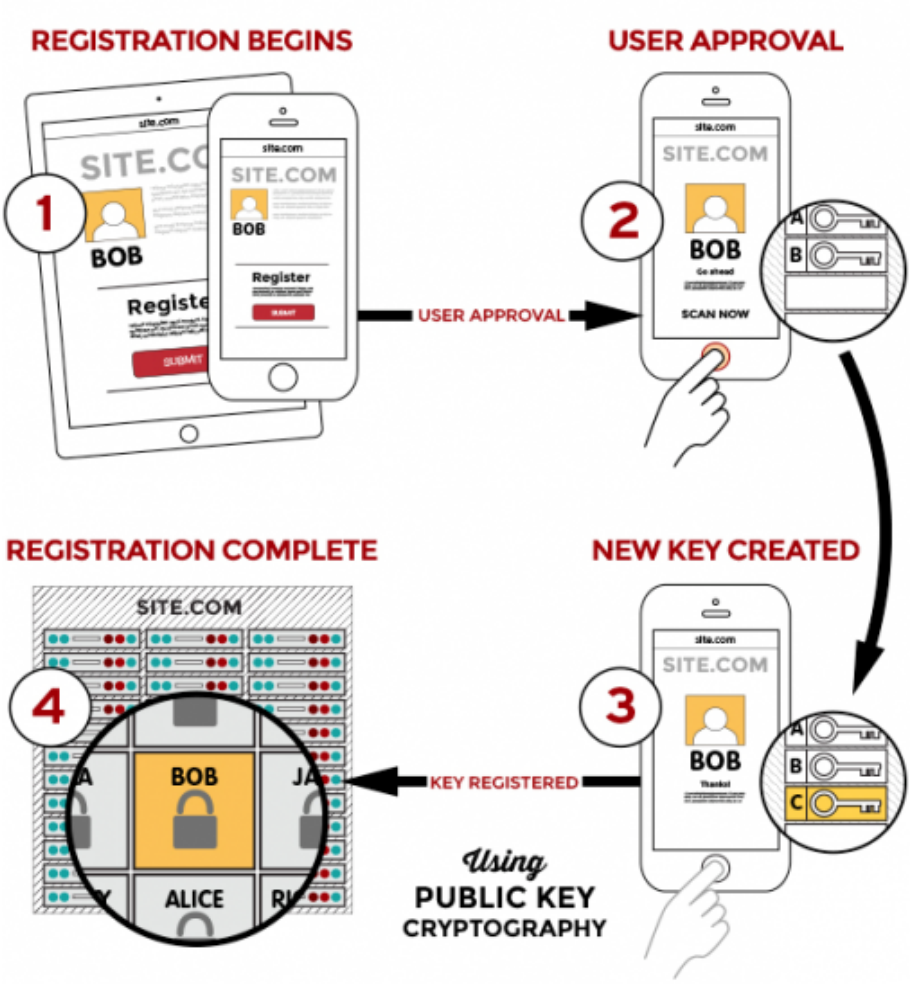


Figura 3.4: Protocolo FIDO[6]

[6]FIDO Alliance

4. Diseño del Testigo Digital para Android

Hasta ahora se ha descrito el problema concreto a resolver y el contexto donde se desarrolla el TFM. Las secciones anteriores analizan los diferentes mecanismos de acceso al hardware seguro en Android haciendo hincapié en su posible viabilidad para la testificación digital.

En esta sección se detallará el diseño de una solución que cubra los requisitos del testigo digital, sirviendo así como prueba de concepto para el testigo digital en Android. Cabe destacar que, aunque el testigo digital se encuentra definido y recogido en diversos trabajos mencionados así como en una patente, las directrices que proporciona son generales, para que puedan adaptarse a diferentes plataformas, desde wearables hasta vehículos. Este TFM propone una solución adaptada para plataformas Android, proporcionando también un prototipo para su validación. Por lo tanto esta sección propone un diseño específico tomando como base la retroalimentación durante toda la elaboración del TFM. Se destacarán aquellos puntos en los que la definición del testigo digital ha sido variada para permitir su adaptación a Android, y si esta variación tiene algún inconveniente destacable o, si por el contrario, representa una mejora no vislumbrada con anterioridad por la carencia de un prototipo en el Capítulo 5.

4.1. Abordando los requisitos del Testigo Digital

Como se detalló en la Capítulo 2.3, el testigo digital debe cumplir con una serie de requisitos básicos para ser considerado como tal: disponer de un núcleo de confianza (**REQ1.**), ser gestionado o estar autorizado por un responsable humano (**REQ2.**), atender a un conjunto de políticas de configuración, entre las que están las opciones de seguridad y gestión de las evidencias digitales (**REQ3.**), contar con un registro de evidencias (**REQ4.**), y ser capaz de transmitir las evidencias a otras entidades autorizadas (**REQ5.**).

Algunos de estos requisitos, además, deben complementarse con lo que se especifica en las normas de gestión de evidencias electrónicas que se encuentran en [2].

Teniendo en cuenta tanto los requisitos del testigo digital como las funcionalidades de seguridad que aporta Android, se analizarán a continuación las soluciones adoptadas para cada uno de estos requisitos.

4.1.1. Núcleo de confianza

El núcleo de confianza puede conseguirse con el *SE* o el *TEE* descritos en el Capítulo 3. El *SE* se encuentra soldado a la placa, por lo que lo consideramos más adecuado para este requisito. El soporte nativo para el *SE* ha sido introducido en la versión 28 de Android (*Android Pie 9.0*), por lo que se requiere que, además de contar con *SE*, el dispositivo disponga de, al menos, esta versión del sistema.

4.1.2. Responsable humano

Desde el inicio de los dispositivos móviles estos están asociados a una persona física mediante la tarjeta SIM, pues para obtener una es necesario la identidad de una persona física. Además, estos dispositivos son personales e instransferibles. Sin embargo, para el propósito del testigo digital esta asociación no es suficiente y requiere de otro método de identificación unequívoca de la persona. Es por ello que se requiere de la identificación biométrica aportada por Android a partir de *Android 6.0 Marshmallow*, en concreto la huella dactilar, para esta vinculación usuario-dispositivo y permitir así la autenticación y autorización de usuarios. De esta manera, el usuario queda vinculado al dispositivo, siendo responsable de las evidencias almacenadas en él y de las acciones que se realicen sobre ella.

4.1.3. Políticas de configuración

Este requisito debe cumplirlo la aplicación para el testigo digital más que el dispositivo en si. El dispositivo debe permitir la conexión con servidores remotos, recayendo sobre la aplicación la tarea de aplicar esta configuración. Por tanto, el dispositivo debe de tener algún tipo de conexión (3G/4G, Wifi, Bluetooth,...). El uso de una conexión de corto o largo alcance dependerá de la aplicación y del servidor, y de los requirimientos de estos para la configuración de dispositivos.

4.1.4. Registro de evidencias

Este requerimiento se refiere al almacenamiento de la evidencia, por lo que se requiere que el sistema cuente con un sistema de almacenamiento seguro, ya sea un sistema de archivos o base de datos. El sistema Android permite el almacenamiento privado de sus aplicaciones [31], a dicho almacenamiento solo tiene acceso la aplicación en cuestión, quedando restringido el acceso del resto de aplicaciones y del usuario. Las bases de datos de las aplicaciones se sitúan en dicho almacenamiento, por lo que de igual manera sólo serían accesibles desde la aplicación propietaria. Además, el sistema permite el cifrado de archivos y de base de datos, por tanto, la aplicación podría cifrar las evidencias y sus datos antes de su almacenamiento. Sin embargo, tal y como comprobaremos más adelante, cuando nos encontramos ante un dispositivo con acceso *root*, esta ubicación deja de ser privada, pues el usuario puede acceder fácilmente desde el sistema de archivos y consultar sus datos, por tanto, el cifrado tanto de datos como de base de datos se hace obligatorio para cumplir este requisito.

4.1.5. Transmisión de evidencias

Al igual que con las políticas de configuración, el sistema debe de contar con algún tipo de conexión (3G/4G, Wifi, Bluetooth, NFC,...), pero será la aplicación la que defina la conexión final necesaria para dicha transmisión. Se incluyen aquí tanto el envío de datos a un servidor, que podría requerir de conexiones de largo alcance, como de transmisión de datos entre dispositivos cercanos. En cuanto a los datos necesarios para la transmisión, se pueden obtener los relativos al dispositivo conociendo el modelo.

4.1.6. Resumen de características técnicas para abordar los requisitos

Se pueden resumir los requerimientos anteriores en dispositivos Android que cuenten con las siguientes características para ser utilizados como testigo digital:

- Android Pie 7.0 o superior (Versión 28 o superior).
- Disponga de *SE* integrado en la placa.
- Disponga de autenticación biométrica.

4.2. Especificación de componentes principales para el almacenamiento de las evidencias digitales

Considerando los requisitos y las soluciones que aporta Android, se propone la siguiente solución. El diseño se orienta al desarrollo de una aplicación que funcione como testigo digital, asegurando la integridad y el no repudio de la evidencia, desde su adquisición, hasta su transmisión a un servidor autorizado, manteniendo siempre la cadena de custodia y la trazabilidad de las evidencias digitales.

La Figura 4.1 muestra el escenario deseado para el testigo digital [2]. En azul se encuentran marcados aquellos componentes que cuya funcionalidad será diseñada de forma específica para Android en este TFM.

Durante este proyecto se definen dos entidades para la comunicación principales: el sistema Android y el servidor web. Sin embargo, es necesario entender primero el enganche con la arquitectura del testigo digital que se basa en la definición de los componentes señalados de la Figura 4.1. De esta manera, los siguientes apartados de esta sección abordan los criterios por los cuales se ha regido la adaptación para Android de los cuatro componentes principales diseñados: el gestor de operaciones usuario-dispositivo (Capítulo 4.2.1), el gestor de evidencias electrónicas (Capítulo 4.2.2), la base de datos para almacenar las evidencias cifradas (Capítulo 4.2.3) y el almacenamiento seguro con control de acceso (Capítulo 4.2.4).

A continuación se detallarán cada uno de estos componentes, y su funcionalidad en este proyecto. No serán considerados aquí los mecanismos criptográficos ni el gestor contractual, pues en el caso del primero este viene dado de forma intrínsica por el dispositvo, por lo que no

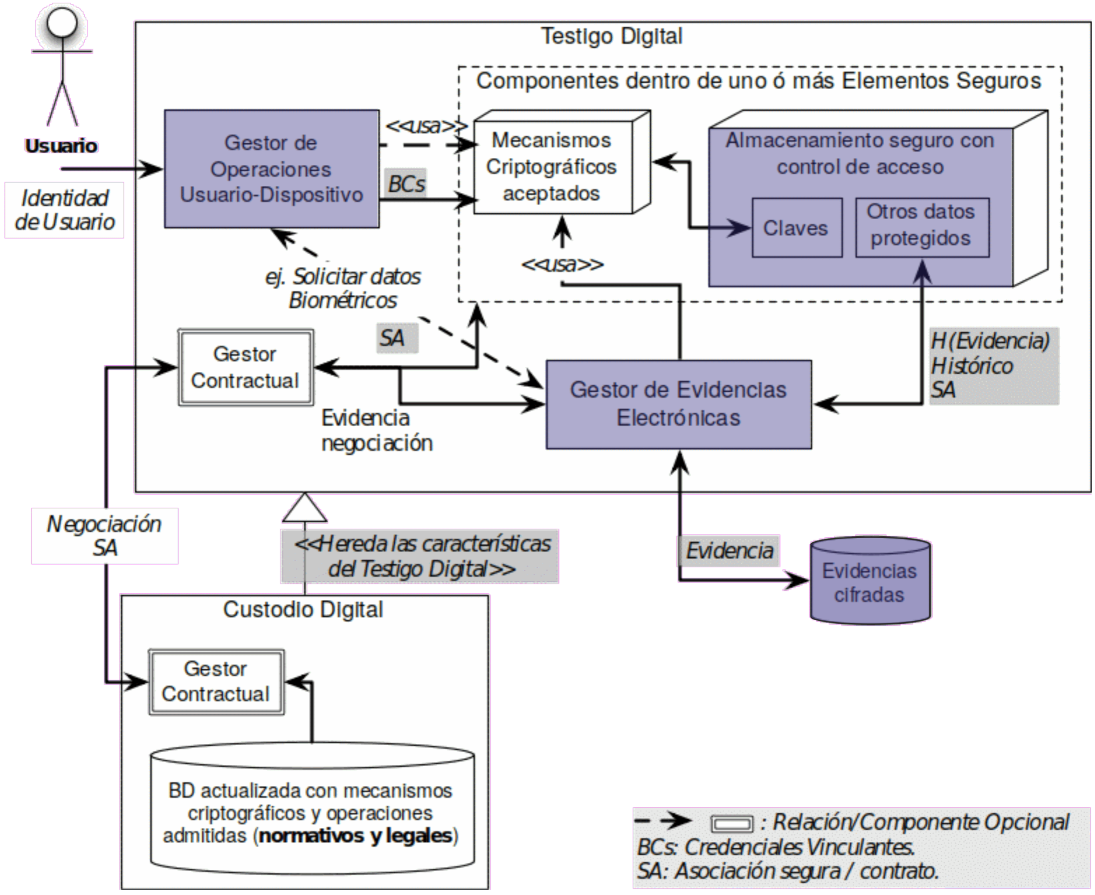


Figura 4.1: Escenario del testigo digital

requiere de implementación, y en el segundo caso este gestor es parte de la figura del custodio digital, quedando este fuera de los objetivos de este TFM.

Los Capítulos 4.3 y 4.4 se centran en definir, respectivamente, el diseño de la aplicación y del servidor.

4.2.1. Gestor de operaciones Usuario-dispositivo

El gestor de operaciones Usuario-dispositivo se corresponde con la aplicación de usuario. Esta es la encargada de realizar todas las operaciones necesarias para la creación de credenciales que permitan vincular al usuario y al dispositivo. Además, esta es la encargada de realizar de realizar la comunicación con el resto de componentes para permitir así la adquisición y transmisión de evidencias. Esta aplicación será detallada en el Capítulo 4.3.

4.2.2. Gestor de evidencias electrónicas

A continuación, se hace referencia al gestor de evidencias electrónicas, el cuál se corresponde con el gestor de almacenamiento en la Figura 4.2. Este gestor se utiliza para almacenar las evidencias cifradas en el dispositivo. Para ello, durante la adquisición de la evidencia, esta se cifra y se almacena en este gestor, utilizando para ello una extensión propia.

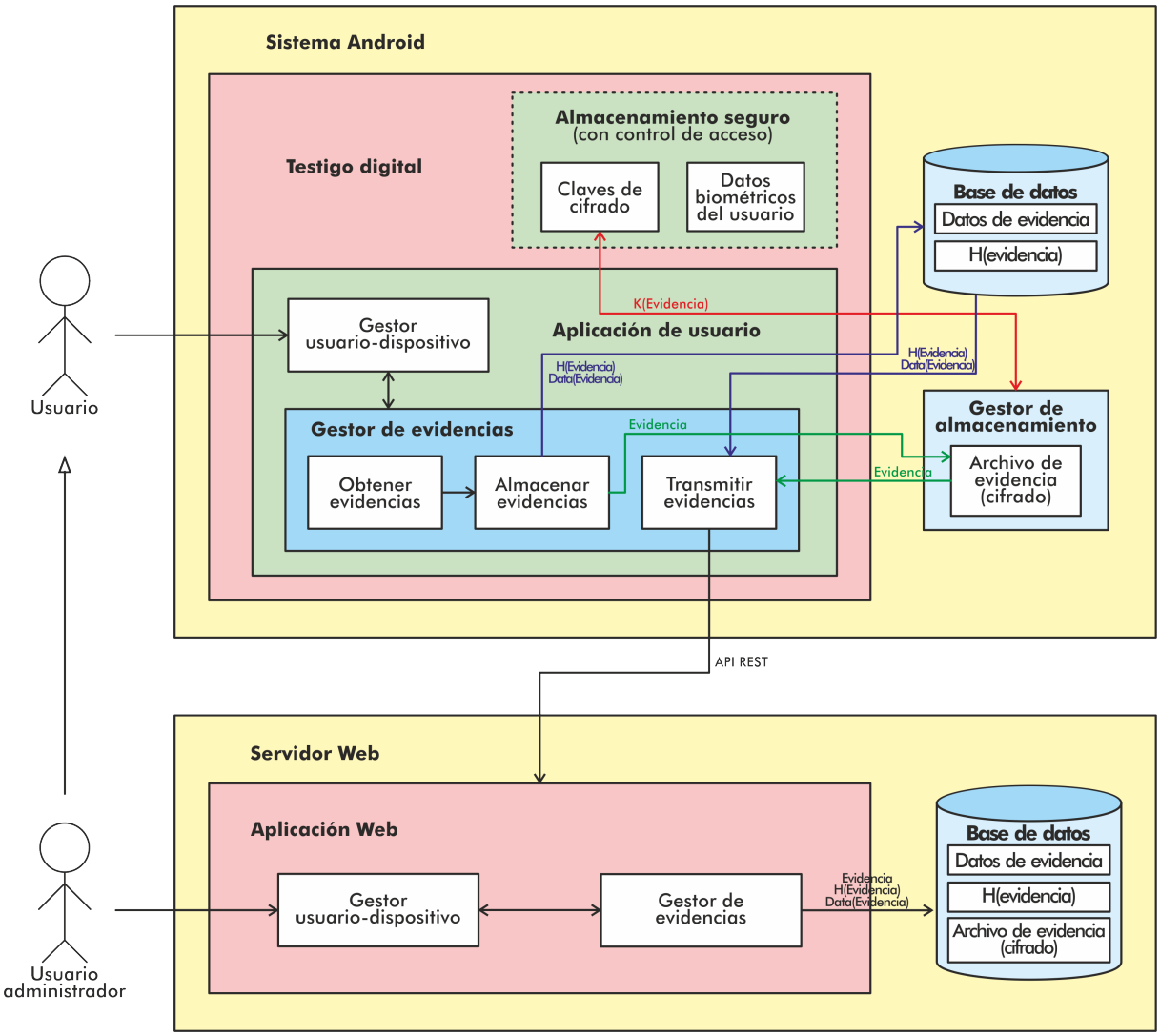


Figura 4.2: Componentes principales

En el caso del servidor web (Capítulo 4.4), sólo se tendrán la aplicación web y la base de datos, tal y como muestra la Figura 4.2.

4.2.3. Base de datos para almacenar las evidencias cifradas

En esta sección se detallará la base de datos y su cifrado, haciendo especial hincapié en aquellas secciones más relevantes para el propósito de este proyecto.

Se ha optado por utilizar *SQLite* como motor de base de datos, pues se integra muy fácilmente con *Android*, además de ser muy simple y fácil de usar. Sin embargo, este motor no tiene mecanismos de seguridad, y aunque *Android* restringe el acceso a la carpeta donde se almacena la base de datos de la aplicación, si el dispositivo se encuentra *rooteado* esta carpeta es fácilmente accesible desde el sistema de archivos. Como se observa en la Figura 4.3, una vez se accede a la carpeta de la base de datos, se puede abrir el archivo y ejecutar consultas sobre dicha base de datos. Además, se comprueba en la Figura 4.4 que si se extrae el archivo, este guarda parte de su contenido en texto plano, como muestra [32]. Estas capturas han sido

tomadas sobre un emulador con *Android 9.0*, que por defecto se encuentra *rooteado*.

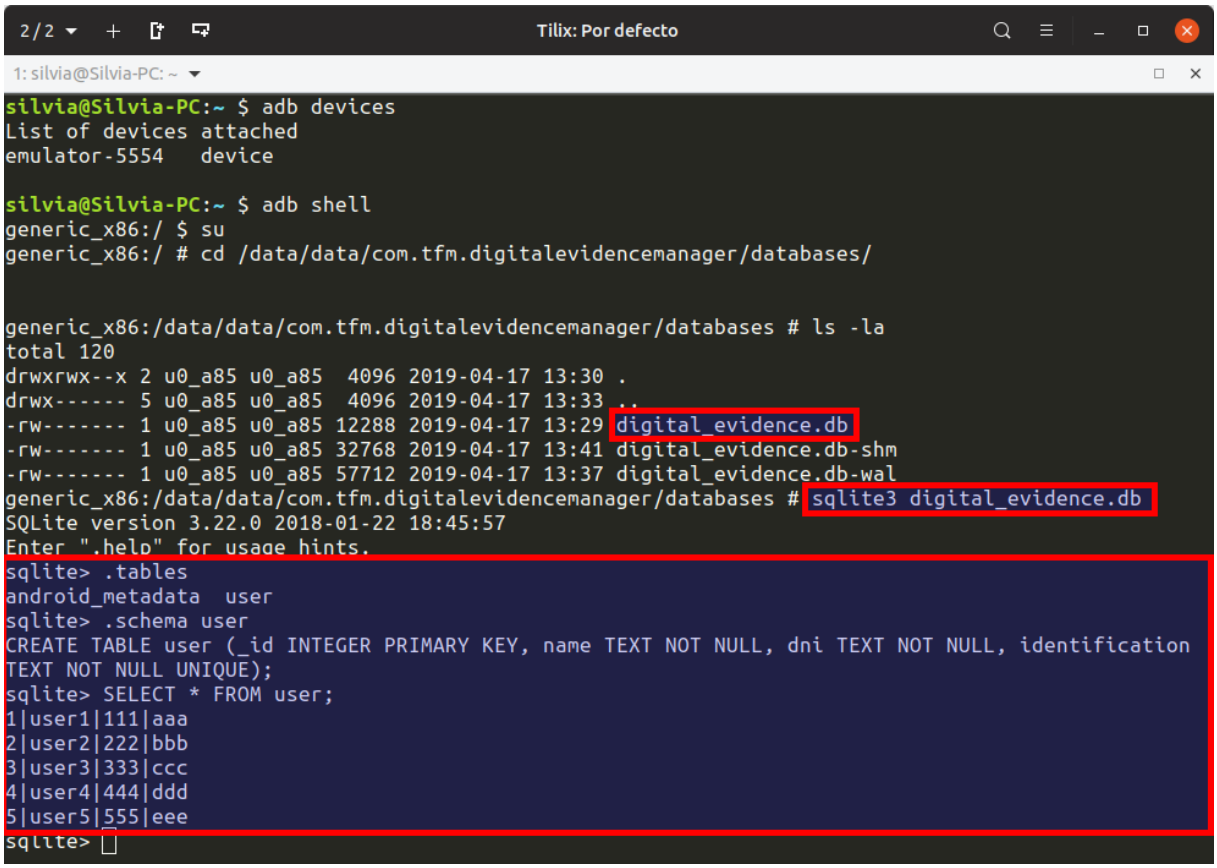


Figura 4.3: Acceso a la base de datos no cifrada y su contenido desde Ubuntu

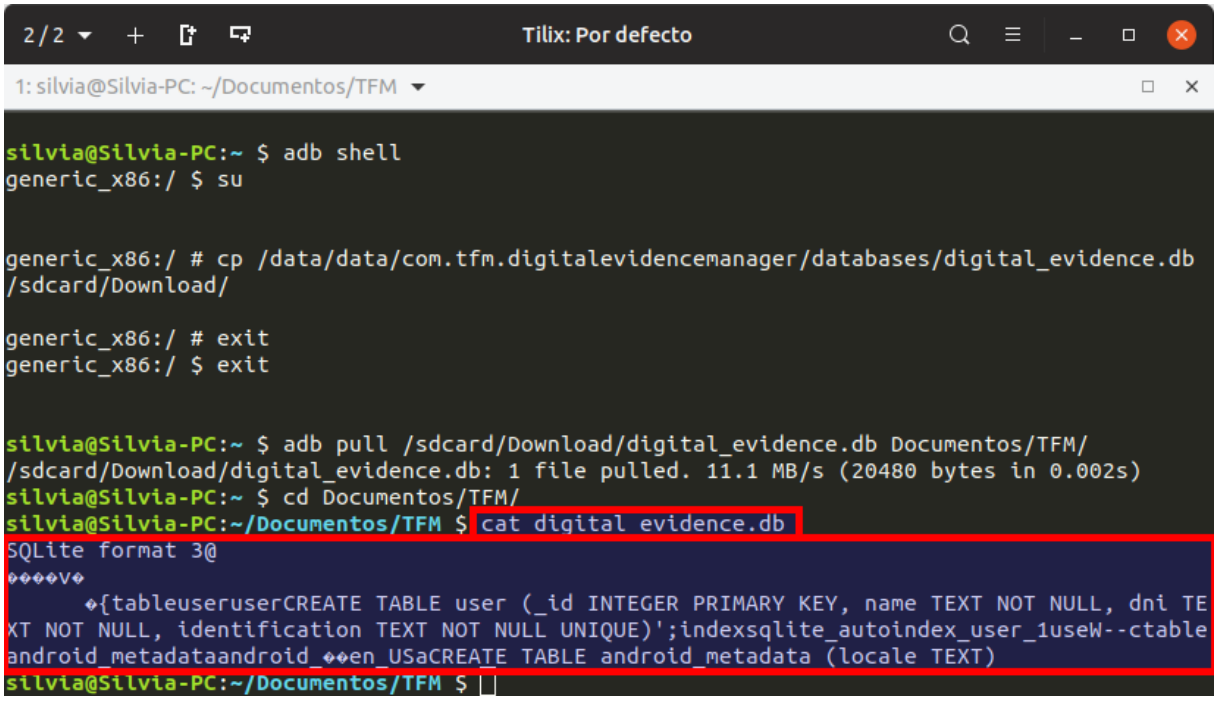


Figura 4.4: Contenido en texto plano de la base de datos no cifrada

Por tanto, se debe de buscar un mecanismo de cifrado que permita proveer de seguridad a la base de datos, para que, en caso de que el dispositivo quede comprometido y el archivo sea extraído, no se pueda acceder a su contenido.

Se han barajado dos opciones para cifrar la base de datos, *SQLite Encryption Extension* [33] y *SQLCipher* [34]. Se muestra a continuación en la Tabla 4.1 una comparativa de ambas opciones:

Tabla 4.1: Comparación de tipos de opciones para cifrar la base de datos

	SQLite Encryption Extension	**SQLCipher**
Ventajas	- Implementación oficial para SQLite - Cifra el archivo completo	- OpenSource - Fácil instalación en el proyecto - Cifra el archivo completo
Inconvenientes	- Licencia de pago - Accesos lentos - Mayor tamaño	- Accesos lentos - Mayor tamaño
Algoritmos de cifrado	- RC4 - AES-128 en modo OFB - AES-128 en modo CCM - AES-256 en modo OFB	- AES-256 en modo CBC (por defecto)

Una característica que comparten ambas opciones consideradas es el cifrado de todo el archivo de base de datos, pues otras opciones que no llegaron a considerarse consistían en cifrar filas de la base de datos, o cifrar los datos antes de guardarlos. Sin embargo, el esquema de base de datos continuaba siendo visible al abrir el archivo como texto plano. Otras características comunes entre ambas opciones es el mayor tiempo de acceso a los datos, pues en cada conexión es necesario descifrar el archivo, además de un mayor tamaño. Finalmente dado que *SQLCipher* protege los accesos a base de datos mediante una contraseña y es *OpenSource* ha sido escogido para la realización de este prototipo, pues se considera que estos motivos son suficientes para esta elección.

Se comprueba en la Figura 4.5 que utilizando esta librería no se puede acceder al contenido de la base de datos, ni siquiera a su esquema. Se puede ver también que el peso de la base de datos es mayor, aún cuando el contenido es el mismo. También se observa en la Figura 4.6 que ahora al intentar abrir el archivo de base de datos, tal como se hizo en la Figura 4.4, éste no es texto plano.

Se comprueba también que el archivo *APK* de la aplicación ha incrementado considerablemente su tamaño, unos 7MB fijos aproximadamente respecto al tamaño original. Este incremento de tamaño en el archivo *APK* final se debe a que *SQLCipher* utiliza su propia implementación de *SQLite*, y por tanto necesita añadir estas librerías para funcionar correctamente, como se puede ver en [35].

Se analizarán a continuación la diferencia de tiempo y tamaño entre la base de datos cifrada

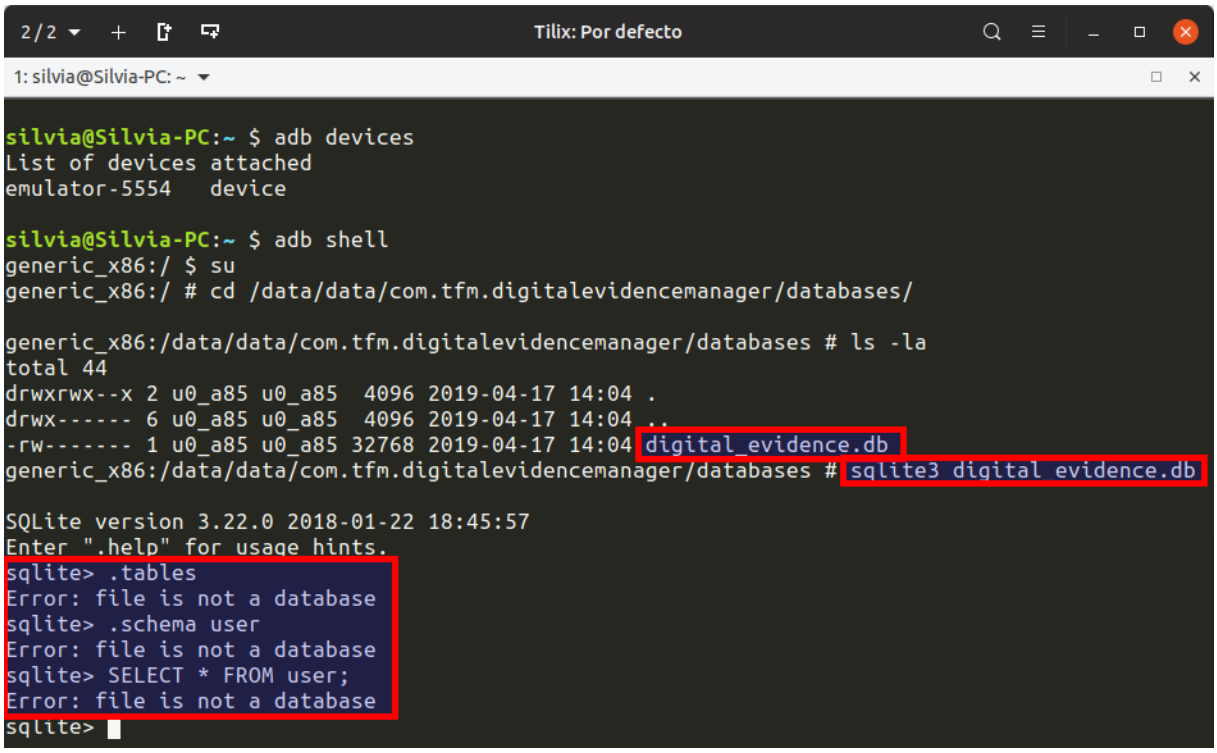


Figura 4.5: Acceso a la base de datos cifrada desde Ubuntu

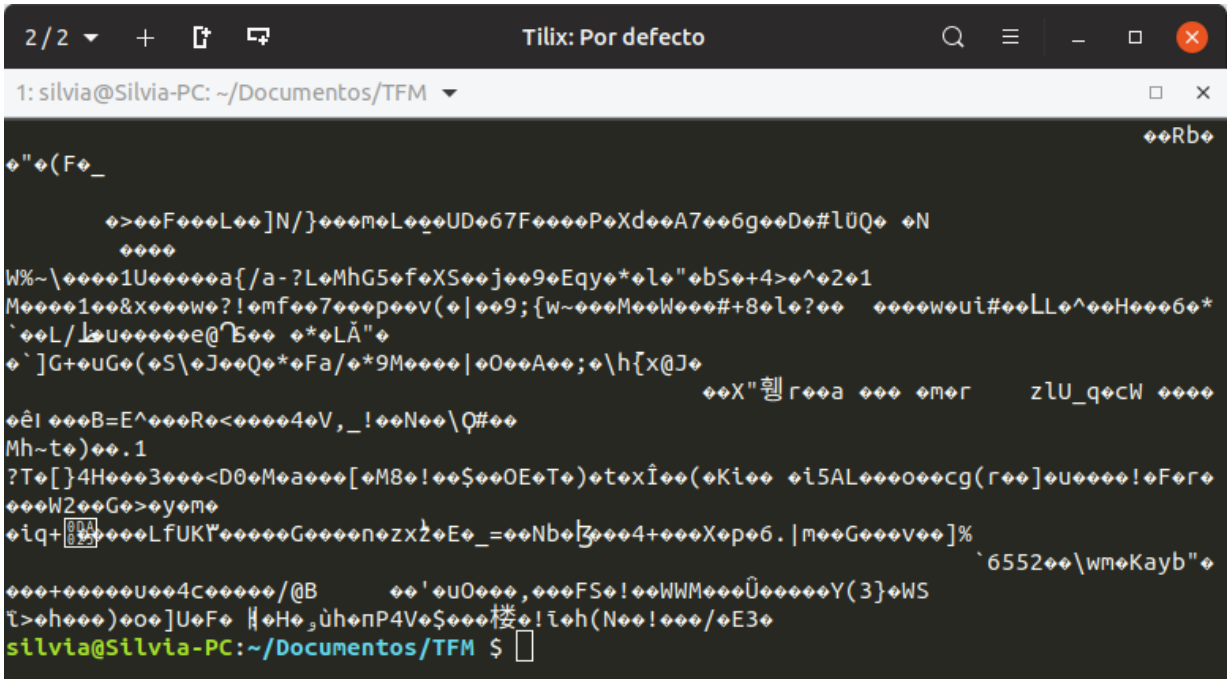


Figura 4.6: Contenido de la base de datos cifrada

y sin cifrar. Para ello se insertan un número cada vez mayor de elementos en la Tabla *user*, abriendo y cerrando una conexión en cada inserción. Para estas mediciones se utiliza el código 4.1, y se mostrarán los resultados obtenidos en la Tabla 4.2.

```
fun timeTest(value:Int){
  val timeElapsed = measureTimeMillis {
    for (i in 1..value){
      User(null, i.toString(), "user".plus(i), System.currentTimeMillis()).save(this)
    }
  }

  val dbFile = this.getDatabasePath(DATABASE_NAME)
  val fileSize = dbFile.length()

  Log.d("TEST", "---------- " + value + " elements ----------")
  Log.d("TEST", "Total rows: " + User.getAll(this).size)
  Log.d("TEST", "Total time: " + timeElapsed/1000 + "s" + timeElapsed%1000 + "ms")
  Log.d("TEST", "Total file size: " + fileSize/1024 + "K")
}
```

Código 4.1: Función utilizada para medir los tiempos de acceso a base de datos

Tabla 4.2: Tiempos de ejecución y tamaños de archivo con la base de datos cifrada y sin cifrar

	Base de datos sin cifrar		**Base de datos cifrada**	
Número de inserciones	**Tiempo total**	**Tamaño archivo**	**Tiempo total**	**Tamaño archivo**
10	214ms	20K	7s530ms	32K
100	1s221ms	20K	71s567ms	32K
1000	7s503ms	80K	722s516ms (∼12min)	92K
10000	78s674ms	600K	7566s300ms (∼2h10m)	624K

Los datos mostrados en la Tabla 4.2 se corresponden a ejecuciones realizadas en el emulador. Para medir los tiempos se han realizado las inserciones en 3 ejecuciones, y se ha calculado la media aritmética de estas. En cada inserción se ha realizado una conexión con la base de datos. Se concluye que al cifrar la base de datos se gana en seguridad, pero se pierde mucho rendimiento. En cuanto al tamaño de archivo, éste no varía entre ejecuciones, siendo la base de datos cifrada ligeramente mayor.

4.2.4. Almacenamiento seguro con control de acceso

El *secure element* se empleará con dos objetivos principales en la implementación de la solución propuesta: almacenar los datos biométricos del usuario y almacenar las claves de cifrado de las evidencias.

En cuanto al almacenaje de los datos biométricos, estos son almacenados por el sistema, el usuario no tiene ningún control sobre ellos desde el testigo digital. Esto se debe a que los datos biométricos del usuario fueron almacenados por el la aplicación gestora de los datos biométricos, por tanto el usuario no podrá acceder ni modificarlos desde la aplicación del

testigo digital, únicamente podrá utilizarlos para verificar su identidad. Esto es especialmente importante, porque en tanto a que se mantena la cadena de confianza en el inicio del dispositivo móvil, la verificación de la identidad del individuo recae en la arquitectura de seguridad del dispositivo móvil.

Respecto al almacenaje de claves, estas claves son utilizadas para el cifrado de la evidencia al almacenarla en el sistema de archivos. Estas claves se almacenarán mediante el sistema *Keystore*, del que se habló en el Capítulo 3.2.1.

4.3. Diseño de la aplicación de Usuario

Como se observa en la Figura 4.7, la aplicación que tendrá la labor de ser el gestor de operaciones entre el usuario y el dispositivo (Figura 4.2), se encuentra dividida en una serie de partes bien diferenciadas: por un lado estaría el *core* de la aplicación, y por otro las implementaciones propias de *Android*. A continuación se describen de las ventajas que ofrece ese diseño.

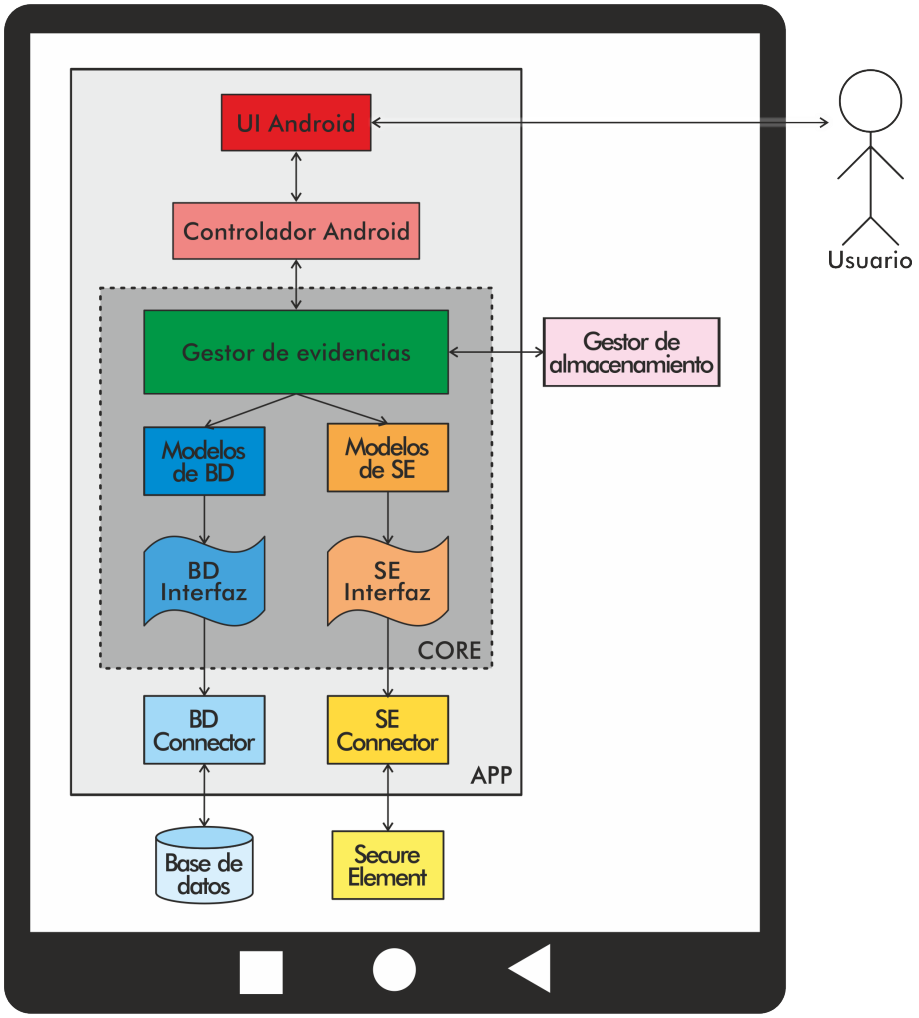


Figura 4.7: Diseño de la aplicación de usuario

4.3.1. Core de la aplicación

El *core* de la aplicación ha sido diseñado de forma que pueda ser fácilmente exportado a otras plataformas, es decir, en caso de querer desarrollar el testigo digital en otra plataforma, no sería necesario modificar el *core* de la aplicación, únicamente los conectores finales con dicha plataforma. Este *core* se encuentra dividido en una serie de capas: gestor de evidencias (4.3.1), modelos (4.3.1) e interfaces (4.3.1).

Gestor de evidencias

El gestor de evidencias es la capa que se encuentra a más alto nivel, en ella está contenida la funcionalidad principal de la aplicación. Esta capa será la encargada de comunicarse con el controlador de la plataforma *Android* y con el gestor de almacenamiento, utilizando para ello la funcionalidad aportada por los modelos. Por tanto, esta capa será completamente independiente de la plataforma en la que se ejecute, y no tendrá conocimiento del motor de base de datos o *secure element* utilizados.

Es en esta capa dónde se construirá, con ayuda de los modelos, la funcionalidad propia de la aplicación, es decir, es aquí donde se proporcionará al sistema la funcionalidad de adquirir una evidencia o transferir esta al servidor.

Modelos

A continuación, se encuentra la capa de modelado de los objetos necesario para el funcionamiento del gestor de evidencias, es decir, aquí se encontrará la capa correspondiente a lo que en programación orientada a objetos se denominan *objetos* y sus clases.

Esta capa estará formada por los diferentes *objetos* que forman la aplicación y los métodos representativos de cada uno, de los que se hablará más adelante en el apartado de prototipado.

Al igual que la capa del gestor de evidencias, esta capa no tendrá conocimiento del motor de base de datos utilizado, unicamente utilizará la funcionalidad aportada por las interfaces para interactuar con dicho motor.

Interfaces

Esta capa es la encargada de mantener la relación de métodos y funciones que serán necesarios para la comunicación con la base de datos y el *secure element* del dispositivo. Se trata solo de una relación de métodos con los tipos de los argumentos de entrada y salida de cada uno, no la implementación en si de estos métodos.

Por tanto, estas interfaces se hacen imprescindibles para poder llevar a cabo la portabilidad del *core* de la aplicación a otra plataforma, o incluso para mantener la plataforma, pero cambiar el motor de base de datos o el *secure element*. De esta forma, si se quiere implementar el testigo digital en Android, pero cambiar el motor de base de datos, unicamente sería necesario desarrollar el conector final con la base de datos implementando esta interfaz.

En las interfaces de base de datos se incluyen también los contratos de base de datos. Estos contratos son clases en las que se indican el nombre de la tabla y los campos de cada una

de estas, además de los enumerados utilizados en caso de haberlos. Además, estos contratos deben ser válidos independientemente del motor de base de datos y del sistema utilizado. Sirven, por tanto, de referencia a la hora de la creación de los modelos en la capa superior, y de referencia para el desarrollo de la estructura de la base de datos final.

4.3.2. Partes dependientes del sistema operativo

Se detalla a continuación los componentes que dependen intrínsecamente del sistema operativo sobre el que se desarrolle la solución. En este caso se está desarrollando el testigo digital en la plataforma Android, por lo que el diseño estará centrado en esta plataforma. Aquí se encuentra, por un lado, la interfaz de usuario y su controlador, y por otro, los conectores con la base de datos y con el *secure element*. Mientras que aquí se proporcionan detalles generales, se ofrecerá una visión más específica durante el prototipado, en el Capítulo 5.

Interfaz de usuario y Controlador Android

Por un lado, la interfaz de usuario será la encargada de interactuar con el usuario. Esta interfaz de usuario será la responsable de recoger los datos introducidos por el usuario y pasarlos al controlador, para después recoger los datos del controlador y mostrárselos al usuario. Por otro lado, el controlador Android será el encargado de recoger los datos proporcionados por la interfaz de usuario, y de interactuar con el gestor de evidencias, para finalmente mandar los resultados de este a la interfaz.

Conectores con base de datos y secure element

Estos conectores serán los encargados de la implementación final de la conexión con el motor de base de datos elegido y con el *secure element* concreto. Son la única parte de la herramienta que tienen conocimiento del motor de base de dato o del *secure element* utilizado. Estos conectores implementan las interfaces del core de la aplicación para realizar su función concreta.

Por ejemplo, en el caso de la base de datos, se detallarán aquí las sentencias SQL de creación y eliminación de tablas (utilizando para ellos los contratos de base de datos), así como las diferentes funciones definidas por la interfaz. Mientras que en el caso del *secure element* se encontrarán aquí los detalles concretos de la utilización de la librería *OMAPI* o del sistema *Keystore*.

En resumen, estos conectores son el nexo de unión entre el motor de base de datos o el *secure element* y la aplicación desarrollada, permitiendo así la portabilidad de la aplicación a otras plataformas, pues como ya ha sido comentado en el la sección de las interfaces, para permitir el uso de otro *secure element* o motor de base de datos unicamente se necesitaría desarrollar estos conectores.

4.4. Diseño del servidor web

El objetivo del servidor Web es unicamente la recepción de evidencias enviadas desde la aplicación de usuario, por lo que el diseño estará centrado en esta funcionalidad.

Como se observa en la Figura 4.8, el servidor se encuentra dividido en tres partes principales: gestor de evidencias, *API REST* e interfaz de usuario.

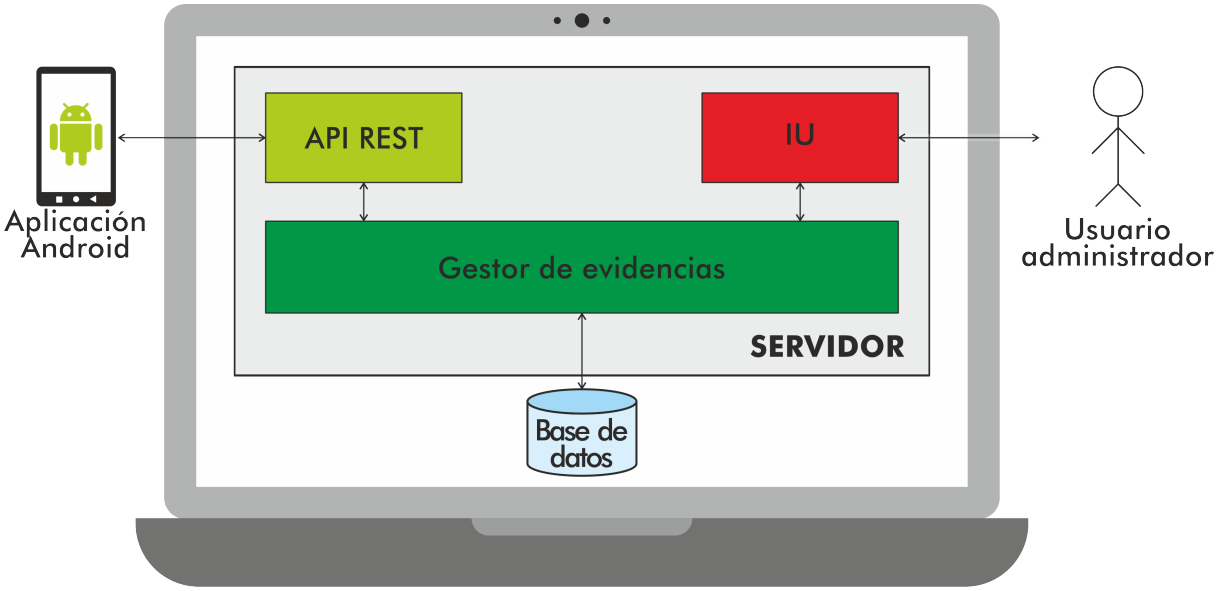


Figura 4.8: Diseño del servidor

Gestor de evidencias

En el caso del servidor Web, el gestor de evidencias se encarga de por un lado, proporcionar la funcionalidad necesaria a la *API REST* para la comunicación con la aplicación móvil, y por otro, se encarga de la comunicación de datos con la interfaz de usuario.

La funcionalidad de este gestor está limitada al propósito de la validación (Capítulo 6), centrándose en la recepción de la evidencia y generando un listado y detalle de dichas evidencias, además de permitir la correcta configuración de la aplicación.

API REST

La *API REST* será la encargada de exponer la funcionalidad del servidor web a la aplicación de usuario. Esta incluirá *endpoints* tanto de recepción de evidencias, como de configuración de la aplicación.

Interfaz de usuario

Al igual que en la aplicación Android, la interfaz de usuario será la encargada de interactuar con el usuario y recoger los datos introducidos por este para enviarlos al gestor de evidencias, además de recoger la respuesta recibida por el gestor de evidencias y mostrarla al usuario.

En este caso, el servidor contará con el acceso de un usuario administrador, siendo en esta primera versión de la aplicación el único usuario que tendrá acceso.

5. Prototipado del Testigo Digital

Las secciones anteriores explican las fases hasta llegar al diseño de la solución propuesta, que se detalla de forma general para comprender cómo se ha realizado la adaptación del testigo digital al entorno Android. Se espera que los pasos anteriores ayuden a exportar esta solución a otras plataformas considerando criterios similares. En esta sección se detallará el prototipo desarrollado en este TFM, por lo que, partiendo del diseño, se define una solución específica para la plataforma Android en su versión 9.0 y un dispositivo concreto, el Google Pixel 3. Por lo tanto esta sección está muy enfocada a preparar el prototipo para la adaptación del testigo digital en Android, que es uno de los objetivos propuestos. Se comenzará hablando de los casos de uso concretos de este prototipo, para continuar después con los detalles de su implementación.

Este prototipo cumple con los requisitos del testigo digital nombrados en el Capítulo 4.1, por tanto será una aplicación Android que permita la adquisición, almacenaje y transmisión de forma segura de las evidencias electrónicas a un servidor autorizado.

5.1. Casos de uso

Los casos de uso que cubre el prototipo de testigo digital desarrollado se muestran en la Figura 5.1. Como puede observarse, los casos de uso de la aplicación serán realizados por el usuario, mientras que los del servidor serán realizados por el sistema.

5.1.1. Casos de uso de la aplicación

Como se observa en la Figura 5.1, todos los casos de uso de la aplicación, salvo la autenticación, dependen de ésta para ser realizados, es decir, para realizar cualquier acción en la aplicación el usuario deberá autenticarse previamente.

Además, todos aquellos casos de uso que implican tener una evidencia en el sistema requieren que previamente se haya realizado alguna vez la acción de adquisición de una evidencia, aunque ésta no tiene que ser necesariamente en la misma sesión. Es decir, un usuario podría haber realizado la adquisición de una evidencia y varios días más tarde podrá consultar los detalles sobre esta, pero siempre que vaya a adquirir una evidencia, o consultar los detalles sobre esta, ha debido de autenticarse en el sistema.

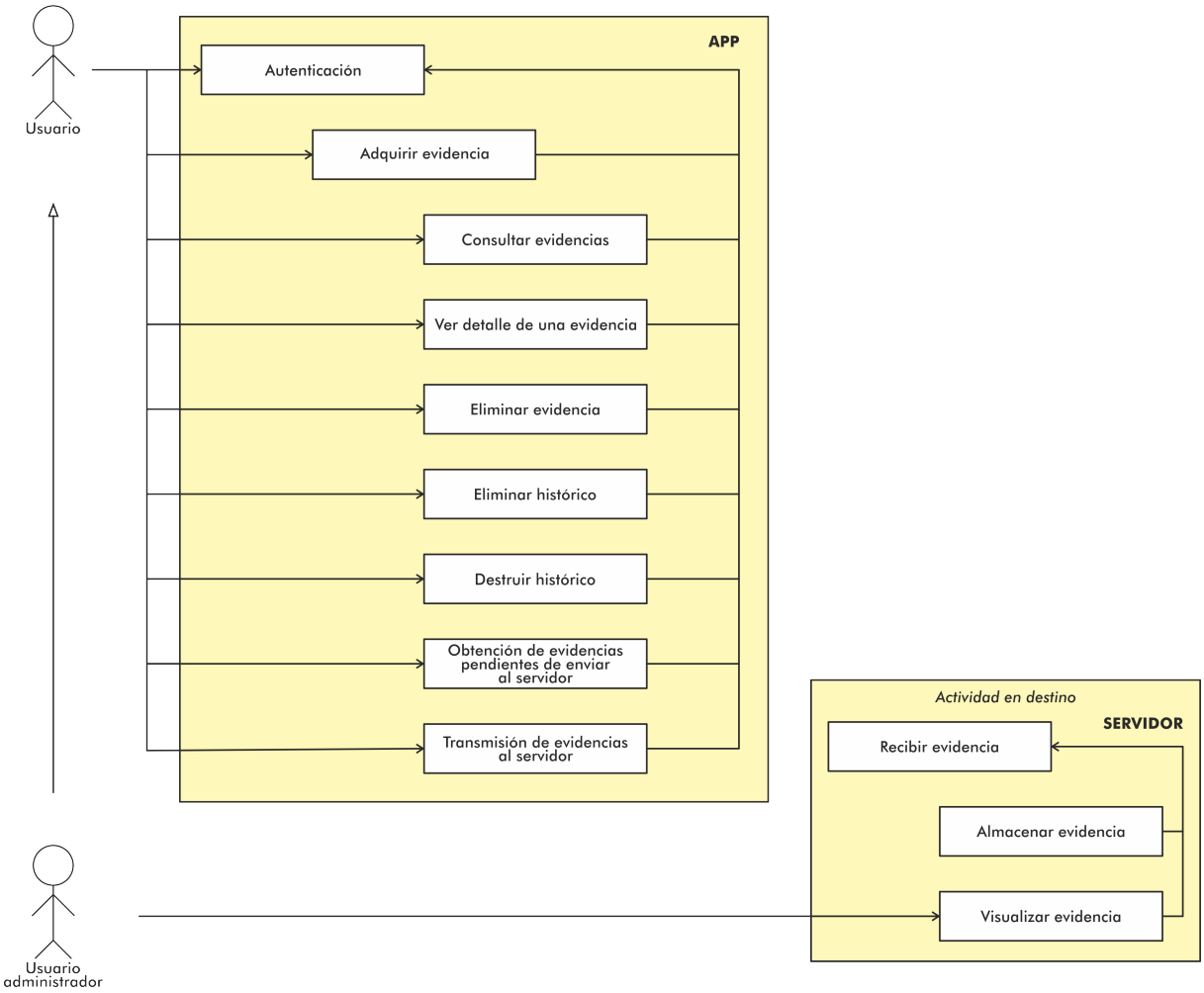


Figura 5.1: Diagrama de casos de uso de la herramienta

Autenticación

Se requiere que el usuario esté identificado biometricamente en la aplicación para su funcionamiento. De esta forma se asegura que el usuario que utiliza la aplicación es quien dice ser. Por tanto, al iniciar la aplicación se pedirá al usuario que se identifique mediante la huella antes de realizar cualquier otra acción. Al final de todos los casos de uso, en la Figura 5.5a se muestra la pantalla de autenticación de la aplicación.

Para realizar la autenticación biométrica en la aplicación se ha utilizado `BiometricalPrompt`, como se muestra en el segmento de código 5.1 [36], característica añadida en *Android Pie 9.0*. Esta característica construye un cuadro de diálogo solicitando la huella al usuario, quedando como se muestra en la Figura 5.2. En versiones anteriores se hubiera utilizado `fingerprint manager`, pero al estar centrados en dispositivos con *Android Pie 9.0* o superior, no es necesario que se mantenga la compatibilidad con versiones anteriores.

```kotlin
private fun showBiometricalPrompt(){
    val biometricPromptBuilder = BiometricPrompt.Builder(this)
    biometricPromptBuilder.setTitle(getString(R.string.biometric_prompt_title))
    biometricPromptBuilder.setSubtitle(getString(R.string.biometric_prompt_subtitle))
    biometricPromptBuilder.setDescription(
        getString(R.string.biometric_prompt_description)
    )
    biometricPromptBuilder.setNegativeButton(
        getString(R.string.biometric_prompt_negative_button),
        this.mainExecutor,
        DialogInterface.OnClickListener({ _, _->})
    )
    val biometricPrompt = biometricPromptBuilder.build()
    val authenticationCallback = this.getAuthenticationCallback()

    biometricPrompt.authenticate(
        CancellationSignal(),
        this.mainExecutor,
        authenticationCallback
    )
}
```

Código 5.1: Función de autenticación biométrica en la app

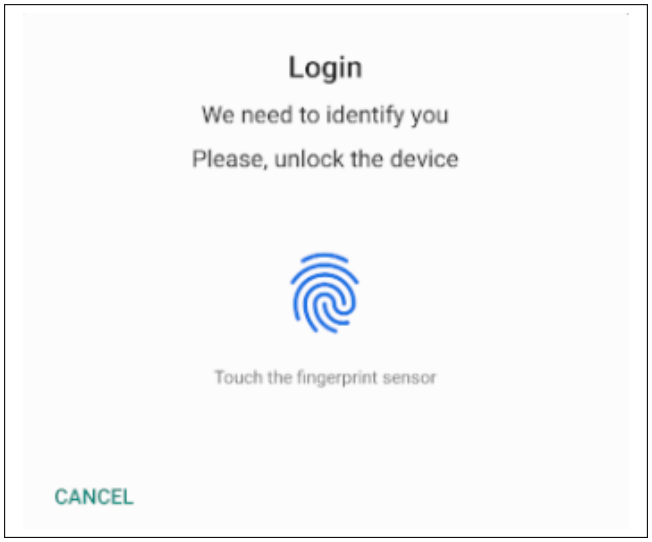


Figura 5.2: Diálogo para la autenticación por huella

Sin embargo, la utilización de la huella dactilar para la autenticación del usuario ha planteado ciertos problemas: Android ofrece la posibilidad de autenticar al usuario en la aplicación de esta manera, pero no permite su identificación. Por tanto, no se pueden crear distintos roles de acceso a la aplicación, solo se puede dar o revocar el acceso a la aplicación a todos los usuarios del dispositivo Android a la vez.

Adquirir evidencia

Mediante la adquisición de una nueva evidencia, el sistema muestra el gestor de archivos de Android y el usuario selecciona el archivo deseado, este se incorpora al listado de evidencias almacenadas por la aplicación. Durante este proceso, guardaremos la evidencia cifrada con

cifrado *AES* en la carpeta privada de la aplicación, la clave de cifrado de esta en el *SE* y un conjunto de datos relacionados con la evidencia, además de su hash *MD5*, en base de datos. En una primera versión de la aplicación unicamente se permite la adquisición de imágenes y de una en una. Lo ideal hubiera sido almacenar también el hash *MD5* de la evidencia en el *SE*, pero debido a ciertos problemas con este aspecto esto no ha sido posible, sin embargo, el acceso al *SE* para realizar este almacenamiento será detallado en secciones posteriores.

Este proceso de adquisición de evidencia puede resumirse en el diagrama de flujo de la Figura 5.3.

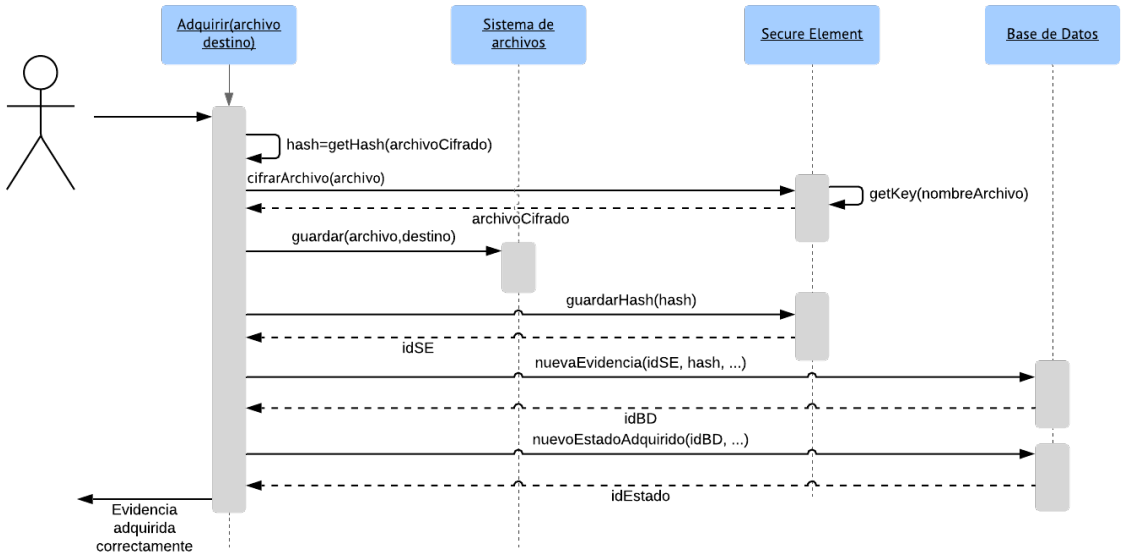


Figura 5.3: Diagrama de flujo de la adquisición de una evidencia

La evidencia adquirida es almacenada en la carpeta privada de la aplicación de forma cifrada. Para el cifrado de archivos se utiliza la función definida en el código 5.2. Como se puede observar aquí, tanto el algoritmo como la *suite* de cifrado son configurables en un archivo externo de constantes. En este caso se ha utilizado *AES* tanto para el cifrado como para la *suite* de cifrado. También se observa aquí la utilización de una extensión propia para almacenar las evidencias en el sistema de archivos.

```kotlin
fun encryptFile(file: File): File? {
    val key = Local.secret_key
    val encryptedFile = File(file.absolutePath + Constants.EVIDENCE_EXTENSION)
    val inputStream = FileInputStream(file)
    val outputStream = FileOutputStream(encryptedFile)

    try {
        val secretKey = SecretKeySpec(key.toByteArray(), Constants.CIPHER_ALGORITHM)
        val cipher = Cipher.getInstance(Constants.CIPHER_SUITE)
        cipher.init(Cipher.ENCRYPT_MODE, secretKey)

        val inputBytes = ByteArray(file.length().toInt())
        inputStream.read(inputBytes)
        val outputBytes = cipher.doFinal(inputBytes)
        outputStream.write(outputBytes)

    } catch (e: GeneralSecurityException) {
        throw CryptoException("Error in encrypt file '" + file.name + "': " + e.message)

    } catch (e: IOException) {
        throw CryptoException("Error with I/O while encrypting: " + e.message)

    } finally {
        inputStream.close()
        outputStream.close()
    }

    return encryptedFile
}
```

Código 5.2: Cifrado de un archivo

Para almacenar esta evidencia en la carpeta privada de la aplicación, hay que tener en cuenta que, al estar trabajando con versiones de Android mayores que *Android 6.0 Marshmallow* se necesita pedir expresamente los permisos al usuario [37]. Se requiere del permiso `android.permission.WRITE_EXTERNAL_STORAGE` para poder almacenar en dicha carpeta, el cuál está calificado como *permiso peligroso*, pues con dicho permiso se puede acceder a información sensible del usuario [38]. Por tanto, hay tener en cuenta que debe hacer la aplicación si estos permisos son denegados por el usuario, en este caso, sin dichos permisos no se podrá obtener la evidencia.

Una vez cifrada la evidencia y guardada en la carpeta privada del usuario, se procedería a almacenar el *hash* de la evidencia sin cifrar en el *SE*. Tal y como nombramos anteriormente, este almacenamiento no ha sido posible finalmente. Sin embargo, los detalles técnicos de conexión con el *SE* se encuentran detallados en el Capítulo 4.2.4. Al igual que con el cifrado, se observa en el código 5.3 que el tipo de función hash utilizada es fácilmente modificable desde el archivo de constantes. En este caso, se utiliza hash *MD5*, pues aunque éste se encuentre roto no se utiliza para almacenar ninguna contraseña, sino únicamente para comprobar que el archivo no ha sido modificado. Además como se comprueba en la Tabla 5.1, la función hash *MD5* tiene menor tamaño de salida, por lo que es la más adecuada para el propósito de este proyecto al contar con un almacenamiento muy limitado en el *SE*. Para calcular la función hash del archivo se utiliza únicamente el contenido del archivo, no se considera ni el nombre ni los metadatos.

```kotlin
fun getHashFromFile(file: File): String? {
    var md = MessageDigest.getInstance(Constants.HASH_TYPE)
    md.update(file.readBytes())
    val hash = md.digest().toTypedArray()

    return byteArrayToHexString(hash)
}
```

Código 5.3: Obtención de hash de un archivo

Tabla 5.1: Comparación de tipos de hash

Función hash	Tamaño de salida	Actualmente roto
MD5	128 bits	Si
SHA-1	160 bits	Si
SHA-256	256 bits	No

Después de almacenar el hash, se crea la entrada de la evidencia en la base de datos. Se almacena tanto la información de la evidencia, como el usuario y dispositivo que realizaron la adquisición. Se almacena también los metadatos y el hash de la imagen adquirida.

Dados los problemas de identificación de usuario descritos en el Capítulo 5.1.1, durante la adquisición de la evidencia se utilizará un usuario genérico como responsable de esta adquisición.

Consultar evidencias

Mediante esta opción el usuario consulta una lista con todas las evidencias almacenadas en el dispositivo. En una primera versión de la aplicación estas aparecerán ordenadas descendentemente por fecha de adquisición y no se permitirá el filtrado o la ordenación de evidencias.

De cada evidencia se verá el nombre dado por el usuario, su identificación única, la fecha de adquisición y la extensión del archivo. Puede observarse un ejemplo de listado de evidencias en la Figura 5.5b

Ver detalle de una evidencia

Con esta opción, el usuario consulta toda la información almacenada acerca de una evidencia. También se le muestra al usuario toda la información relativa a los diferentes estados por los que ha pasado. Un ejemplo de este detalle de evidencia se puede encontrar en la Figura 5.5c.

Eliminar evidencia

Cuando el usuario selecciona esta acción, el sistema libera espacio de la memoria del dispositivo. Esta acción eliminará la evidencia del sistema de archivos, pero no de base de

datos. Además, la realización de esta acción genera un nuevo estado para la evidencia, de manera que se podrá seguir la trazabilidad de la evidencia en caso de que esta haya sido eliminada.

En una primera versión de la aplicación esta opción se aplica a una sola evidencia cada vez, desde la vista de detalle de la evidencia. Para realizar esta acción el sistema pide confirmación al usuario.

Eliminar histórico

Al seleccionar esta opción, el sistema elimina de base de datos todas aquellas evidencias que ya han sido enviadas al servidor, y que están en estado que permite su borrado, esto es, enviadas completamente al servidor y que el servidor ha respondido que se puede eliminar. Si las evidencia no habías sido borradas del sistema de archivos, las elimina también. Idealmente, también se deberían de eliminar los hashes de estas evidencias del *SE*. Para realizar esta acción el sistema pide confirmación al usuario.

Esta acción es una acción segura, es decir, en caso de que alguna evidencia no se encuentre completamente enviada al servidor, esta no será eliminada, por tanto, no se pide doble confirmación ni nueva verificación de usuario.

Destruir histórico

Mediante esta opción el sistema elimina todo el contenido de la base de datos y del sistema de archivos. Se trata de una opción peligrosa, pues elimina todos los datos de la aplicación y es una acción irreversible, es por ello que para poder realizar esta acción, el sistema pide confirmación al usuario y una nueva identificación biométrica para así comprobar la identidad del usuario.

El proceso seguido por esta acción puede verse en el diagrama de flujo de la Figura 5.4.

Inicialmente, se pedirá confirmación al usuario y una nueva autenticación biométrica. A continuación, se enviarán al servidor todas las evidencias pendientes de envío, para posteriormente, adquirir la base de datos como una nueva evidencia, y enviarla también al servidor. Una vez que este proceso ha sido realizado con éxito, se eliminan los archivos del gestor de almacenamiento, y finalmente se elimina también el archivo de la base de datos. Idealmente, también se deberían de eliminar los hashes de estas evidencias del *SE*.

Por tanto, tras esta acción no habrá nada en la base de datos ni en el almacenamiento del dispositivo.

Obtención de evidencias pendientes de enviar al servidor

Cuando el usuario selecciona esta opción, el sistema comprueba si quedan evidencias pendientes enviar al servidor. Esta opción no requiere de conexión a Internet, pues en una primera versión de la aplicación no se contempla la transimisión de la evidencia entre dispositivos y por tanto toda la información relativa a la evidencia se encuentra en el dispositivo actual.

Con esta opción no se envían datos, únicamente se comprueba si quedan evidencias pendientes de sincronización con el servidor. Para ello, se comprueba que el estado más reciente

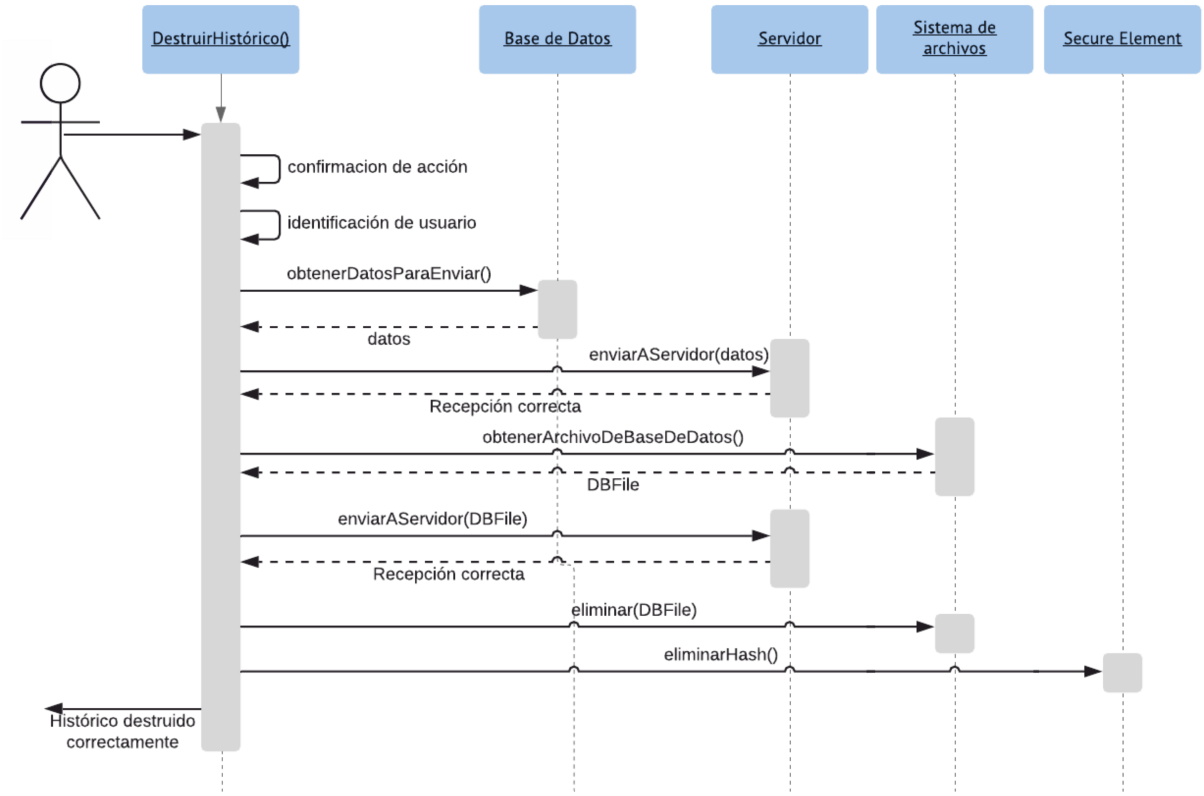


Figura 5.4: Diagrama de flujo de la destrucción del histórico

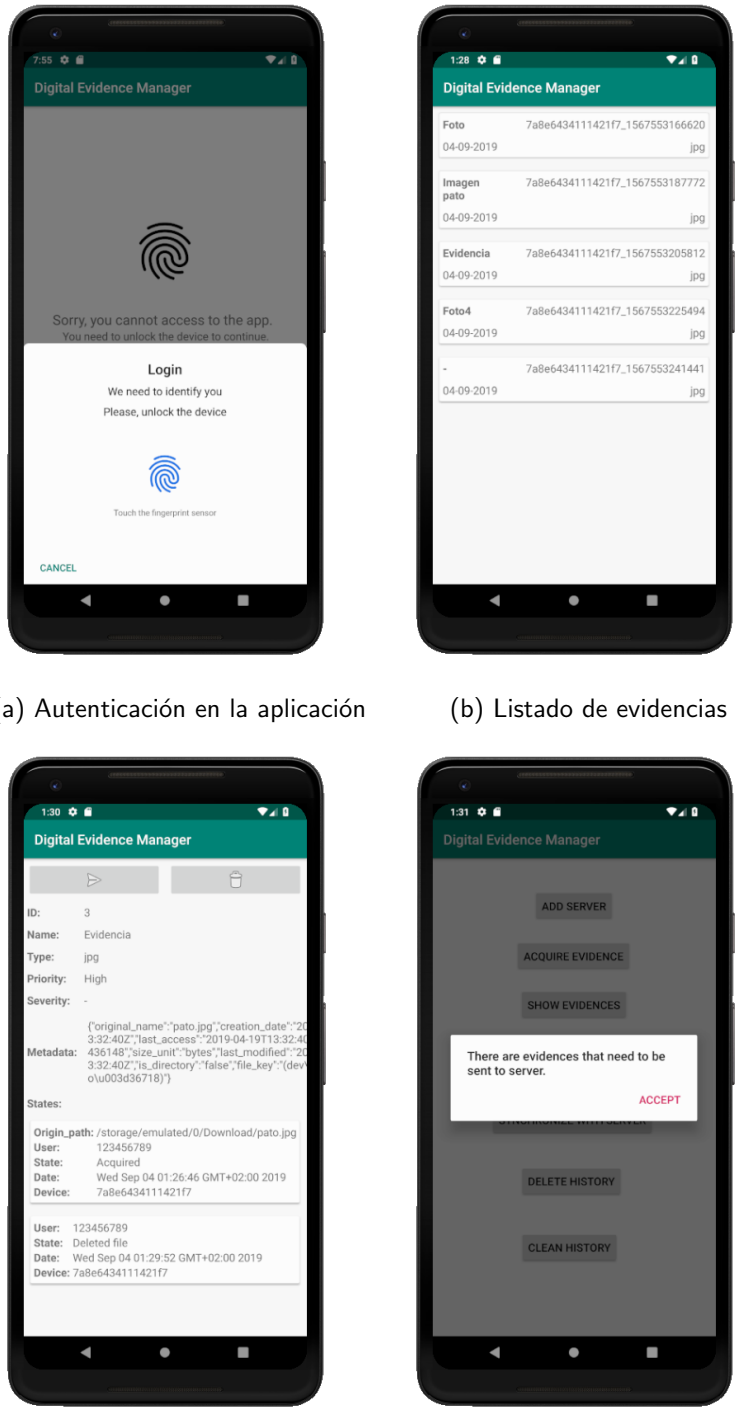
de todas las evidencias es el de *enviado al servidor* y que el servidor ha respondido que todas ellas se pueden eliminar.

Además, en esta primera versión de la aplicación, en caso de tener evidencias pendientes de enviar al servidor, esta opción no especifica cuales son estas evidencias, unicamente indica si tenemos evidencias pendientes o no, como puede observarse en la Figura 5.5d.

Transmisión de evidencias al servidor

Al transmitir evidencias al servidor, el sistema envía a este todas aquellas evidencias que la opción *obtención de evidencias pendientes de enviar al servidor* ha considerado que no están completamente enviadas. Para ello, primero se firma cada evidencia individualmente y posteriormente se envía al servidor. En una primera versión de la aplicación el usuario podrá elegir enviar todas las evidencias pendientes al servidor, o de una a una desde la vista detalle de la evidencia.

Una buena opción para firmar la evidencia hubiera sido utilizar la la clave privada almacenada en el DNI del usuario, utilizando en dicho caso la tecnología *NFC* del dispositivo para realizar la lectura del DNI. Sin embargo, este aspecto ha quedado en pequeñas pruebas de concepto que no han podido ser integradas finalmente en la aplicación debido a problemas de tiempo.



(a) Autenticación en la aplicación

(b) Listado de evidencias

(c) Detalle de la evidencia

(d) Comprobación de evidencias pendientes de envío al servidor

Figura 5.5: Capturas de pantalla de la aplicación

5.1.2. Casos de uso del servidor

A continuación se hablará de los casos de uso del servidor. Cómo ya se mencionó anteriormente, la funcionalidad del servidor es muy limitada en este prototipo. En este caso, el usuario visualizará la evidencia, y el resto de acciones serán realizadas por el sistema.

Recepción de evidencias

El sistema recibe las evidencias enviadas por la aplicación de usuario, y las almacena en base de datos. Esta recepción de evidencias se producirá a través de una API REST que será detallada en los apartados siguientes. La evidencia se recibe cifrada, y por tanto, se almacenará cifrada. En este caso, tanto el archivo de la evidencia como sus datos se almacenan en base de datos.

Visualización de evidencias

Al igual que en la aplicación móvil, el usuario podrá visualizar un listado con las evidencias almacenadas en el servidor. En este listado se mostrará unicamente el identificador de la evidencia.

Este listado puede observarse en la Figura 5.6a.

Ver detalle de una evidencia

Así mismo, el usuario podrá visualizar el detalle de una evidencia concreta. De esta forma, el usuario verá toda la información almacenada acerca de la evidencia.

5.2. Desarrollo de la aplicación de Usuario

La aplicación de usuario ha sido desarollada utilizando *Kotlin* como lenguaje. Se ha escogido este lenguaje pues es, junto a *Java*, lenguaje oficial en *Android* [39]. Además, pequeños ejemplos realizados anteriormente con dicho lenguaje ponían de manifiesto que la curva de aprendizaje era pequeña, así como sus ventajas durante el desarrollo, por ejemplo, se puede escribir funcionalmente el mismo código que en *Java*, pero utilizando mucho menos código.

Además, la aplicación se ha desarrollado en dos idiomas diferentes: español e inglés, siendo el inglés e idioma por defecto de la aplicación. En este caso, la aplicación toma el idioma del dispositivo, por lo que para cambiar el idioma de la aplicación será necesario realizar este cambio de idioma en el dispositivo.

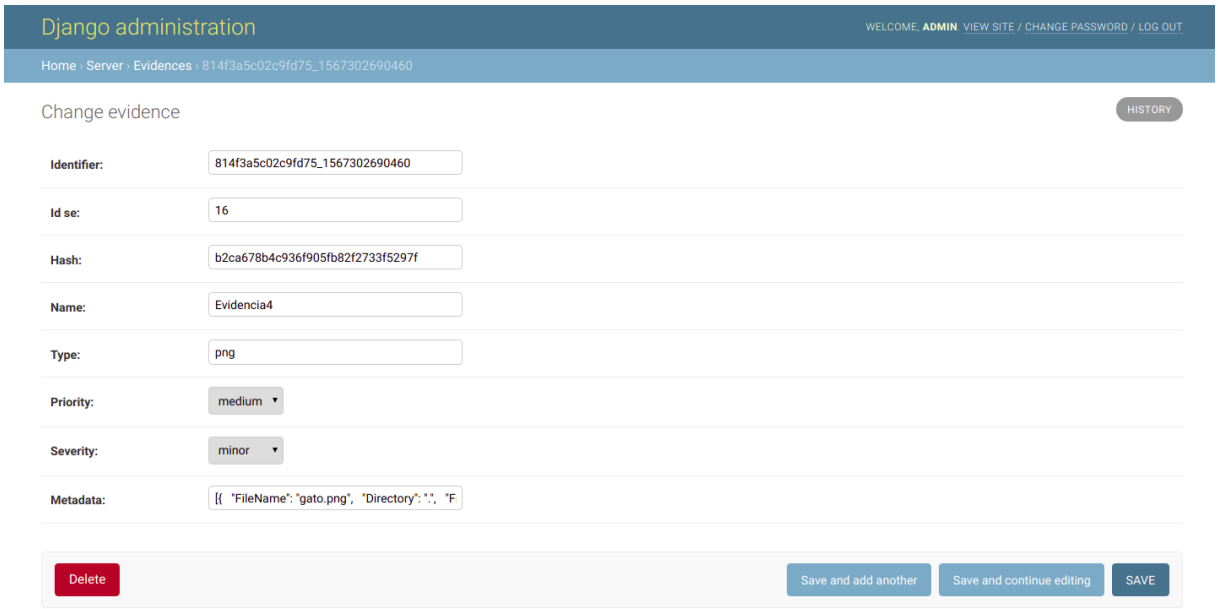
5.2.1. Desarrollo de la aplicación

A continuación se detallará la implementación de la aplicación de usuario desarrollada como prototipo. En el diagrama de clases mostrado en la Figura 5.7 se comprueba que para este desarrollo se han utilizado 4 clases: *evidencia*, *usuario*, *dispositivo* y *estado*. Cada una de estas clases representa un objeto de este sistema, y contiene las propiedades y funciones propias de cada uno.

Evidencia

Esta clase representa una **evidencia**. De cada evidencia se almacenan el identificador del *hash* del archivo almacenado en el *secure element*, dicho hash, el tipo de archivo, su prioridad,

(a) Listado de evidencias



(b) Detalle de la evidencia

Figura 5.6: Capturas de pantalla del servidor

su criticidad y todos los metadatos disponibles. Se guarda además un nombre dado por el usuario que permita la identificación de la evidencia por éste, este nombre no es modificable una vez registrado. Ninguno de estos campos es modificable tras la adquisición de la evidencia.

Se guarda también un campo *unique_id*, que es el *UUID* del dispositivo y el timestamp de la adquisicion separadas por un guión. Este *unique_id* sería un identificado único de la evidencia en cualquier base de datos, pues el *UUID* del dispositivo es único, y no se puede adquirir dos evidencias justo a la vez por el mismo dispositivo. No se usan los identificadores de base de datos para identificar la evidencia entre distintas bases de datos pues este identificador se asigna automáticamente, no se tiene control sobre esta asignación, de manera que un id de base de datos de una evidencia en la aplicación puede corresponder con otra evidencia en el

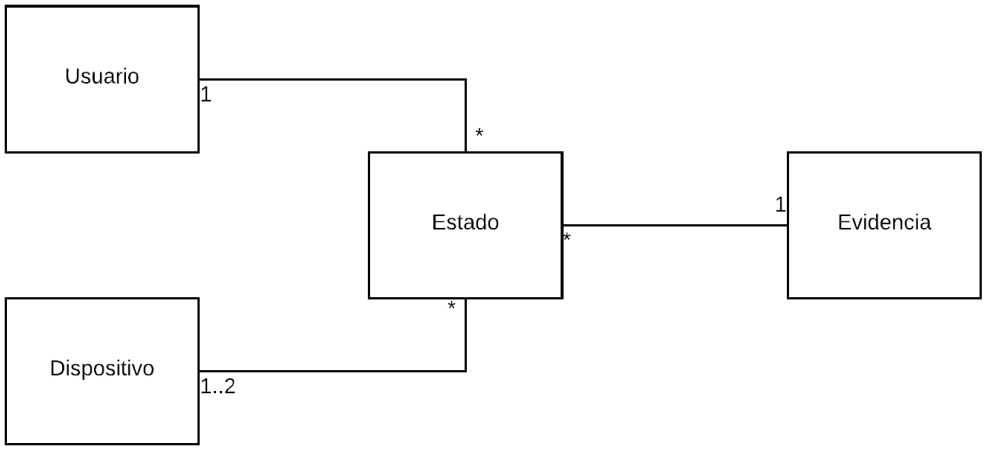


Figura 5.7: Diagrama de clases de la aplicación

servidor, por recibir este evidencias de distintos dispositivos.

Usuario

Esta clase representa un **usuario** de la aplicación. De cada usuario se almacena su nombre, DNI e identificador único de usuario.

En este prototipo no se tendrá acceso a la gestión de usuarios, es decir, una vez creado un usuario no será modificable, y sólo se podrá eliminar mediante la opción *destruir histórico*, opción que elimina toda la base de datos y su contenido, además de todas las evidencias almacenadas en el gestor de almacenamiento.

Dispositivo

Esta clase simula un **dispositivo**. Este dispositivo puede representar a un servidor con el que se sincronizará la aplicación, o a un dispositivo Android al que enviar o recibir evidencias. De este dispositivo almacenaremos su *UUID* único del dispositivo, un valor booleano indicando si es servidor o no, y un valor de conexión. Por ejemplo, en caso de un servidor, este valor de conexión sería su dirección ip, mientras que en el caso de ser otro dispositivo Android, este podría ser una dirección MAC.

Al igual que en el caso del usuario, en este prototipo no se permitirá la gestión de dispositivos desde la aplicación.

Estado

Representa cada uno de los **estados** de una evidencia. Cada evidencia tiene asociados una serie de estados. Estos estados serán inmutables, es decir, una vez creados no serán modificables. Con estos estados se seguirá la traza de lo ocurrido a la evidencia desde su

adquisición hasta que la base de datos sea eliminada. Cada estado está asociado con una evidencia, y con el usuario y dispositivos en los que se generó dicho estado, además de una fecha. Además de estos campos comunes, cada estado tiene una serie de campos únicos. Los posibles estados que puede tener una evidencia son los siguientes:

- **Adquirido:** Este estado representa la adquisición de una evidencia. Además de los campos comunes, este estado tendrá la ruta de origen de la evidencia. Cada evidencia tiene asociado un solo estado adquirido.
- **Archivo eliminado:** Este estado representa la eliminación del archivo del sistema de archivos. Por ejemplo, por haber realizado la adquisición de un archivo erróneo y posteriormente eliminarlo. Sin embargo la aplicación mantendrá una traza de que se adquirió una evidencia y luego se eliminó, y esta traza sería enviada igualmente al servidor.
- **Enviado:** Este estado representa un envío de la evidencia a otro dispositivo que no sea servidor. Además de los campos comunes a todos los estados, se guardará el identificador del dispositivo receptor de la evidencia. En este caso, la evidencia sería enviada mediante NFC a un dispositivo de características similares, pero de igual o mayor nivel en la cadena de custodia. Este envío queda fuera del alcance de este proyecto, pero dejamos este estado en base de datos por dejar dicha base de datos preparada para futuras ampliaciones del proyecto.
- **Enviado al servidor:** Este estado representa un envío de la evidencia a un servidor. Además de los campos comunes, se almacena el identificador del servidor y el valor devuelto por el servidor indicando si se puede o no eliminar la evidencia. Se guarda el identificador del dispositivo pues se podrían tener diversos servidores y necesitaremos saber a cual se está conectando el dispositivo. Tanto para este envío como para el envío entre dispositivos, se creará un objeto de tipo dispositivo en nuestro sistema, o se utilizará uno ya existente. En esta prueba de concepto únicamente se va a conectar con un servidor, pero este estado está preparado para el envío a múltiples servidores.

Además de estas clases se hablará también acerca de la implementación del diseño de la aplicación de usuario mostrado en el Capítulo 4.3, detallando unicamente la implementación del core de la aplicación, y de los conectores, tal y como se muestra en la Figura 5.8. No incluimos aquí el detalle del controlador Android ni de la interfaz, pues no es relevante para el prototipo del testigo digital.

Desarrollo de los modelos

Aquí se encuentran desarrollados los modelos con los que se comunicará el gestor de evidencias. Cada uno de estos modelos esta formado por una serie de parámetros, y una serie de métodos. En general, todos los modelos tienen en común métodos para guardar el objeto en base de datos, obtener un objeto de base de datos dado un identificado, u obtener un listado de objetos. Además, cada uno de estos modelos tendrá métodos propios, necesarios en

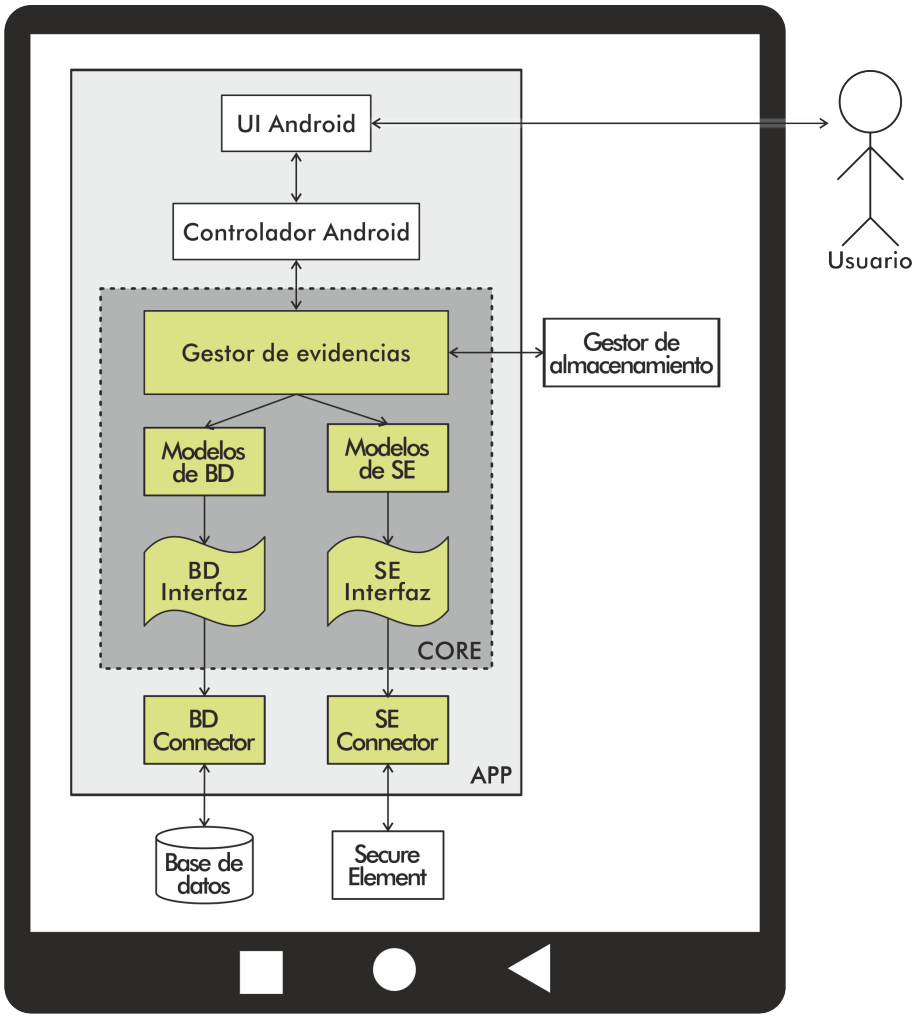


Figura 5.8: En color las partes detalladas de la aplicación

función del objeto que representen. Los distintos modelos implementados son los que se han detallado en el Capítulo 5.2.1.

Por ejemplo, una versión simplificada del modelo *Device* y su función *save* esta representado en el segmento de código 5.4. Esta función es la encargada de guardar el objeto en base de datos.

```
class Device(var id: Long?, val uuid: String, val isServer: Boolean) {
    fun save(context: Context) {
        val db = DeviceDB(context)
        var values = ContentValues()
        values.put(DeviceEntry.COLUMN_NAME_UUID, this.uuid)
        values.put(DeviceEntry.COLUMN_NAME_IS_SERVER, this.isServer)

        this.id = db.insert(values)
        if (this.id!! < 0) {
            throw ErrorSavingModel("Model ${this.javaClass.simpleName}
            not saved properly with this data [${values}]")
        }
    }
    ...
}
```

Código 5.4: Segmento del modelo *Device*

Aquí se observa como el constructor de la clase recibe obligatoriamente el *uuid* del dispositivo y un valor indicando si es servidor o no, mientras que el *id* es opcional. También se observa que estos tres parámetros tienen *getters*, pero solo el *id* tiene *setter*, pues este objeto será creado con todos los campos salvo el *id*, y este será asignado a posteriori tras ser almacenado en base de datos.

Esta ha sido una de las grandes ventajas que ha aportado *Kotlin* a este proyecto, pues en sólo una línea se tiene definido lo que en *Java* serían algunas más. En este caso, el símbolo de interrogación indica que ese parámetro puede ser `null`. Además, la utilización de los modificadores `var` y `val` hacen que dicho parámetro sea mutable o inmutable respectivamente, lo que hace que en este último caso el objeto solo cuente con métodos *getters* y no se permitan valores `null`. Como podemos observar, en este caso se ha utilizado el constructor primario, que asigna automáticamente los valores a los parámetros. En caso de ser necesario pueden utilizarse constructores al estilo *Java* que permitan realizar más acciones, como se ha utilizado en la clase *estado*.

Además, el método *save* recibe el contexto de la aplicación, pues este es necesario en la plataforma Android para las conexiones con la base de datos. El paso de este contexto desde el controlador Android hasta las clases conectores con la base de datos y con el *secure element* ha hecho que este prototipo no sea del todo portable a otras plataformas, tal y como se había descrito en el Capítulo 4.3, sin embargo, sigue siendo portable al cambio de motor de base de datos y de *secure element*. En este método *save*, se guardan los parámetros del objeto en un diccionario de tipo `ContentValues`, y se pasa al conector para que este realice la inserción final en la base de datos. El resultado de esta inserción será el id del objeto.

En caso de que el objeto no pueda ser insertado en base de datos, el id devuelto tendrá valor -1, por lo que en ese caso se lanza una excepción, que será capturada en el gestor de evidencias electrónicas para mostrar un mensaje al usuario indicando que la acción no ha podido ser completada con éxito. Además, con el doble signo de exclamación se indica que este valor nunca va a ser `null`, y en caso de serlo lanzaría otra excepción.

Todos los modelos han sido construidos siguiendo una lógica similar.

Desarrollo de las interfaces

En esta capa se encuentran desarrolladas las interfaces y los contratos de base de datos. En estas interfaces se encuentra la definición de todos los métodos necesarios por los modelos. En el segmento de código 5.5 se muestra la interfaz del objeto *device* a modo de ejemplo.

```
interface DeviceInterface {
    fun insert(values : ContentValues) : Long

    @Throws(SomeRowsReturned::class)
    fun getByUUID(uuid : String) : ContentValues?

    @Throws(SomeRowsReturned::class)
    fun getByID(id : Long) : ContentValues?
}
```

Código 5.5: Interfaz del modelo *Device*

De forma general, en estas interfaces se encuentran métodos para insertar en base de datos, y obtener un objeto según diferentes identificadores. Además, cada modelo tiene definidos sus propios métodos.

Además, se encuentran aquí también los contratos de base de datos, de los que se habló en el Capítulo 4.3.1. Como ejemplo se muestra el contrato simplificado del objeto *device* en el segmento de código 5.6.

```
object DeviceContract {
    object DeviceEntry : BaseColumns {
        const val TABLE_NAME = "device"
        const val COLUMN_NAME_ID = BaseColumns._ID
        const val COLUMN_NAME_UUID = "uuid"
        const val COLUMN_NAME_IS_SERVER = "is_server"
    }
}
```

Código 5.6: Contrato simplificado del modelo *Device*

En este contrato se observa la definición del nombre de la tabla, y de los diferentes campos que tendrá la base de datos.

Desarrollo de los conectores

Esta capa se centra en la conexión con la base de datos *SQLite* cifrada que se mencionó en la Sección 4.2.3 y en el *secure element Titán M* del que se habló en la Sección 3.2.2. Estos conectores implementan las interfaces mencionadas en el apartado anterior, aportando los detalles propios de la base de datos y *secure element* utilizados.

En el caso de la base de datos, se muestra a modo de ejemplo el conector con la Tabla *device* en el segmento de código 5.7. Los conectores han seguido la misma separación que los modelos, pues de esta manera se mantenía por separado la funcionalidad y descripción de cada tabla. Además, se ha incluido aquí también una clase *helper*, necesaria para la conexión concreta con *SQLite*. En este *helper* se incluyen también funciones de inicialización de la base de datos que solo se utilizará para la base de datos de test.

```
class DeviceDB(context: Context) : DeviceInterface {
  ...

  private val dbHelper = DBHelper(context)

  override fun insert(values: ContentValues): Long {
    val db = dbHelper.getWritableDatabase(Local.db_password)

    val id = db.insert(DeviceContract.DeviceEntry.TABLE_NAME, null, values)
    db.close()

    return id
  }
  ...
}
```

Código 5.7: Conexión con el modelo *Device* de la base de datos

Se puede observar aquí la sentencia de creación y borrado de la tabla en base de datos, así como el desarrollo concretos del método `insert`. Además, se observa en la cabecera de la clase

que esta implementa la interfaz desarrollada anteriormente.

En el caso del *secure element*, se tiene por un lado el cifrado de la evidencia utilizando el sistema *Keystore*, tal, explicado en [40].

Además, el SE se podría haber utilizado para almacenar el hash de la evidencia. Esto no se ha podido integrar finalmente en el proyecto, pero se explicarán las pruebas de concepto realizadas. Para esta conexión se ha utilizado la librería *OMAPI* mencionada en la Sección 3.2.2. En el segmento de código 5.8 [26] se muestra como primero es necesario inicializar el `SEService`, posteriormente se selecciona el `reader` deseado de la lista de `readers` disponibles en el dispositivo y una vez obtenido se inica sesión y se abre un canal. Una vez abierto el canal, la forma de comunicación con el SE es mediante comandos *APDU*.

```
fun writeInSecureElement(hash: String){
    val se = SEService()

    if(se.isConnected()){
        val readers = se.getReaders()
        var seReader : Reader
 var cont = 0

 while(cont < readers.length && myReader == null){
            if(readers[cont].getName() == Local.type){
                seReader = readers[cont]
            }
        }

 if(seReader){
  val session = myReader.openSession()

  if(!session.isClosed()){
      val channel = session.openBasicChannel()

      if(channel.isOpen()){
          channel.transmit(Utils.newHashAPDU(hash))
      }

      session.closeChannels()
  }

  myReader.closeSessions()
  se.shutdown()
 }
    }
}
```

Código 5.8: Conexión con el *Secure Element*

Tras realizar pruebas con el dispositivo Google Pixel 3 con este código, se observa que este sólo posee el `reader` denominado *SIM1*. Como se observa en la documentación oficial [25] y [26], los tipos de `readers` disponibles son:

- SIM: si el reader es un lector de SIM
- SD: si el reader es un lector de SD
- eSE: si el reader es un lector de embedded SE

Por tanto, en el dispositivo de pruebas, el chip Titán M no aparece al listar los SE del dispositivo de pruebas, pese a ser un SE. En este caso, el chip Titán M debería de aparecer como `reader` de tipo *eSE*. Esto se debe a que este chip es un SE tan seguro, que el fabricante no ha dado acceso a su uso mediante esta librería.

5.2.2. Base de datos del prototipo

En el siguiente apartado se detallará el modelo entidad-relación utilizado en la base de datos de la aplicación de usuario. Este modelo entidad-relación se muestra en la Figura 5.9.

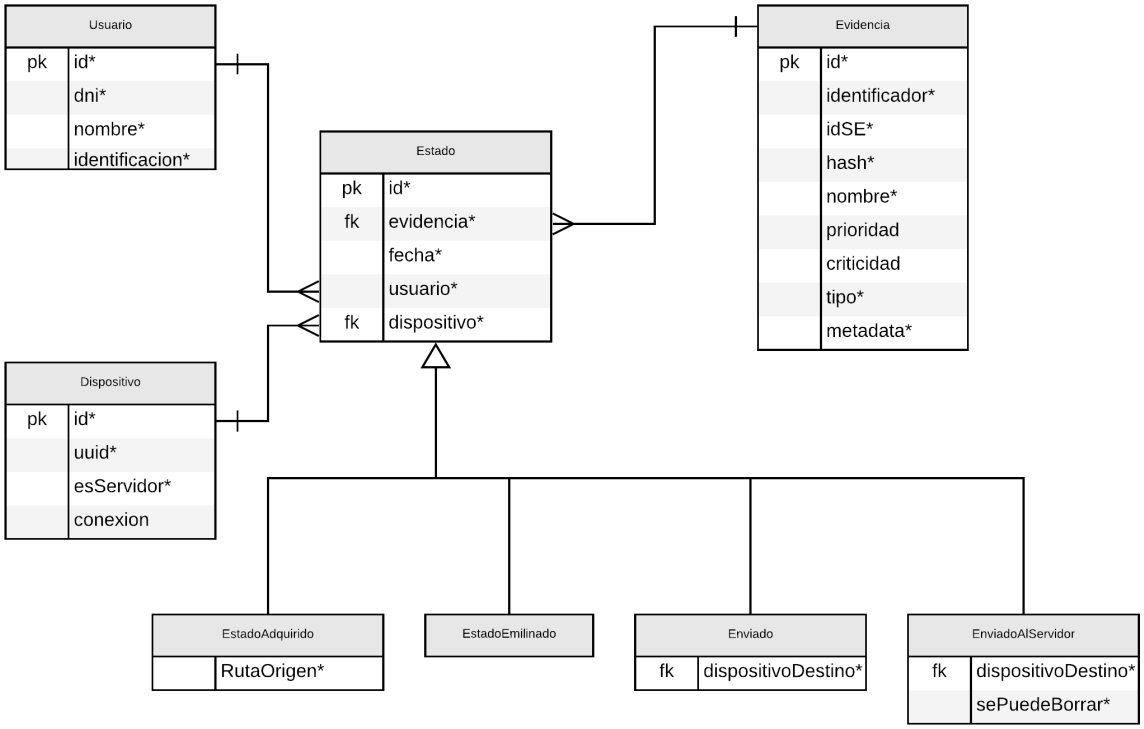


Figura 5.9: Modelo entidad-relación de la base de datos de la aplicación

En la Figurase observa que este modelo entidad-relación es parecido al diagrama de clases mostrado en los apartados anteriores. En cuanto a las diferentes entidades que se observan en el diagrama, todas han sido modelados como tablas simples incluyendo los campos mostrados en el diagrama, salvo la entidad *estado*, que ha sido modelado tal y como se muestra a continuación.

La entidad *estado* ha sido tratada de forma especial, pues cuenta con un enumerado de tipos, que son los distintos estados mencionados en el Capítulo 5.2.1, y cada uno de estos cuenta con sus propios parámetros. Por tanto, una opción habría sido utilizar la herencia para el desarrollo de esta base de datos, utilizando una tabla base con los campos comunes, y diferentes tablas hija para cada uno de los estados finales, incluyendo en cada una solo los parámetros del estado que representa, tal y como muestra la Figura 5.9. Sin embargo, *SQLite* es una base de datos muy simple y no cuenta con herencia de tabla, como si cuentan otros motores de base de datos como por ejemplo *PostgreSQL*.

Para simular esta herencia, en *SQLite* fueron valoradas tres opciones, indicando a continuación las ventajas e inconvenientes de cada una de ellas:

Opción 1. Incluir los campos de todos los estados en una misma tabla, indicando como *no obligatorios* todos los campos no comunes, y manteniendo los comunes como obligatorios. De esta forma, para controlar que un estado tuviera rellenos únicamente sus campos, se utilizarían restricciones de bases de datos para hacer obligatorios los campos según el estado. En la Figura 5.10 se observa el modelo entidad-relación de esta opción.

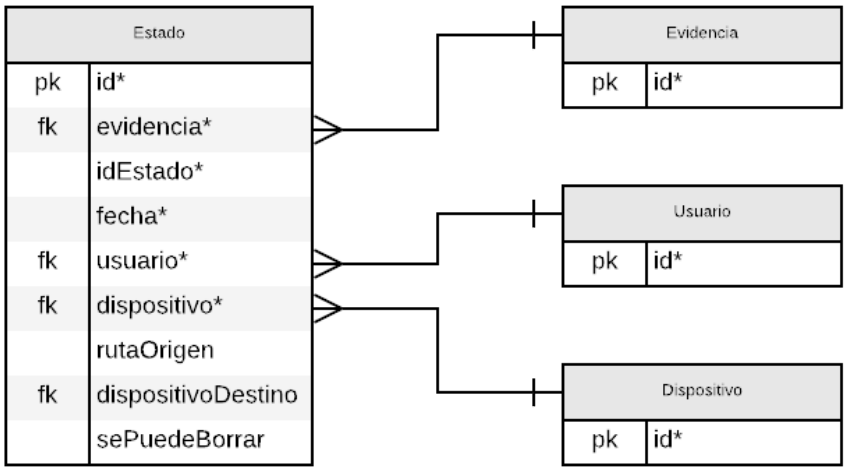


Figura 5.10: Modelo entidad-relación simplificado de la opción 1

- **Ventajas.** En el caso de necesitar nuevos estados, o eliminar alguno ya existente, solo se modificaría una tabla, pues solo requeriría de añadir columnas y restricciones a una tabla ya existente. Además, las consultas serían sobre una única tabla.
- **Inconvenientes.** Al hacer cualquier consulta se van a tener todos los campos, independientemente del estado que se haya consultado. Además, las restricciones de base de datos deben de estar bien definidas, pues complican el modelado durante el desarrollo.

Opción 2. Incluir una tabla completamente independiente para cada estado, de esta manera, cada tabla estado tendría todos los campos comunes, además de sus propios campos. Este caso se visualiza en la Figura 5.11.

- **Ventajas.** El modelado quedaría más claro, al obtener un estado obtendríamos solo sus campos. Al no tener restricciones en base de datos, el desarrollo sería mas simple.
- **Inconvenientes.** Si fuera necesario modificar los campos comunes, habría que modificarlos en todas las tablas. Además, para añadir o eliminar nuevos estados tendríamos que crear tablas y relaciones nuevas. Al realizar consultas sobre una evidencia habría que consultar varias tablas.

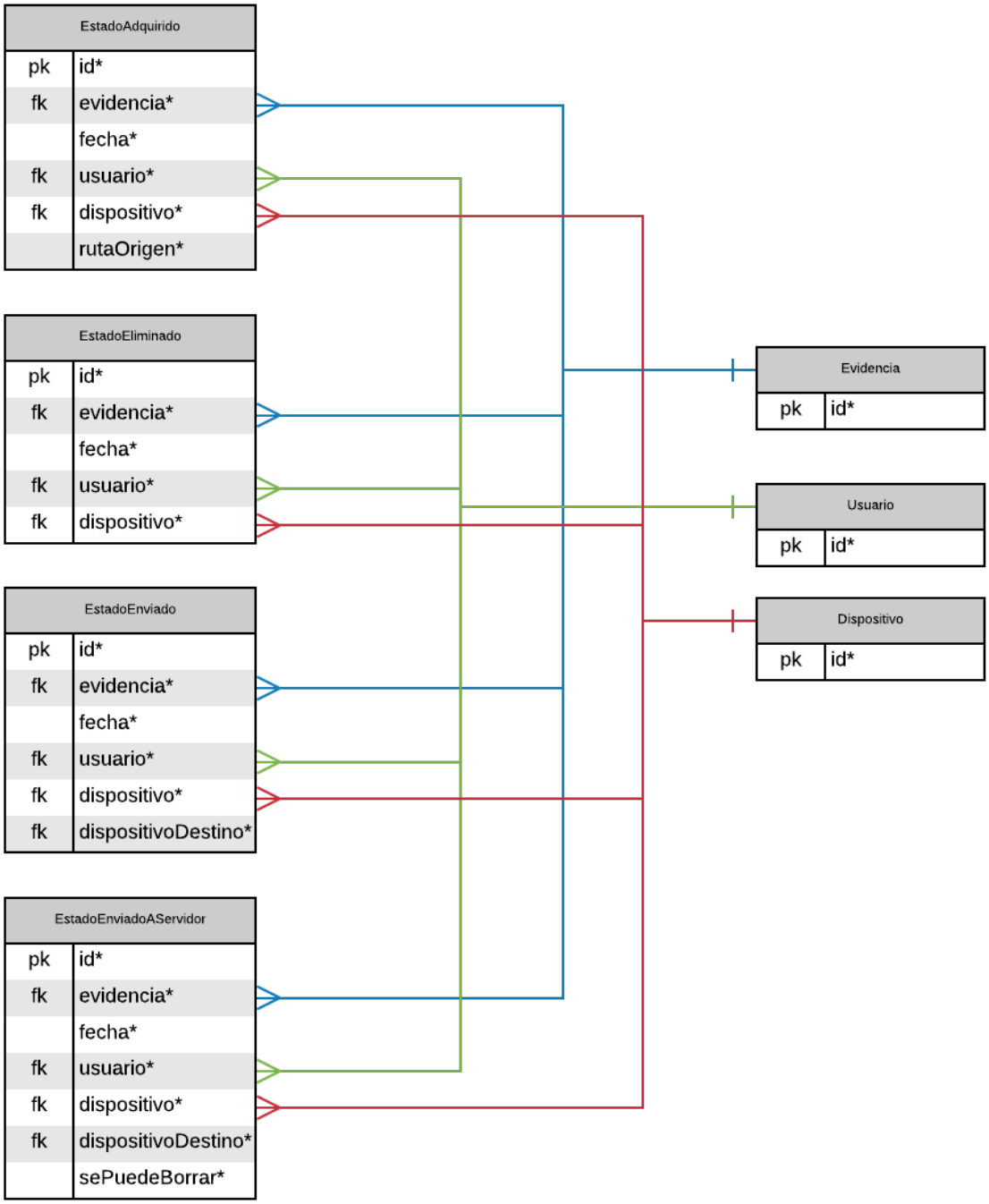


Figura 5.11: Modelo entidad-relación simplificado de la opción 2

Opción 3. Otra opción para simular la herencia sería crear una tabla base *State*, con los campos comunes, y tablas individuales para cada uno de los estados, con la información propia de cada uno. De esta forma, se tendría en cada tabla estado una *foreign key* apuntando a la tabla base. En este caso, habría que restringir que cada entrada de la tabla base *State* pertenezca a un solo estado hijo.

- **Ventajas.** Evitaría repetir campos de la tabla base, por tanto para modificar los

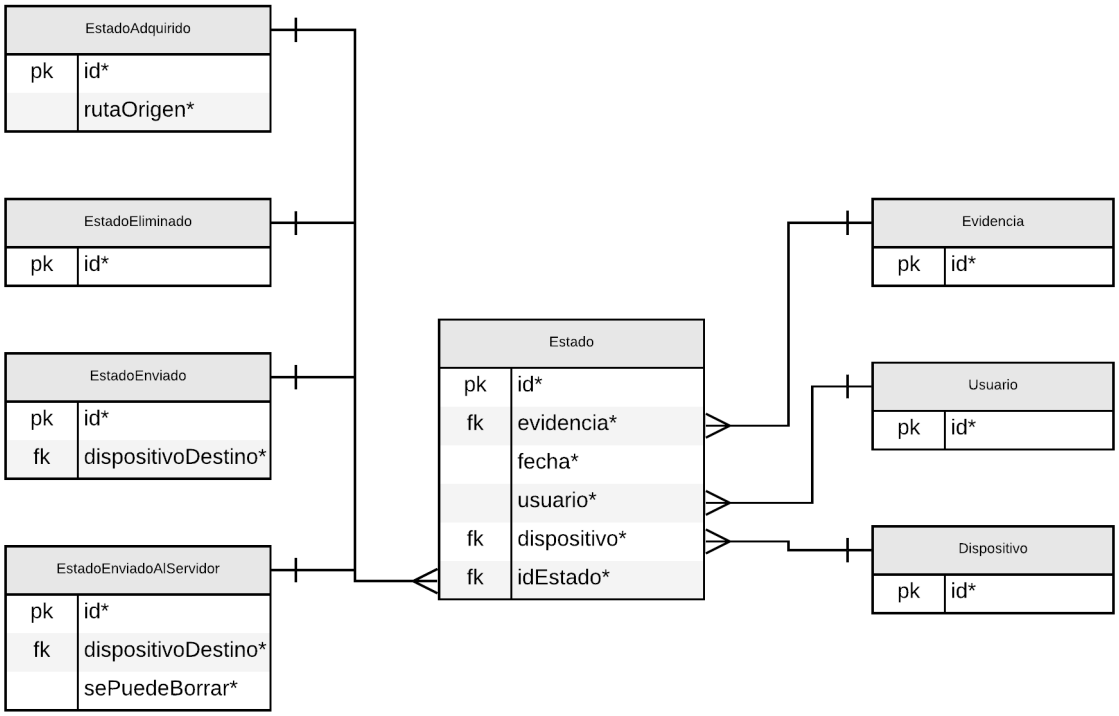


Figura 5.12: Modelo entidad-relación simplificado de la opción 3

campos comunes alguno solo necesitaríamos modificar una tabla. Es la opción que mejor simularía la herencia.

- **Inconvenientes.** Complicaría el desarrollo pues para cada estado necesitaría crear entradas en dos tablas. La restricción mencionada anteriormente se complica al utilizar múltiples tablas. Al realizar consultas sobre un solo estado se necesitaría acceder a dos tablas.

Tras este análisis, se descarta la **Opción 2.**, pues implicaría tener duplicidad de código para los campos comunes, lo que podría terminar traduciéndose en código inconsistente si al modificar los campos comunes estos no son cambiados en todas las tablas. Además, esta opción queda también descartada por el rendimiento de las consultas multitabla con la base de datos cifrada.

La **Opción 3.** es la más adecuada para simular la herencia, sin embargo también es descartada, pues al igual que con la **Opción 2.**, necesitaríamos realizar consultas multitabla, y estas deberían de reducirse al mínimo en la medida de lo posible, ya que al utilizar la base de datos cifrada se pierde mucho rendimiento.

Se opta por tanto por la **Opción 1.**, pues aunque las restricciones de base de datos son un inconveniente, es la opción con la que se obtendrá mejor rendimiento en las consultas. Para crear las restricciones sobre esta tabla, se ha creado una función que las genere de forma automática. Para ello, se mantiene una lista por cada estado, guardando en ella los campos no comunes de cada uno. De esta forma, añadir un nuevo estado implicaría añadir una nueva

lista en el contrato de base de datos, y la función automáticamente se encargaría de modelar las restricciones, sin tener que modificarlas manualmente.

Es por esto último por lo que se crearán test automáticos para comprobar que dichas restricciones han sido realizadas correctamente. Así, al modificar los modelos, únicamente será necesario lanzar los test para comprobar que todo sigue funcionando correctamente.

5.3. Desarrollo del servidor Web

Tal y como se mencionó en el Capítulo 4.4, este servidor cuenta con una *API Rest*, una base de datos, una interfaz de un usuario y el gestor de evidencias, que está realizado en Python 3.8 y Django 2.2.4. Además, para la construcción de dicha *API* se ha utilizado *Django Rest Framework*.

En cuanto a la interfaz de usuario, solo se permitirá el acceso a un usuario administrador, pudiendo unicamente ver el listado de evidencias y el detalle de cada una. Para el desarrollo de esta interfaz se ha utilizado la interfaz del panel de administración de *Django*.

5.3.1. Endpoints del servidor

Además, la forma de comunicación entre el servidor y la aplicación de usuario es una *API Rest*. A continuación se detallan los diferentes *endpoints* con los que cuenta dicho servidor.

Configuración del terminal

Este *endpoint* es el encargado de la configuración del terminal. Este será utilizado una única vez por terminal para realizar el intercambio de claves públicas entre la aplicación y el servidor. La clave pública del servidor será utilizada para cifrar la evidencias en el dispositivo, y así poder descifrar los datos de la evidencia una vez recibidos.

Comprobación del estado de conexión

Este *endpoint* se encarga de ver si el servidor actual se encuentra activo mediante la utilización del comando *ping*. Para ello, la aplicación móvil conectará con este *endpoint*, y si este le manda respuesta, será que el servidor actual se encuentra activo.

Recepción de evidencias

Este *endpoint* es utilizado para realizar el envío de evidencias desde la aplicación de usuario. Dicha aplicación enviará los datos de la evidencia, su hash, y el archivo de evidencia cifrado.

Para la realización de este envío se han utilizado los serializadores aportados por *Django Rest Framework*. Para ello, al igual que en *Kotlin*, se han desarrollado cada una de las clases, como puede verse en el segmento de código 5.9, y posteriormente se ha creado un serializador como el del segmento de código 5.10 para cada una de estas clases.

```
class Device(models.Model):
    uuid = models.CharField(unique=True, blank=False, max_length=32)
    is_server = models.BooleanField(blank=False)
    connection = models.CharField(max_length=32)

    def __str__(self):
        return self.uuid
```

Código 5.9: Clase *Device*

```
class DeviceSerializer(serializers.ModelSerializer):
    class Meta:
        model = Device
        fields = (
            'uuid',
            'is_server',
            'connection',
        )
```

Código 5.10: Serializador de la clase *Device*

Así, el dispositivo del usuario realiza el envío de los datos en formato *JSON*, que tras recibirse es validado por su serializador correspondiente y almacenado de forma automática en base de datos.

6. Validación de la solución

En esta sección se detallan las pruebas realizadas para comprobar el correcto funcionamiento de la aplicación.

Además de las pruebas de validación de la aplicación detalladas en este capítulo, se han realizado multitud de pruebas para comprobar el funcionamiento de las distintas herramientas, cómo integrarlas en esta aplicación y su grado de adecuación con este proyecto. Por ejemplo, se han utilizado las pruebas mencionadas en el Capítulo 4.2.3 para comprobar el rendimiento de la base de datos tras su cifrado.

Así, se han realizado pruebas con el sistema *Keystore* desde diferentes aplicaciones para comprobar la accesibilidad de las claves desde diferentes aplicación, como se mencionó en el Capítulo 3.2.1. También se han realizado múltitud de pruebas con la librería *OMAPI* para comprobar la conexión con el SE. En este sentido, no solo se han realizado pruebas con dispositivos que cumplían con los requisitos de la aplicación, sino que también han sido también ejecutadas en dispositivos que no cuentan con un SE, pero si con TEE. De esta forma, se ha asegurado que los resultados eran exactamente los esperados.

También se han realizado pruebas de lectura de datos del DNIe desde un terminal Android con tecnología NFC, sin embargo, se ha quedado en fase de pruebas y no ha podido ser incluido finalmente eb el proyecto.

A continuación se pasa a detallar las pruebas específicas de validación de la aplicación desarrollada.

6.1. Entorno de pruebas

El dispositivo de pruebas utilizado ha sido un dispositivo Google Pixel 3 que cuenta con Android 9 como sistema operativo, y con el chip *Titán M* como *secure element*. Este dispositivo fue detallado en el Capítulo 3.2.2.

6.2. Pruebas automáticas

Tanto la aplicación móvil como el servidor web se encuentran cubiertos por tests automáticos, estos han sido realizados únicamente sobre los procesos críticos de la aplicación.

Para la realización de estos tests automáticos se ha seguido el patrón AAA (*Arrange Act Assert*). Como se observa en el segmento de código 6.1 esto indica que cada test está divido en 3 partes diferenciadas, separadas por espacios. Al principio se encuentran la declaración de

variables y preparación del entorno del test, a continuación se encontraría la sentencia que queremos verificar, y finalmente los asertos que comprueban la validez del test.

```
def test(self):
 # arrange
 value1 = 1
 value2 = 2
 expected_value = 3

 # act
 value = sum(value1, value2)

 # assert
 self.assertEqual(value, expected_value)
```

Código 6.1: Ejemplo de test automático utilizando AAA, escrito en Python

6.2.1. Aplicación Android

Android cuenta con herramientas para el desarrollo de tests automáticos para sus aplicaciones. Por defecto, al crear una nueva aplicación el IDE genera un entorno para pruebas unitarias y otro para pruebas de integración (*intrumented tests*). Los tests unitarios se lanzan en el ordenador del desarrollador y sirven para verificar que los los métodos y funciones se encuentran funcionando correctamente. Los *instrumented* tests se ejecutan en el dispositivo Android, y utilizan además elementos del sistema, como el *context* de Android y conexión real con la base de datos.

Paralelamente al desarrollo de la aplicación Android se ha llevado a cabo el desarrollo de los tests de esta. Cabe destacar que sólo se han creado tests sobre aquellos procesos que han sido considerados más críticos, y estos únicamente de integración, pues en el caso de la adquisición de evidencias, se quería comprobar la integración real con la base de datos y el *secure element*.

Únicamente se han realizado test para verificar la adquisición de la evidencia, el modelo *State*, y un par de acciones realizadas sobre las evidencias. El modelo *State* ha sido verificado por contar con restricciones a nivel de base de datos. Los *mocks* se han utilizado solo para las conexiones con el servidor. En el segmento de código 6.2 se muestra un ejemplo de estos test realizados en Kotlin.

```kotlin
@RunWith(AndroidJUnit4::class)
class StateIntegrationTests {

    private lateinit var helper: DBHelper
    private lateinit var context: Context

    @Before
    fun setUp() {
        context = InstrumentationRegistry.getTargetContext()
        helper = DBHelper(context, true)
        helper.cleanAndRecreate(context)
    }

    @After
    fun tearDown() {
        helper.clean(context)
    }

    @Test
    fun state_acquired_save_ok() {
        val user = createUser(context)
        val device = createDevice(context)
        val evidence = createEvidence(context)
        val state = State(
            null,
            evidence,
            StateContract.ACQUIRED,
            Date(),
            user,
            device,
            "path"
        )
        val expectedId = 1L

        state.save(context)

        assertEquals(expectedId, state.id)
    }
    ....
}
```

Código 6.2: Test de la aplicación

Para la realización de estas pruebas se ha utilizado una base de datos de pruebas, que es creada al inicio de la ejecución de los tests, y destruida tras su finalización. Esto puede observarse en los métodos setUp y tearDown del segmento de código 6.2. Estos métodos serán los encargados de recrear y eliminar dicha base de datos. En la Figura 6.1 se observa como la aplicación supera todos los tests satisfactoriamente.

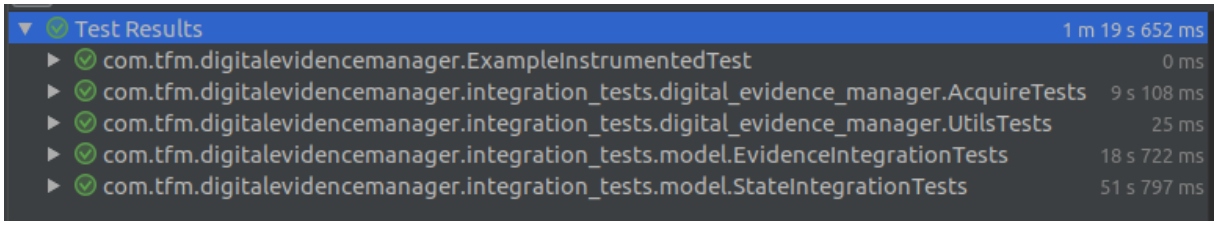


Figura 6.1: Resultados de los tests automáticos de la aplicación

6.2.2. Servidor

De forma similar, se han preparado tests para verificar los diferentes endpoints del servidor. Para ello, Python cuenta con el framework *pytest*, que genera automáticamente un entorno de test que recreará una base de datos para la ejecución de estos. Dicha base de datos será destruida al finalizar los tests. En el segmento de código 6.3 se encuentra un ejemplo de test del servidor, en este caso se trata de tests de los serializadores.

```
1  class DeviceSerializerUnitTest(TestCase):
2
3      def test_all_fields(self):
4          device = {'uuid': '111', 'is_server': True, 'connection': '0.0.0.0'}
5          device_serializer = DeviceSerializer(device)
6
7          valid = device_serializer.is_valid()
8
9          self.assertEqual(valid, True)
10
11     def test_uuid_mandatory(self):
12         device = {'is_server': True, 'connection': '0.0.0.0'}
13         device_serializer = DeviceSerializer(device)
14
15         valid = device_serializer.is_valid()
16
17         self.assertEqual(valid, False)
```

Código 6.3: Test del servidor

Al igual que en la aplicación Android, solo se han realizado tests automáticos sobre los procesos críticos del sistema.

6.3. Pruebas manuales

Mediante pruebas manuales se ha verificado la interfaz de usuario y el resto de métodos que no han sido testeados con pruebas automáticas. Por ejemplo, se han llevado a cabo aquí verificaciones de que los listados mostraban unicamente los datos que tenían que mostrar, y que los botones habilitados o deshabilitados en cada momento eran los correctos.

Además, se ha verificado que la adquisición de la evidencia se realiza correctamente y los datos almacenamos han sido los correctos.

También se ha verificado de forma manual el envió y recepción de evidencias entre la aplicación móvil y el servidor web.

7. Conclusiones y trabajo futuro

Este nuevo enfoque planteado a la gestión de evidencias hace que, tanto el auge de los dispositivos móviles, como su gran papel en la obtención de datos, y las nuevas características hardware de seguridad de los dispositivos, sea posible un futuro el planteamiento del testigo digital para aportar validez a las evidencias en un proceso judicial.

La solución planteada implementa la figura del Testigo Digital en los dispositivos Android de manera novedosa, aprovechando la arquitectura de seguridad con la que cuenta este sistema. Siendo además la única de las herramientas analizadas en el Capítulo 2 en la que la evidencia se define en el propio dispostivo, por un usuario que se entiende interesado en el análisis de la evidencia. Es un punto de vista distinto por ejemplo que cuando se ve desde la perspectiva de un dispositivo requisado por la policía. En este otro caso deberían considerarse otros requisitos para la adquisición que se han discutido para el caso del testigo custodio, no cubiertos en este trabajo.

Este proyecto se centra en el análisis de la gestión de evidencias en dispositivos Android basado en la figura del Testigo Digital, como una solución que permite a los usuarios decidir qué evidencias digitales quieren enviar para su análisis a una entidad confiable. La solución propuesta implica tanto el diseño de una aplicación de usuario (para el testigo digital) como un servidor que recibirá las evidencias y las guardará para su posterior análisis por parte de los expertos. Sin embargo, este proyecto se centrará en la aplicación de usuario, dejando el servidor que recepciona las evidencias como una aplicación web que permita validar la solución. Se destaca que se ha puesto especial énfasis en la fase de obtención de evidencias (Capítulo 5.2), así la transmisión se ha definido como parte de las características del testigo. Aunque no era el objetivo principal, ha requerido la implementación y configuración del servidor, como punto oficial de recogida de evidencias, siendo otra contribución derivada de este trabajo. Esto permite tener un elemento que permita extender esta solución a otros dispositivos, contando con un servidor que valide las evidencias digitales, siempre y cuando se respeten las pautas que se indican más adelante para su uso.

Cabe destacar que os recientes adelantos tecnológicos de los dispositivos personales podrían influir positivamente para permitir poco a poco la inclusión de testigos digitales o soluciones similares en los procesos oficiales. Como bien se demuestra en [15], si los dispositivos son capaces de sustituir por ejemplo a los DNI físicos en un momento dado, no es descabellado imaginar que puedan ser empleados para otras funciones delicadas como la testificación digital.

Este proyecto ha cumplido con los objetivos propuestos, pues pretendía analizar la viabilidad del concepto de testigo digital de forma práctica, además de desarrollar una herramienta

proactiva que sirviera de prueba de concepto de este.

La prueba de concepto desarrollada durante este TFM sienta las bases para desarrollar sistemas más complejos en un futuro, incluso para exportar el trabajo a otros sistemas. Por ejemplo, estas serían las mejoras que podrían realizarse sobre dicho prototipo:

- **Identificación de usuario**. Actualmente se todas las acciones realizadas conllevan la asignación de un usuario, sin embargo este no tiene forma de identificarse, por lo que actualmente se utiliza un usuario genérico.
- **Ampliación de adquisición de todo tipo de archivos**. Actualmente solo se permite la adquisición de imágenes, esto podría ampliarse y admitir todo tipo de archivos. Junto a este punto podría permitirse también la adquisición de más de un archivo cada vez. Realizar la adquisición de todo tipo de archivos desde Android no es difícil, pues consiste en cambiar la configuración al abrir el gestor de archivos para seleccionar una imagen, en concreto es necesario cambiar el tipo de `Intent` utilizado al crear la actividad. En el caso de la adquisición de imágenes, se ha utilizado el `Intent.ACTION_PICK`. De esta forma, seleccionando el `Intent` adecuado, el dispositivo mostraría todo el gestor de archivos en vez de filtrar por imágenes.
- **Permitir la adquisición de archivos grandes**. Ahora mismo la adquisión se realiza utilizando el método `File.readBytes()`, el cual tiene una limitación de 2Gb.
- **Firmar la evidencia con el DNIe antes de enviarla al servidor**. El DNIe actual permite firmar documentos. Mediante la tecnología NFC se permite el acceso a los datos de estos DNIe desde el dispositivo Android. Se ha realizado una pequeña prueba de concepto pero esta no ha podido ser añadida finalmente al proyecto.
- **Mejoras del listado de evidencias**. El listado de evidencias podría mostrar opciones de filtrado y ordenación, además de un seleccionable para permitir el borrado y envío de varias evidencias a la vez, sin ser todas.
- **Transmisión de evidencias entre dispositivos Android**. Esta transmisión podría realizarse mediante el uso de la tecnología NFC.
- **Utilizar blockchain para seguir la cadena de custodia de la evidencia**. Sería interesante implementar *blockchain* para seguir esta cadena de custodia digital. Este es de hecho un planteamiento que ya se ha sugerido en algunos artículos relacionados con el testigo digital.
- **Optimización de consultas de base de datos**. Actualmente las consultas de base de datos son costosas en cuanto a tiempo, una posible mejora sería la optimización de dichas consultas para evitar consultas multitabla y reducir al mínimo los accesos a base de datos.

Bibliografía

[1] R. Ayers, S. Brothers, and W. Jansen, *Guidelines on Mobile Device Forensics*. National Instue of Standards and Technology, 2014.

[2] A. Nieto, R. Román, and J. López, *Digital Witness: Safeguarding Digital Evidences by using Secure Architectures into Personal Devices*. Universidad de Málaga.

[3] *Multinorma UNE 71505:2013 - Gestión de Evidencias Electrónica*.

[4] *Norma UNE 71506:2013 - Evidencias electrónicas*.

[5] *Norma UNE 19010:2015 - Redacción de informe pericial*.

[6] "Qué son las evidencias electrónicas." https://www.csuc.cat/es/e-administracion/evidencias-electronicas/que-son-las-evidencias-electronicas.

[7] A. Nieto, *Apuntes de la asignatura Informática Forense, Máster Universitario en Ingeniería Informática*. UMA.

[8] H. Chung, J. Park, and S. Lee, "Digital forensic approaches for amazon alexa ecosystem." https://www.sciencedirect.com/science/article/pii/S1742287617301974.

[9] J. Mpreno, I. Leguias, and M. V. Lombardo, "Revisión sobre la forensía en dispositivos móviles con sistemas operativos android." https://revistas.utp.ac.pa/index.php/id-tecnologico/article/view/2076/3031.

[10] "Virustotal: Getting started." https://developers.virustotal.com/reference.

[11] J. Ranchal, "Microsoft mejora la seguridad de windows 10: Tpm 2.0 obligatorio." https://www.muycomputer.com/2016/07/28/seguridad-de-windows-10/.

[12] A. Llorente, "Módulo tpm para ciberseguridad en el coche conectado." https://www.seguridadprofesionalhoy.com/modulo-tpm-para-ciberseguridad-en-el-coche-conectado/.

[13] IEEE Security and Privacy, *Biometric authentication on mobile devices*.

[14] Área de Innovación y Laboratorio de ElevenPaths, "Carrier level immutable protection (clip): nuestra tecnología segura y confiable para el empoderamiento de operadores." https://empresas.blogthinkbig.com/tecnologia-clip-empoderamiento-operadores-ciberseguridad/.

[15] M. Arjona, "Tu móvil será tu dni antes de lo que imaginas." https://blogthinkbig.com/dni-movil-telefonica.

[16] "Arquitectura de la plataforma." https://developer.android.com/guide/platform/.

[17] "System and kernel security." https://source.android.com/security/overview/kernel-security.html.

[18] "Cts test for secure element." https://source.android.com/compatibility/cts/secure-element.

[19] "Android enterprise security white paper."

[20] "Android keystore system." https://developer.android.com/training/articles/keystore.

[21] "Android keystore: what is the difference between "strongbox" and "hardware-backed" keys?." https://proandroiddev.com/android-keystore-what-is-the-difference-between-strongbox-and-hardware-backed-keys-4c276ea78fd0.

[22] J. O'Donoghue, "As easy as pie: Google's android implementation of omapi." https://globalplatform.org/as-easy-as-pie-googles-implementation-of-omapi-in-android-paves-the-way-for-simple-deployment-of-secure-apps-across-a-broad-range-of-mobile-devices/.

[23] "Iso/iec 7816."

[24] "Smart card application protocol data unit."

[25] GlobalPlatform Technology, *Open Mobile API Specification Version 3.3*.

[26] GlobalPlatform Technology, *Open Mobile API – Android Binding Version 0.0.0.5 for OMAPI v3.3*.

[27] X. Xin, "Titan m makes pixel 3 our most secure phone yet." https://www.blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/.

[28] N. Modadugu and B. Richardson, "Building a titan: Better security through a tiny chip." https://android-developers.googleblog.com/2018/10/building-titan-better-security-through.html.

[29] B. S. Delgado, "Así es titan m, el chip de seguridad exclusivo de los google pixel." https://elandroidelibre.elespanol.com/2018/10/google-seguridad-titan-m.html.

[30] "How fido works." https://fidoalliance.org/how-fido-works/.

[31] "Opciones de almacenamiento." https://developer.android.com/guide/topics/data/data-storage?hl=es-419.

[32] F. Akowuah, A. Ahlawat, and W. Du, *Protecting sensitive sata in Android SQLite databases using TrustZone*. Electrical Engineering and Computer Science Department of Syracuse University.

[33] "Sqlite encryption extension: Documentation." https://www.sqlite.org/see/doc/release/www/readme.wiki.

[34] "Full database encryption for sqlite." https://www.zetetic.net/sqlcipher/.

[35] "Tutorial: Add sqlcipher to your android app." http://lomza.totem-soft.com/tutorial-add-sqlcipher-to-your-android-app/.

[36] "Biometricprompt." https://developer.android.com/reference/android/hardware/ biometrics/BiometricPrompt.html.

[37] "Cómo solicitar permisos durante el tiempo de ejecución." https://developer.android.com/training/permissions/requesting.html.

[38] "Permissions overview." https://developer.android.com/guide/topics/permissions/overview.

[39] T. Rodríguez, "Kotlin ya es un lenguaje oficial en android: ¿qué implicaciones tiene y por qué es tan importante?." https://www.xatakandroid.com/programacion-android/kotlin-ya-es-un-lenguaje-oficial-en-android-que-implicaciones-tiene-y-por-que-es-tan-importante.

[40] J. Sena, "Using the android keystore system to store and retrieve sensitive information." https://medium.com/@josiassena/using-the-android-keystore-system-to-store-sensitive-information-3a56175a454b.

[41] "Probar tu app." https://developer.android.com/studio/test.

[42] "Framework de acceso a almacenamiento." https://developer.android.com/guide/topics/providers/document-provider.

[43] "Five android net scanning tools for mobile troubleshooting." https://www.techrepublic.com/blog/five-apps/five-android-net-scanning-tools-for-mobile-troubleshooting/r.

[44] "Metasploit basics. part 13: Exploiting android mobile devices." https://www.hackers-arise.com/single-post/2018/07/06/Metasploit-Basics-Part-13-Exploiting-Android-Mobile-Devices.

[45] "Trusted platform module information for intel nuc." https://www.intel.co.uk/content/www/uk/en/support/articles/000007452/mini-pcs.html.

[46] "Introduction to secure elements." https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Secure-Element-15May2018.pdf.

[47] H. Chung, J. Park, and S. Lee, "Ieee security and privacy." https://www.sciencedirect.com/science/article/pii/S1742287617301974.