UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Seguridad en Dispositivos Móviles: Explorando las Tecnologías de Entorno de Ejecución Confiable

Trabajo fin de máster

Máster Universitario en Ciberseguridad y Ciberinteligencia

Autor: Josep Comes Sanchis

Tutor: Héctor Marco Gisbert

Curso 2023-2024

Resum

Amb el continu creixement de l'ús de dispositius mòbils a la nostra vida diària, la seguretat de la informació emmagatzemada i transmesa a través d'aquests dispositius es converteix en una preocupació fonamental. La proliferació d'aplicacions mòbils per a tasques quotidianes, des de la banca fins a la salut, amplifica la importància de salvaguardar les dades personals contra possibles amenaces cibernètiques. En aquesta era d'interconnexió digital, on els dispositius mòbils han esdevingut companys indispensables en les nostres activitats diàries, és crucial explorar solucions efectives per garantir la integritat i la confidencialitat de les dades en tot moment.

Aquest treball se centra a explorar les tecnologies d'Entorn d'Execució Confiable (TEE) com una solució integral per abordar els desafiaments de seguretat en dispositius mòbils. Els TEE ofereixen una gamma de beneficis significatius per a la seguretat en dispositius mòbils. Aquests beneficis inclouen proporcionar un entorn segur i aïllat al dispositiu, on es poden executar aplicacions i processos crítics, protegint així les dades crítiques davant accessos no autoritzats. A més, els TEE garanteixen la integritat del sistema i l'autenticitat de les aplicacions, cosa que prevé la manipulació o substitució d'aplicacions per part de tercers malintencionats. La investigació s'enfoca a comprendre com aquestes tecnologies poden garantir un entorn segur per a l'execució d'aplicacions i resguardar la integritat de les dades contra possibles amenaces.

A continuació, es presenta el desenvolupament d'una aplicació bancària, dividida en quatre parts: la primera consta d'una aplicació per a Android que actua com a interfície gràfica d'usuari; la segona, és una aplicació que s'executa a l'entorn normal i actua com a intermediària entre l'aplicació d'Android i l'aplicació fiable (TA); la tercera part és la TA, que s'executa a l'entorn segur i s'encarrega de gestionar el material criptogràfic; i, finalment, la quarta part comprén un petit servidor extern que simula una entitat bancària real.

Finalment, el treball destaca els aspectes més rellevants de la investigació, especialment centrats en la implementació pràctica de tecnologies de TEE a dispositius mòbils. Cal subratllar la necessitat de desenvolupar dues aplicacions diferents: una per a Android i una altra per a l'entorn segur, com a mesura fonamental per garantir la seguretat davant d'atacs cibernètics. A més, s'emfatitzen les contribucions específiques de l'estudi, que inclouen la descripció detallada del desenvolupament d'una aplicació bancària. Aquesta aplicació no només demostra l'efectivitat i la utilitat de les tecnologies de TEE en entorns mòbils, sinó que també presenta un enfocament innovador per abordar desafiaments de seguretat específics en aplicacions financeres. Per exemple, en executar aplicacions sensibles en un entorn segur, els TEE protegeixen contra atacs de codi maliciós que podrien comprometre el sistema operatiu principal del dispositiu. En proporcionar una solució pràctica i efectiva, aquest estudi contribueix significativament al camp de la seguretat en dispositius mòbils, destacant la importància de la implementació de tecnologies de TEE com una mesura crucial per protegir la integritat i confidencialitat de les dades de l'usuari.

Paraules clau: TEE, Trustzone, Arm, Dispositiu mòbil, Android, Seguretat de la informació, Aplicació confiable

Resumen

Con el continuo crecimiento del uso de dispositivos móviles en nuestra vida diaria, la seguridad de la información almacenada y transmitida a través de estos dispositivos se convierte en una preocupación fundamental. La proliferación de aplicaciones móviles para tareas cotidianas, desde la banca hasta la salud, amplifica la importancia de salvaguardar los datos personales contra posibles amenazas cibernéticas. En esta era de interconexión digital, donde los dispositivos móviles se han convertido en compañeros indispensables en nuestras actividades diarias, es crucial explorar soluciones efectivas para garantizar la integridad y la confidencialidad de los datos en todo momento.

Estos beneficios incluyen proporcionar un entorno seguro y aislado en el dispositivo, donde se pueden ejecutar aplicaciones y procesos críticos, protegiendo así los datos sensibles contra accesos no autorizados. Además, los TEE garantizan la integridad del sistema y la autenticidad de las aplicaciones, lo que previene la manipulación o sustitución de aplicaciones por parte de terceros malintencionados. La investigación se enfoca en comprender cómo estas tecnologías pueden garantizar un entorno seguro para la ejecución de aplicaciones y resguardar la integridad de los datos contra posibles amenazas.

A continuación, se presenta el desarrollo de una aplicación bancaria, dividida en cuatro partes: la primera consta de una aplicación para Android que actúa como interfaz gráfica de usuario; la segunda, en una aplicación que se ejecuta en el entorno normal y actúa como intermediaria entre la aplicación de Android y la aplicación confiable (TA); la tercera parte es la TA, que se ejecuta en el entorno seguro y se encarga de gestionar el material criptográfico; y, finalmente, la cuarta parte comprende un pequeño servidor externo que simula una entidad bancaria real.

Finalmente, el trabajo destaca los aspectos más relevantes de la investigación, con un enfoque en la implementación práctica de tecnologías de TEE en dispositivos móviles. Se resalta la necesidad de desarrollar dos aplicaciones distintas: una para Android y otra para el entorno seguro, como medida fundamental para garantizar la seguridad frente a ataques cibernéticos. Además, se enfatizan las contribuciones específicas del estudio, que incluyen la descripción detallada del desarrollo de una aplicación bancaria. Esta aplicación no solo demuestra la efectividad y utilidad de las tecnologías de TEE en entornos móviles, sino que también presenta un enfoque innovador para abordar desafíos de seguridad específicos en aplicaciones financieras. Por ejemplo, al ejecutar aplicaciones sensibles en un entorno seguro, los TEE protegen contra ataques de malware que podrían comprometer el sistema operativo principal del dispositivo. Al proporcionar una solución práctica y efectiva, este estudio contribuye significativamente al campo de la seguridad en dispositivos móviles, destacando la importancia de la implementación de tecnologías de TEE como una medida crucial para proteger la integridad y la confidencialidad de los datos del usuario.

Palabras clave: TEE, Trustzone, Arm, Dispositivo móvil, Android, Seguridad de la información, Aplicación confiable

Abstract

With the continued growth in the use of mobile devices in our daily lives, the security of information stored and transmitted through these devices becomes a critical concern. The proliferation of mobile applications for everyday tasks, from banking to healthcare, has raised the importance of safeguarding personal data against potential cyber threats. In this era of digital interconnection, where mobile devices have become indispensable companions in our daily activities, it is crucial to explore effective solutions to ensure data integrity and confidentiality at all times.

This paper focuses on exploring Trusted Execution Environment (TEE) technologies as a comprehensive solution to address mobile device security challenges. TEEs offer a range of significant benefits for mobile device security. These benefits include providing a secure, isolated environment on the device where critical applications and processes can run, thus protecting sensitive data from unauthorized access. In addition, TEEs ensure system integrity and application authenticity, which prevents tampering or substitution of applications by malicious third parties. The research focuses on understanding how these technologies can ensure a secure environment for application execution and safeguard data integrity against potential threats.

The following is the development of a banking application, divided into four parts: the first consists of an Android application that acts as a graphical user interface; the second, an application that runs in the normal environment and acts as an intermediary between the Android application and the trusted application (TA); the third part is the TA, which runs in the secure environment and is responsible for managing the cryptographic operations; and finally, the fourth part comprises a small external server that simulates a real banking institution.

Finally, the paper highlights the most relevant aspects of the research, with a focus on the practical implementation of TEE technologies in mobile devices. It highlights the need to develop two different applications: one for Android and one for the secure environment, as a fundamental measure to ensure security against cyber-attacks. In addition, the specific contributions of the study are emphasized, which include the detailed description of the development of a banking application. This application not only demonstrates the effectiveness and usefulness of TEE technologies in mobile environments, but also presents an innovative approach to address specific security challenges in financial applications. For example, by running sensitive applications in a secure environment, TEEs protect against malware attacks that could compromise the device's core operating system. By providing a practical and effective solution, this study contributes significantly to the field of mobile device security, highlighting the importance of implementing TEE technologies as a crucial measure to protect the integrity and confidentiality of user data.

Key words: TEE, Trustzone, Arm, Mobile device, Android, Information security, Trusted application

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

El presente trabajo se centra en un estudio detallado de los *Trusted Execution Environment* (TEE), una tecnología ampliamente utilizada en dispositivos móviles. En esta investigación se exploran las características y funcionalidades de esta tecnología, así como las diferentes implementaciones en el entorno de los dispositivos móviles. Además, se tendrá en cuenta un modelo de amenazas para evaluar los posibles riesgos y vulnerabilidades asociados con el uso de TEE. En última instancia, se utilizará toda esta información para desarrollar una aplicación, que se respalde en un TEE para mejorar la seguridad durante el inicio de sesión.

1.1 Motivación

En la actualidad, los dispositivos móviles se han convertido en una parte integral de nuestras vidas, utilizados para una amplia gama de actividades, desde la comunicación hasta las transacciones financieras. Sin embargo, esta creciente dependencia también ha llevado consigo un aumento considerable en las amenazas cibernéticas [1], como por ejemplo las constantes campañas de *smishing* [2], los clásicos troyanos bancarios [3], u otro tipo de ataques más sofisticados [4].

Ante tal escenario, surgen nuevos desafíos significativos en la protección de la información personal y sensible de los usuarios. En este contexto, los *Trusted Execution Environments* (TEE) se posicionan como una tecnología clave en la protección de la seguridad y la privacidad en dispositivos móviles. Esta tecnología ofrece un entorno seguro y aislado del entorno de ejecución principal del dispositivo, en el que se garantiza que los datos sensibles se almacenan, procesan y protegen frente a cualquier tipo de ataque.

Al mismo tiempo, la arquitectura Arm ha emergido como la opción dominante en dispositivos móviles debido a su eficiencia energética y rendimiento. En los últimos años, esta arquitectura ha ampliado su alcance, extendiéndose más allá de los teléfonos móviles para abarcar también ordenadores personales y servidores [5]. Una de las características destacadas de la arquitectura Arm son sus extensiones, que permiten agregar funcionalidades adicionales a los procesadores. Entre estas extensiones, destacan las *Security Extensions*, introducidas en la arquitectura Armv6K, que forman parte de la arquitectura de seguridad TrustZone. Estas extensiones proporcionan un conjunto de funciones de seguridad a nivel de hardware, facilitando así la implementación de TEEs.

1.2 Objetivos

El presente trabajo tiene por objetivo realizar un estudio exhaustivo sobre los Trusted Execution Environments en dispositivos móviles, destacando su importancia y aplicación práctica. Los objetivos que se persiguen en este estudio son:

- Explorar el concepto de Trusted Execution Environment y sus aplicaciones en dispositivos móviles.
- Estudiar la arquitectura Arm y las extensiones de seguridad de los procesadores Arm que respaldan la arquitectura de seguridad TrustZone.
- Analizar la implementación del Trusted Execution Environment en dispositivos móviles de diferentes fabricantes.
- Evaluar los posibles riesgos y vulnerabilidades asociados con el uso de TEE mediante un modelo de amenazas.
- Motivar un caso de uso relevante que demuestre la aplicación práctica y los beneficios de un Trusted Execution Environment en comparación con el modelo de amenazas analizado para el desarrollo de una aplicación de banca móvil.

1.3 Estado del arte

Los dispositivos móviles desempeñan un papel fundamental en nuestra vida diaria, siendo utilizados para una amplia gama de actividades, desde transacciones comerciales hasta comunicaciones personales. La confianza en la seguridad y privacidad de estos dispositivos es esencial para garantizar su uso sin problemas y proteger la información sensible. Con el fin de garantizar esta confianza, los dispositivos móviles modernos cuentan con un TEE. Esta tecnología tuvo sus raíces en los primeros *smartphones* de Nokia a principios del siglo XXI, gracias al trabajo pionero de sus ingenieros [6].

Para entender el nacimiento de esta tecnología, hay que remontarse a la década de 1990, cuando la industria de las telecomunicaciones experimentó una revolución significativa. En ese momento, las comunicaciones móviles se basaban en señales analógicas, lo que planteaba preocupaciones de seguridad debido a la falta de cifrado. Durante esta década, el estándar *Global System for Mobile Communications* (GSM) fue adoptado para las comunicaciones móviles. Este estándar, diseñado para redes de segunda generación (2G), introdujo la digitalización de las comunicaciones móviles, lo que proporcionó capacidades de cifrado de extremo a extremo. Por otro lado, las redes móviles estaban mayormente operadas por entidades gubernamentales, hasta que la desregulación abrió el mercado a operadores privados. Por último, a finales de la década de 1990, el soporte para aplicaciones de terceros escritas en el lenguaje de programación Java, que podían descargarse desde Internet, transformó rápidamente los teléfonos de dispositivos cerrados a sistemas abiertos que cada vez se asemejaban más al PC. Como resultado del aumento de usuarios, una disminución del control gubernamental sobre la industria y más fuentes de vulnerabilidades potenciales, la seguridad de los dispositivos emergió como un nuevo problema en el diseño de los teléfonos móviles.

Con estas nuevas condiciones, surgió la necesidad de garantizar la integridad de los dispositivos móviles. Las autoridades y operadores de red buscaron proteger información crítica, como el *International Mobile Equipment Identity* (IMEI), de modificaciones no autorizadas una vez que los dispositivos salían de la cadena de producción.

En los años 90, las medidas de seguridad se basaban principalmente en soluciones de software y dependían en gran medida del secretismo interno de las organizaciones. Sin embargo, el principal problema de este enfoque, es que funciona hasta que se descubren los detalles de la implementación, lo que llevó a la búsqueda de soluciones más sólidas.

El interés por una plataforma de seguridad coherente y reforzada por hardware surgió de un equipo de ingenieros que trabajaban con pagos móviles y seguridad. La idea inicial era introducir un chip de seguridad independiente para aislar físicamente los procesos críticos. No obstante, introducir un chip hardware adicional requería de un elevado coste, por lo que la idea fue descartada.

En los albores del nuevo milenio, un ingeniero de Nokia propuso una innovadora solución que combinaba hardware y software para crear un entorno seguro. Esta solución, implementada en los teléfonos *Base Band 5*, introdujo un modo de seguridad aislado utilizando un único chip físico. Esta iniciativa allanó el camino para futuros avances en la seguridad de dispositivos móviles. Alrededor de 2003, Arm propuso a Nokia desarrollar un aislamiento de hardware de todo el sistema para la ejecución segura en cualquier chip que implementara la arquitectura de seguridad de Arm, que más tarde se conocería como Arm Trustzone [7] (pp. 18-24).

A medida que el TEE se estableció como un estándar en Nokia, la compañía se interesó en los estándares de seguridad móvil a nivel internacional. Participaron activamente en foros de estandarización para garantizar que los estándares emergentes fueran compatibles con su solución. De este modo, Nokia presidió un grupo de trabajo dentro del *Trusted Computing Group* (TCG). Aunque este grupo estaba formado por compañías del sector de PC, era el único que por los años 2000 trabajaba con estándares de seguridad hardware para uso global.

El concepto de TEE fue descrito públicamente en el año 2009 por Open Mobile TerminalPlatform en su especificación Advanced Trusted Environment [8]. Tiempo después, el organismo encargado de la normalización de TEE se convirtió en otro foro del sector, *GlobalPlatform* (GP). Con dos foros industriales, existía el riesgo de que acabaran con especificaciones incompatibles entre sí. En 2010, GP publica su primer estándar TEE Client API 1.0 [9] que define la comunicación entre las aplicaciones que se ejecutan en un TEE, y las aplicaciones que se ejecutan en el entorno de ejecución principal. En 2012, GP y TCG anuncian la creación de un grupo conjunto de trabajo centrado en temas de seguridad. Este enfoque colaborativo fue fundamental para armonizar los estándares y promover la interoperabilidad en el campo de la seguridad móvil.

A partir de 2010, Nokia redujo su participación en foros internacionales, lo que afectó su influencia en el desarrollo de la tecnología TEE. A pesar de esto, la investigación y desarrollo en torno a la TEE continuó avanzando. Hoy en día, esta tecnología se encuentra implementada de tres formas distintas:

- La implementación más extendida consiste en dividir los recursos del sistema de manera virtual. En esta configuración divide un único núcleo físico en dos virtuales mutuamente excluyentes, de modo que cuando uno de los dos núcleos virtuales está en uso el otro permanece suspendido.

- La segunda implementación más extendida consiste en el uso de un coprocesador dentro del mismo *System on a chip* (SoC). Este coprocesador cuenta generalmente con sus propios periféricos, y aunque comparte algunos de los recursos del sistema principal, sigue estando aislado del procesador principal. Su uso queda relegado al procesamiento de tareas críticas de seguridad.

- La última configuración disponible es una variante de la anterior, donde el coprocesador se instala de manera externa al SoC, por lo que está completamente aislado de él, sin compartir recursos con el SoC.

1.4 Estructura de la memoria

El presente trabajo se estructura en cinco capítulos que se describen a continuación:

Capítulo 1: Introducción.
En este primer capítulo se presenta una introducción al trabajo, donde se establece la motivación que lo impulsa y se definen los objetivos planteados para la tesis. Además, se proporciona una visión general de la estructura de la memoria.

Capítulo 2: Background.
El segundo capítulo se centra en el marco teórico que fundamenta el trabajo. Se expone el concepto fundamental de Trusted Execution Environment. También se estudia la arquitectura Arm, desde su modelo de negocio hasta la implementación de la arquitectura de seguridad Trustzone. Finalmente se estudian algunas aplicaciones importantes en el contexto de los dispositivos móviles.

Capítulo 3: Estudio de TEE en dispositivos móviles.
En este capítulo se realiza un análisis detallado de las implementaciones de TEE en dispositivos móviles. Se examinan las soluciones ofrecidas por los principales fabricantes para comprender sus características y enfoques en el desarrollo de entornos seguros.

Capítulo 4: Modelo de amenazas.
Este capítulo se centra en el análisis del modelo de amenazas asociado con la seguridad en aplicaciones de banca móvil al utilizar un TEE. Se identifican y analizan las posibles amenazas y vulnerabilidades que pueden afectar a la integridad y confidencialidad de los datos en este contexto.

Capítulo 4: Implementación.
Este capítulo aborda la implementación de una aplicación donde se reflejen las ventajas de utilizar un TEE frente al modelo de amenazas descrito en el apartado anterior.

Capítulo 5: Conclusiones y trabajo futuro.
En este último capítulo se presentan las conclusiones obtenidas a partir del trabajo realizado. Se identifican áreas de mejora y se proponen posibles líneas de investigación futuras que podrían surgir a partir de los resultados de la tesis.

1.5 Convenciones

Durante el trabajo se hará uso de las siguientes convenciones tipográficas:

Itálica
Indica nuevos términos, URLs o extranjerismos. También se utilizará en la expansión de acrónimos.

`Monoespaciado`
Usada en la representación de elementos relacionados con la programación, como pueden ser números en hexadecimal, registros del procesador, instrucciones, etc.

Negrita
Para destacar términos. También se empleará en las listas en las que se definan términos para denotar el término a definir y en las tablas para los campos de cabecera.

MAYÚSCULA
Utilizada para representar acrónimos.

'Comillas simples'
Para indicar que una palabra o expresión se utiliza con un sentido especial.

"Comillas dobles"
Para destacar o encapsular un término específico, indicando que se refiere a un identificador, nombre o cadena de caracteres distintivo en el contexto en el que se presenta, sin implicar un significado metalingüístico especial.

CAPÍTULO 2
Background

Este capítulo explora la teoría fundamental para la seguridad en dispositivos móviles. Se inicia con la investigación del *Trusted Execution Environment* (TEE) y su implementación en estos dispositivos. A continuación, se analiza la arquitectura Arm junto a la tecnología TrustZone, que ofrece el soporte necesario para la implementación de un TEE en procesadores Arm. Finalmente, el capítulo concluye repasando algunas aplicaciones destacadas de TEE en dispositivos móviles. Se profundiza en el mecanismo de arranque seguro, destacando su papel para establecer el TEE como un elemento de confianza. El análisis se extiende a la encriptación de discos en Android, incluyendo un estudio sobre el desbloqueo de dispositivos Android, comentando las implicaciones sobre el cifrado de discos en Android.

2.1 Trusted Execution Environment

Un *Trusted Execution Environment* (TEE) es un entorno de ejecución seguro y aislado que funciona de manera independiente al entorno de ejecución principal o *Rich Execution Environment* (REE) [10] [11]. Cabe resaltar que un TEE no está diseñado para reemplazar el REE ni para gestionar todas sus funcionalidades y aplicaciones. Su principal objetivo es salvaguardar y realizar operaciones críticas, tales como la gestión de claves criptográficas o el manejo de datos confidenciales. Estas operaciones se llevan a cabo en un espacio protegido, seguro y aislado de las aplicaciones y usuarios del REE; esto permite salvaguardarse de posibles ataques dirigidos al REE. Es importante tener en cuenta que este entorno no es intrínsecamente de confianza, ya que ni el software que se ejecuta en él ni el hardware conectado ofrecen más garantías que sus equivalentes en el REE.

Para comprender mejor las diferencias entre un TEE y un REE, es importante examinar sus componentes y características. Los componentes de un TEE son en su mayoría similares a los de un REE, la distinción fundamental radica en la seguridad y simplicidad del TEE. Un TEE incluye un sistema operativo notablemente más simplificado y especializado, diseñado exclusivamente para realizar operaciones críticas y proteger datos sensibles. Estas operaciones son llevadas a cabo por las *Trusted Applications* (TAs), que funcionan en un entorno aislado y seguro, garantizando así la confidencialidad e integridad de los datos.

Además, el TEE incluye un monitor de seguridad que se encarga de conmutar entre el TEE y el REE, asegurando que el contexto de ejecución se guarde y restaure correctamente. Este monitor también garantiza el aislamiento de recursos críticos, controlando el acceso a la memoria y periféricos sensibles, permitiendo que solo las TAs puedan acceder a ellos. Para este propósito, el monitor implementa políticas estrictas de control de acceso, maneja las interrupciones de hardware y gestiona la comunicación segura entre el TEE y el REE.

Durante el arranque del dispositivo, el monitor de seguridad inicializa el TEE configurando las protecciones necesarias para mantener su aislamiento desde el inicio. Todos estos componentes forman la *Trusted Computing Base* (TCB), definida como el conjunto de todos los componentes de un sistema informático críticos para la seguridad [12] (pp. 9-10). El tamaño de la TCB suele medirse en número de líneas de código, número de funciones o número de interfaces externas, y es deseable que sea lo más pequeño posible: a menor tamaño, menor superficie de ataque.

Por otro lado, un REE comprende un sistema operativo completo como Android y las aplicaciones de usuario, conocidas como *Untrusted Applications* (UAs). En este entorno, las aplicaciones y el sistema operativo son más complejos y están más expuestos a posibles vulnerabilidades y ataques. Es importante destacar que los datos manejados dentro del TEE son completamente opacos para el REE. Esto significa que, incluso si se lograra explotar una vulnerabilidad en el sistema operativo del REE, la confidencialidad e integridad de los datos gestionados por el TEE permanecerían intactos [13]. En la siguiente figura se pueden observar las diferencias entre los componentes.

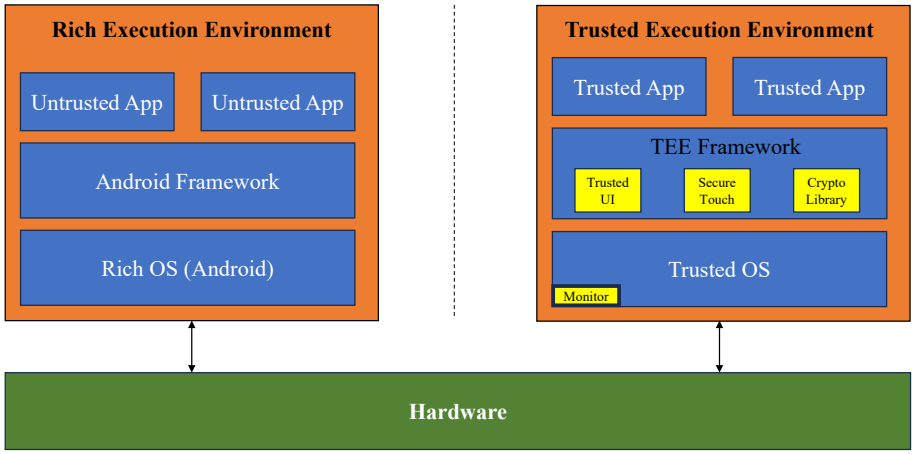


Figura 2.1: Implementación de dos entornos de ejecución separados por hardware.

Desde una perspectiva de hardware, un TEE y un REE también difieren notablemente. El TEE se apoya en componentes de hardware dedicados que proporcionan mecanismos de seguridad avanzados, tales como la criptografía en hardware, áreas de memoria protegida o periféricos de seguridad. En el apartado 2.2.3 se profundiza en la arquitectura de seguridad TrustZone, que ofrece un marco de trabajo para la construcción de un TEE.

La tecnología de los Trusted Execution Environment adquiere especial relevancia en el ámbito de los dispositivos móviles, donde la seguridad de los datos y la protección de la privacidad del usuario son de suma importancia. Los dispositivos móviles modernos albergan un vasto conjunto de información personal y sensible, desde datos financieros hasta información de salud y comunicaciones privadas. En un dispositivo móvil, el TEE permite a los desarrolladores de aplicaciones implementar funcionalidades que requieren un alto nivel de seguridad y protección de datos, como autenticación biométrica, cifrado de datos y gestión de derechos digitales.

Para responder a la diversidad de productos y estrategias de implementación, existen diversas soluciones para TEE en el mercado móvil. Cada fabricante, como Apple o Samsung, sigue una estrategia propia y distintiva en esta área. A continuación, se presenta una tabla que detalla algunas de las soluciones disponibles en el mercado a fecha de redacción del documento.

Compañía	Trusted OS	Configuración	Hardware Utilizado
Apple	iOS Secure Enclave	Coprocesador interno	Apple Secure Enclave
Huawei	iTrustee	Virtualización	Arm TrustZone
Google	Trusty	Coprocesador externo	Google Titan M
Linaro	OPTEE	Virtualización	Arm TrustZone
Qualcomm	QTEE	Virtualización	Arm TrustZone
Samsung	TEEgris	Virtualización	Arm TrustZone
Trustonic	Knibi	Virtualización	Arm TrustZone

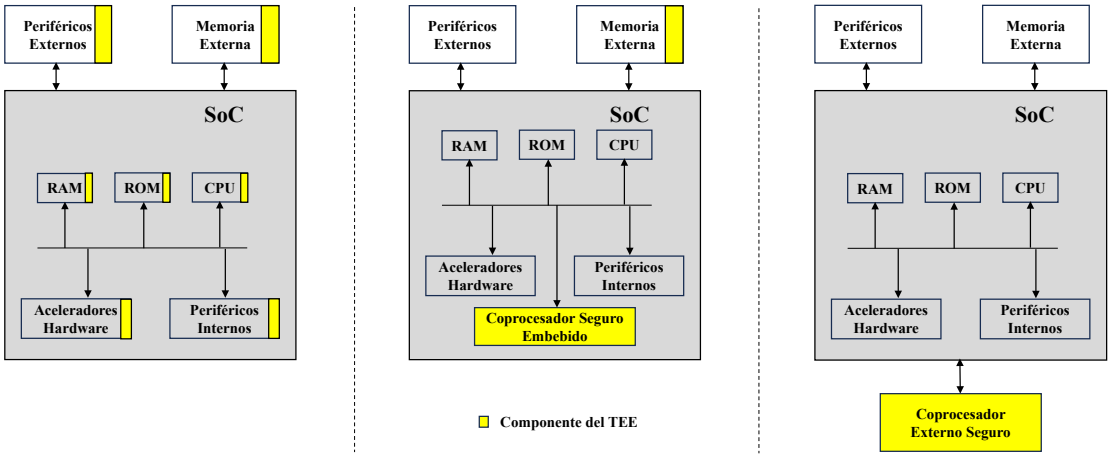
Tabla 2.1: Sistemas operativos seguros clasificados por arquitectura y hardware empleado.

El hardware utilizado en las soluciones de la tabla anterior se puede organizar en tres categorías ordenadas ascendentemente según el nivel de seguridad ofrecido:

1. **Virtualización**: Esta categoría se basa en una configuración que divide un núcleo físico en dos núcleos virtuales mutuamente excluyentes. Cuando uno de los núcleos virtuales está en uso, el otro permanece suspendido.

2. **Coprocesador interno**: En esta configuración, el coprocesador generalmente tiene sus propios periféricos y, aunque comparte algunos recursos con el sistema principal, permanece aislado del procesador principal. Su uso principal se limita al procesamiento de tareas críticas de seguridad. Una ventaja asociada a esta configuración es que, por lo general, las operaciones críticas pueden ejecutarse simultáneamente con el núcleo principal.

3. **Coprocesador externo**: En esta variante, el coprocesador se instala fuera del *System on a Chip* (SoC), asegurando un aislamiento completo del coprocesador con respecto al SoC y evitando cualquier compartición de recursos.



(a) Virtualización. **(b)** Coprocesador interno. **(c)** Coprocesador externo.

Figura 2.2: Comparación de diferentes arquitecturas de seguridad.

En el capítulo 3, se analizan en profundidad las tres configuraciones mencionadas, tomando como referencia las implementaciones realizadas por los actores principales en el sector de los dispositivos móviles: Mediatek, Qualcomm, Apple, Samsung y Google.

2.2 Arquitectura Arm

Arm, anteriormente **Advanced Risc Machines** y originalmente **Acorn RISC Machine**, es una familia de conjuntos de instrucciones RISC. Esta familia se ha convertido en la arquitectura *de facto* para dispositivos que requieran cierta potencia a cambio de un coste energético bajo [14]. De hecho, a fecha de escritura del documento se estima que un 70 % de la población utiliza algún chip basado en la arquitectura Arm [1]. La gran mayoría de sus productos se encuentran en dispositivos móviles, pero también es fácil encontrar electrodomésticos, videoconsolas o sensores IoT que utilizan un SoC de esta arquitectura.

Antes de proseguir, comentar que Arm engloba un conjunto de microarquitecturas clasificadas por familias (p. ej. Arm8, Cortex). Cada una de estas familias, contienen una o más microarquitecturas (p. ej. Armv4, Armv7-A). Finalmente, cada una de estas últimas contienen uno o más núcleos (p. ej. Arm810, Cortex-A7) [2].

En cuanto a su modelo de negocio, Arm Holdings diseña la propiedad intelectual (IP) de los núcleos, que luego serán utilizados para crear chips basados en estos núcleos. Esta IP se vende en forma de diseño junto con un conjunto de herramientas de desarrollo software para que cada empresa interesada pueda fabricar estos chips; también es posible tomar los diseños de Arm como referencia y fabricar chips personalizados.

2.2.1. Juego de instrucciones

Tal y como se ha descrito, existen diferentes familias con diferentes arquitecturas y diferentes núcleos en cada una. Es por eso que dos implementaciones diferentes de una misma arquitectura pueden variar en aspectos como las extensiones que implementa, el tamaño de la caché, el número de fases del *pipeline*, etc. Entre todas las implementaciones se respeta el juego de instrucciones de la arquitectura, conocido como Arm instructions. Este juego de instrucciones parte de un set básico (Armv1) que es soportado en cada nueva arquitectura [15]. En una nueva arquitectura se pueden añadir mejoras, instrucciones nuevas o extensiones nuevas.

Desde el año 1994, todos los procesadores soportan el juego de instrucciones Thumb. Este conjunto de instrucciones cuenta en su haber con las instrucciones más utilizadas del juego de instrucciones Arm, con la diferencia de que el tamaño de una instrucción en este conjunto es de 16 bits. En el año 2003, a partir de la arquitectura Armv4T, se introduce el conjunto de instrucciones **Thumb-2**. Este nuevo conjunto se entiende como un superconjunto de Thumb, añadiendo instrucciones con un tamaño de 32 bits que pueden combinarse en un programa con las instrucciones de 16 bits. Estas nuevas instrucciones de 32 bits permiten a Thumb-2 cubrir la funcionalidad del conjunto de instrucciones Arm. La diferencia más importante entre el conjunto de instrucciones Thumb-2 y Arm es que la mayoría de las instrucciones Thumb de 32 bits son incondicionales, mientras que la mayoría de las instrucciones Arm pueden ser condicionales, motivo por el cual se introduce la instrucción de ejecución condicional `IT` (if-then-else) que permite convertir cualquier operación en una operación condicional [16] (p. 43).

Con el lanzamiento de la arquitectura Armv8, se definen dos nuevos estados de ejecución: **AArch64** y **AArch32** [17] (p. 41). El estado de ejecución AArch64 ofrece las siguientes características:

- 31 registros de propósito general (x0-x30) con un tamaño de 64 bits. Se permite acceder a los 32 bits más bajos del registro empleando el prefijo 'w' (w0-w30).

[1] `https://www.arm.com/company`
[2] `https://en.wikipedia.org/wiki/List_of_ARM_processors`

- Contador de programa (PC), *stack pointers* (SPs) y *Exception Link Registers* (ELRs) de 64 bits.
- Conjunto de instrucciones único, nombrado **A64**. Este conjunto de instrucciones utiliza un tamaño fijo de instrucción de 32 bits.
- Define el modelo de excepciones de Armv8 [18] (pp. 28).
- Soporte para direccionamiento virtual de 64 bits.

Por otro lado, el estado AArch32 ofrece las siguientes características:

- Compatibilidad con las versiones anteriores de la arquitectura Arm.
- Trece registros de propósito general (r0-r12), PC, SP y *Link Register* (LR) con un tamaño de 32 bits. En este estado, el registro LR se puede utilizar como si fuera un ELR.
- Conjuntos de instrucciones **A32** y **T32**. El primero de los conjuntos utiliza un tamaño fijo de instrucción de 32 bits y sería lo que hasta ese momento se conocía como el juego de instrucciones Arm. El segundo de los conjuntos utiliza un tamaño variable de instrucción que puede ser de 16 o 32 bits, y sería lo que hasta ese momento se conocía como el juego de instrucciones Thumb.
- Utiliza el modelo de excepciones de Armv7, que se mapea al modelo implementado en Armv8.
- Soporte para direccionamiento virtual de 32 bits.

También con el lanzamiento de la arquitectura Armv8, se anunciaron tres perfiles de procesadores [3] [17] (p. 40):

- **Application profile (-A)**: Este perfil se dirige a dispositivos de alto rendimiento como PC, servidores o dispositivos móviles. Admiten una *Virtual Memory System Architecture* (VMSA) basada en una *Memory Management Unit* (MMU). Soporta los conjuntos de instrucciones A64, A32 y T32.
- **Real-Time profile (-R)**: Perfil orientado a sistemas de tiempo real como sistemas de control, sistemas de almacenamiento y redes de comunicación. Soporta una *Protected Memory System Architecture* (PMSA) basada en una *Memory Protection Unit* (MPU), aunque también puede implementar una VMSA. Al igual que el perfil anterior, soporta los conjuntos de instrucciones A64, A32 y T32.
- **Microcontroller profile (-M)**: Perfil enfocado en microcontroladores de bajo consumo y bajo costo, como sistemas embebidos en dispositivos IoT. Implementa una variante de la arquitectura de memoria PMSA. El conjunto de instrucciones soportado es una variante del juego de instrucciones T32.

En lo que respecta a las extensiones disponibles, comentar que existen multitud de extensiones para distintos propósitos. Por ejemplo, existen extensiones de virtualización que proporcionan soporte de hardware para virtualizar el estado No seguro del procesador; otro ejemplo serían las Multiprocessing Extensions, que proporcionan un conjunto de

[3] Aunque los perfiles se anunciaron con la arquitectura Armv8, posteriormente se adaptaron para funcionar en versiones anteriores de la arquitectura Arm.

características que mejoran la funcionalidad multiproceso. De entre todas las extensiones disponibles, en este documento solo se van a contemplar las **Security Extensions**. Estas fueron introducidas en la arquitectura Armv6K y engloban un conjunto de características de seguridad hardware para facilitar el desarrollo de aplicaciones seguras [19] (p. 8).

Entre las nuevas características introducidas por estas extensiones, se definen dos estados de seguridad del procesador [20] (pp. 1175-1179): *secure state* (Ss) y *non-secure state* (NSs), que dan soporte a la división de recursos en dos 'mundos': *Normal World* (NWd) para el estado NSs y *Secure World* (SWd) para el estado Ss. El estado en el que se encuentra el procesador es determinado por el bit `NS` del registro `SCR` (*Secure Configuration Register*) 2.3. Si este bit adopta el valor `0` el estado del procesador es **Ss**; por el contrario, si su valor es `1` el estado del procesador es **NSs**. El registro `SCR`, únicamente se puede modificar desde el estado **Ss**.

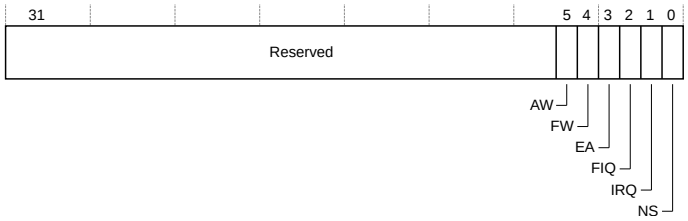


Figura 2.3: Bits del registro `SCR`. Adaptado de [21].

Independientemente del estado en el que esté el procesador, se cumplen las siguientes características:

- Los dos estados de seguridad operan en su propio espacio de direcciones virtuales.
- Se pueden definir controles del sistema de manera independiente en cada uno de los estados.
- Todos los modos del procesador que están disponibles en un sistema que no implementa las extensiones de seguridad, están disponibles en los dos estados.

Por lo que se refiere al juego de instrucciones, las Security Extensions añaden una nueva instrucción cuyo mnemónico es `SMI` (*Software Monitor Instruction*) [4]. También se ven modificadas las siguientes instrucciones [22] (pp. 45-46):

- `CPS`.
- `LDM(3)`.
- `MSR`.
- `RFE`.
- `SRS`.
- Operaciones de datos de configuración de *flags* (incluyendo `MOV`) que escriben en el `PC`, como por ejemplo `ADDS PC,Rn,Rm`.

[4] En las nuevas versiones de la arquitectura (v7+), se ha modificado el mnemónico a `SMC` (*Secure Monitor Call*).

Las Security Extensions introducen también un nuevo modo del procesador llamado **monitor mode**. Este modo se utiliza para gestionar los cambios entre los dos estados y tiene la característica que independientemente del valor del bit `NS`, en este modo el estado del procesador es siempre **Ss**. Solo es posible entrar en este modo desde el estado NSs si se generan alguna de las siguientes excepciones: una interrupción, una excepción del tipo *External Abort* o una llamada explícita con la instrucción `SMC` [23] (p. 73).

En cuanto al sistema de memoria, se proporcionan dos MMUs virtuales, una para cada mundo, además de modificar la *Translation Lookaside Buffer* (TLB) y las memorias caché. En las entradas de la TLB, se añade un nuevo bit denominado `NSTID` (*Non-Secure Table ID*). Este bit determina si la entrada correspondiente está asociada a una tabla de páginas del estado Ss o NSs. En la memoria caché, las líneas se etiquetan con una etiqueta adicional llamada *Secure line tag*. Esta etiqueta indica si la línea contiene datos del estado Ss o NSs [22] (pp. 68-78).

2.2.2. Modelo de excepciones de Armv8

El modelo de excepciones en la arquitectura Armv8 presenta una estructura basada en los niveles de privilegio conocidos como *Exception Level* (EL), que van desde EL0 hasta EL3. Estos niveles definen diferentes niveles de acceso y control en el sistema, siendo EL3 el más alto y EL0 el más bajo. Aunque la arquitectura no especifica directamente qué software se ejecuta en cada nivel [24] (pp. 08-09), es común asociar ciertos tipos de software con cada nivel de privilegio para garantizar un adecuado control y gestión del sistema. Un uso común del tipo de software ejecutado en cada nivel se muestra en la siguiente figura.

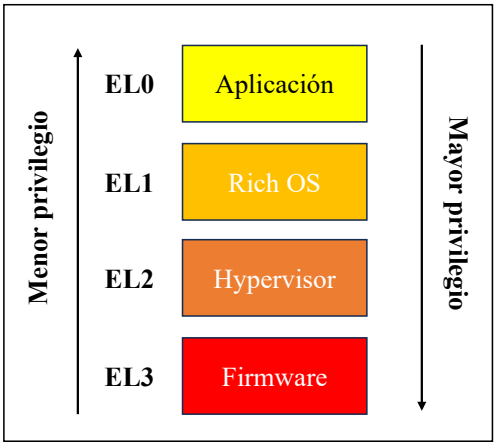


Figura 2.4: Modelo típico de uso de software dependiendo del nivel de privilegio.

Empleando el modelo anterior junto con los dos estados de seguridad proporcionados por las Security Extensions, se adjunta una tabla con el software que se ejecutaría en cada nivel y estado de seguridad.

	Estado	
Nivel de excepción	**Non-secure**	**Secure**
EL0	Aplicación de usuario	Trusted Application
EL1	Rich OS	Trusted OS
EL2	Hypervisor	Secure partition manager
EL3 [5]	Firmware / Secure monitor	

Tabla 2.2: Software previsto para ejecutarse según la el nivel de excepción del procesador.

Tomando como referencia la tabla anterior, para realizar un cambio de estado es necesario que el software que esté en ejecución se encuentre en el nivel de excepción EL1 o superior [25]. Este software ejecutará la instrucción `SMC` o utilizará una excepción FIQ que será atendida por el software del monitor seguro. El monitor seguro entonces, guardará el estado de ejecución actual, para posteriormente restaurarlo una vez se finalice la operación. En la siguiente figura se resume este proceso.

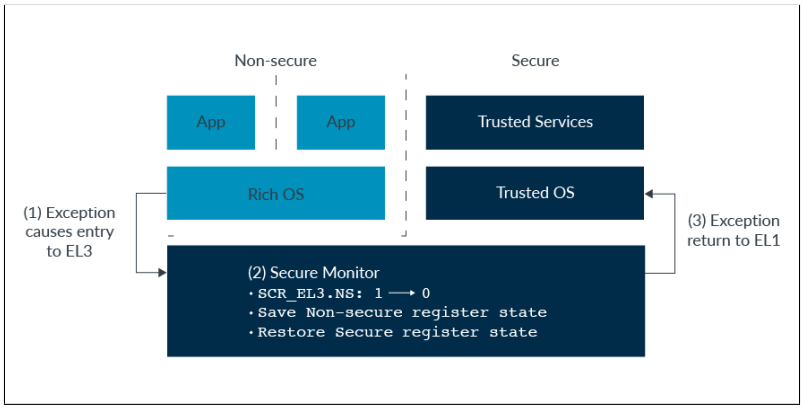


Figura 2.5: Cambio de mundo en procesadores Arm. Adaptado de [19].

2.2.3. Arm Trustzone

TrustZone es el nombre de la arquitectura de seguridad para los procesadores Arm. Su objetivo es ofrecer un marco de trabajo para la construcción de un entorno programable que permite la confidencialidad e integridad de casi cualquier activo ante ataques específicos [23] (p. 34). Actualmente existen implementaciones para los procesadores de perfil -A y -M [26]. En los procesadores de perfil -A, TrustZone se utiliza típicamente para garantizar la seguridad en sistemas operativos completos, mientras que en los procesadores de perfil -M, se enfoca en la protección de aplicaciones y datos críticos en dispositivos de menor potencia y recursos.

TrustZone utiliza el bus de sistema AMBA AXI como parte fundamental de su infraestructura de seguridad. El protocolo AXI proporciona señales de acceso como `AWPROT` y `ARPROT`, las cuales, junto con los tres niveles de protección de acceso `AxPROT`, determinan el tipo de acceso y su viabilidad [27] (pp. 82-83). Las asignaciones de bits `AxPROT` especifican los siguientes atributos:

[5] En este nivel de excepción, el estado del procesador es siempre **Ss**.

- **AxPROT[0]**: Un valor de `1` en este bit indica que el acceso es privilegiado. Por el contrario, un valor de `0` indica que el acceso es no privilegiado.

- **AxPROT[1]**: Un valor de `1` indica que la transacción es no segura. Por otro lado, un valor de `0` indica que la transacción es segura.

- **AxPROT[2]**: Un valor de `1` en este bit indica que el acceso es a instrucciones. Mientras que, un valor de `0` indica que el acceso es a datos.

La especificación AMBA incluye también un bus periférico de bajo consumo y bajo ancho de banda conocido como *Advanced Peripheral Bus* (APB), el cual está conectado al bus del sistema mediante un puente AXI-to-APB. Este bus permite la securización de periféricos como controladores de interrupciones, temporizadores y dispositivos de E/S.

Haciendo uso de las señales anteriores, existen tres componentes principales que se encargan de gestionar la seguridad en el entorno TrustZone [28] (pp. 4-5): *TrustZone Protection Controller* (TZPC) y *TrustZone Memory Adapter* (TZMA), conectados al puente AXI-to-APB, y *TrustZone Address Space Controller* (TZASC), conectado al bus AXI. Es importante tener en cuenta que el uso de TZASC, TZMA y TZPC es opcional dentro de la especificación de TrustZone y puede variar según la implementación del SoC.

El controlador TZASC se dedica a salvaguardar tanto la memoria DRAM como los dispositivos mapeados en la memoria, administrando el acceso y los permisos de lectura y escritura para diversas áreas de memoria. Para cumplir con esta tarea, TZASC divide la memoria en regiones designadas como seguras o no seguras, protegiendo las regiones marcadas como seguras de accesos desde el estado NSs. Por otro lado, el controlador TZMA se enfoca en la memoria ROM y SRAM, implementando un esquema similar al de TZASC. Por último, el controlador TZPC aprovecha las características de TZMA para proteger los periféricos internos del chip, estableciendo el acceso para operaciones en el estado NSs, Ss o ambos. Es importante destacar que el controlador TZPC debe ser configurado desde el SWd. Un ejemplo práctico de este controlador podría ser la introducción de un PIN para autorizar un pago. En esta situación, TZPC reconfiguraría la pantalla y el sensor táctil para que no sean accesibles desde el NWd. En la siguiente figura se puede ver un ejemplo SoC que incorpora estos controladores y su interconexión.

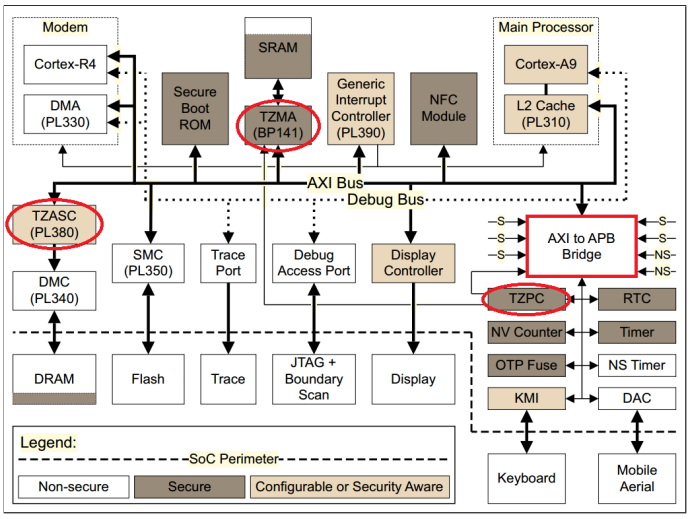


Figura 2.6: Implementación de los controladores de TrustZone en un SoC. Adaptado de [23].

2.3 Verified Boot

El arranque verificado se define como una secuencia de arranque en la que cada imagen de software que se va a ejecutar en el dispositivo está autenticada por un software previamente verificado. Esta secuencia sirve para prevenir la ejecución de código no autorizado o modificado, asegurando que todo el código es comprobado antes de ser ejecutado. La secuencia de arranque conforma una cadena de confianza: cada fase en el arranque sirve como base de confianza para la carga de la siguiente fase [29]. En la actualidad, la secuencia de arranque varía según el SoC instalado y la marca del dispositivo, aunque existen ciertos elementos que son comunes entre las diferentes implementaciones [30] (p. 8).

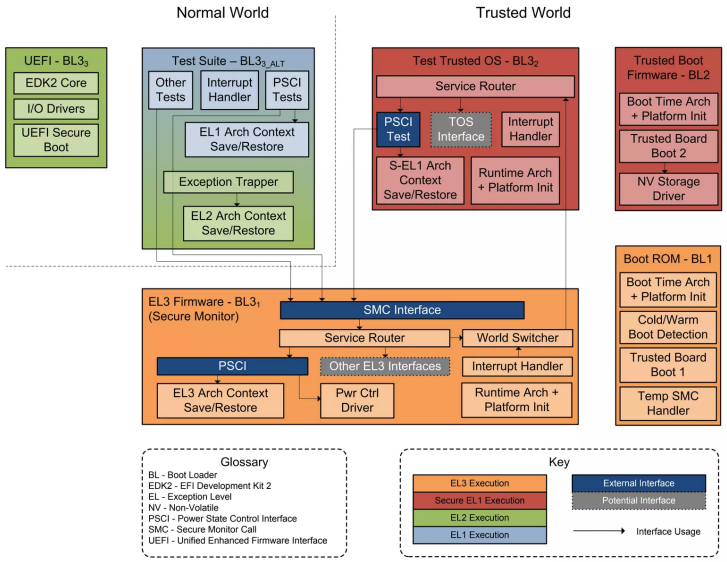


Figura 2.7: Fases comunes del arranque verificado. Adaptado de [31].

En la figura anterior, se puede observar que existen diferentes fases diferenciadas, cada una con un propósito diferente. Durante la primera fase (BL1), es indispensable que el código a ejecutar cuente con un estado implícito de confianza, pues de él depende que se arranque el resto del sistema. Para satisfacer este requisito de seguridad, el código de esta fase, también conocido como *Primary Boot Loader* (PBL), viene pregrabado por el fabricante durante el proceso de manufacturación. Las funciones que desempeña este *bootloader* consisten en realizar procesos básicos como *power-on self-test* o configurar los registros de control. Por último, se verificará y cargará el *Secondary Boot Loader* (SBL), que se ejecutará en la siguiente fase. Al tratarse del primer elemento en la cadena de arranque, el PBL actúa como raíz de confianza del proceso de arranque [32] [33].

Durante la siguiente fase del arranque, el SBL realiza acciones un poco más específicas como configurar el almacenamiento no volátil, configurar el mecanismo de traducción de páginas para el nivel de privilegio S-EL1 y cargar todas las imágenes del tercer nivel (BL3-X).

En este punto, el sistema ejecuta la imagen BL3-1. Este bootloader tiene la responsabilidad de cargar el firmware de EL3 y del monitor de seguridad, además de algunos componentes relacionados con el SWd, como los manejadores de excepciones o los controladores de memoria de TrustZone. BL3-1 ejecutará a continuación dos subfases en paralelo. La primera fase (BL3-2) se ejecuta en el nivel de excepción S-EL1 y su función principal es cargar el bootloader del TEE [34] (p. 3). La otra fase (BL3-3) se ejecuta en el nivel de privilegio EL1 y tiene como objetivo cargar la imagen del firmware no confiable.

Después de que se haya cargado la imagen de UEFI, se carga *Android Boot Loader* (ABL). ABL es el encargado de iniciar el sistema operativo Android en el dispositivo. Una de las funciones clave de ABL es gestionar los modos de arranque especiales, como el modo `recovery` y el modo `fastboot`. En última instancia, una vez que ABL ha completado su tarea, procederá a cargar el kernel de Linux. Tras cargar el kernel de Linux, se inicia el proceso de arranque completo de Android, permitiendo al usuario acceder a todas las funciones y aplicaciones del dispositivo.

Para explicar con más profundidad el proceso de verificación de imágenes durante el arranque, se utilizará la implementación que sigue el fabricante Qualcomm [35]. En un SoC Qualcomm, se siguen los siguientes pasos para verificar una imagen:

1. El cargador reserva tres áreas de memoria para alojar la cabecera ELF de la imagen, la *program header* y el segmento hash, los cuales se leen de un archivo con extensión `mdt`. La program header contiene el código a ejecutar si la verificación tiene éxito. El segmento hash contiene su propia cabecera que especifica el tamaño de todo el segmento hash, el tamaño de la tabla de hashes, el tamaño de la firma de atestación y el tamaño de la cadena de certificados. La tabla hash contenida en el segmento alberga el hash SHA-256 de cada segmento de la imagen ELF, así como de la cabecera ELF y la program header. La firma de la imagen (*signature* en la figura 2.8) se calcula sobre esta tabla de hashes y se añade a ella, junto con la cadena de certificados, para formar el segmento hash. La entrada de la tabla hash correspondiente al propio segmento hash está vacía, ya que todo el segmento hash se autentica durante la verificación de la firma.

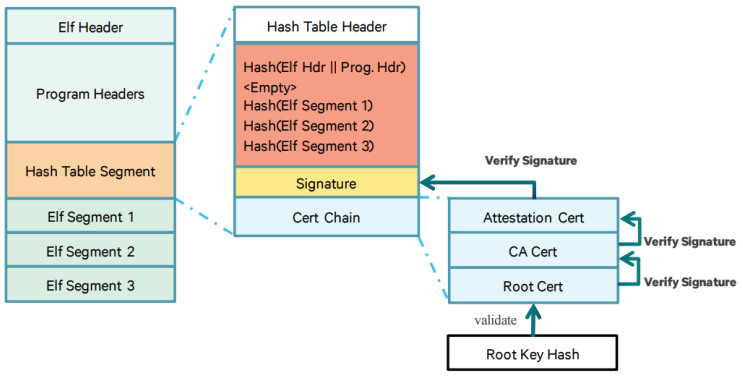


Figura 2.8: Contenido del segmento hash. Adaptado de [36].

2. El cargador verifica la firma del segmento hash validando la cadena de certificados. Qualcomm permite que la cadena de certificados esté formada por dos o tres certificados, donde se incluye opcionalmente el certificado *Attestation CA*. Además, cabe destacar que, si la imagen a cargar ha sido firmada tanto por Qualcomm como por el fabricante del dispositivo, entonces cada cadena de certificados debe verificarse de manera independiente: por un lado se verificará la cadena de certificados de Qualcomm y por otro lado la del fabricante del dispositivo. La validación de una cadena de certificados se observa en la figura 2.9 y se realiza como sigue:

   2.1 El cargador realizará un hash al certificado Root CA. Este primer hash se compara con el valor contenido en un *One-Time-Programmable eFuse* o en la ROM del dispositivo, grabado durante la producción del dispositivo. Si ambos valores coinciden, se procede a verificar los siguientes certificados de la cadena [36].

2.2 A continuación, se valida el certificado Attestation CA. Para realizar esta validación, se sigue el mismo procedimiento que en el paso anterior, solo que ahora el hash del certificado se compara con el campo signature incluido en el propio certificado. Este campo contiene el hash del certificado Attestation CA cifrado con la clave pública del certificado Root CA. El hash contenido en el campo signature se descifra utilizando la clave pública del certificado Root CA para comparar su valor con el obtenido al realizar el hash al certificado, si ambos valores coinciden se procede a comprobar el último certificado.

2.3 El último certificado a validar es el certificado Attestation Certificate. El proceso de validación de este certificado sigue el mismo procedimiento que el certificado anterior, solo que esta vez el campo signature se habrá cifrado con la clave pública del certificado Attestation CA. Si al descifrar el valor del campo signature coincide con el obtenido al realizar el hash del certificado, se habrá validado correctamente la cadena de certificados.

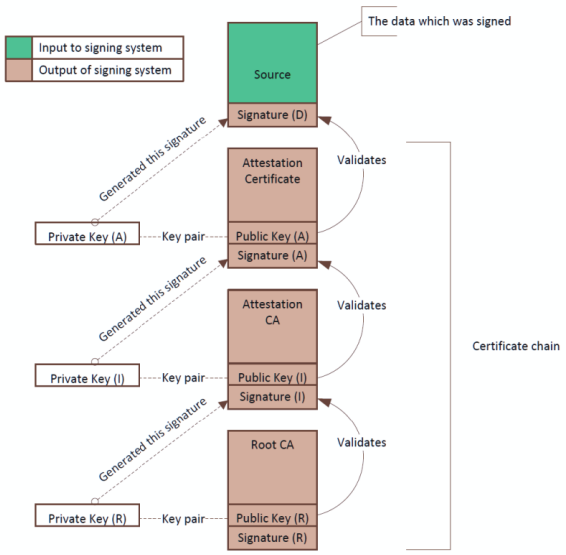


Figura 2.9: Validación de certificados en el arranque de Qualcomm. Adaptado de [36].

3. El cargador valida la cabecera ELF y la program header cargadas en el paso anterior, realizando un hash de ambos elementos y comparándolo con el primer elemento de la tabla hash. Si los hashes no coinciden, la imagen es rechazada.

4. A continuación, el cargador carga los segmentos restantes de la imagen, los cuales se encuentran en archivos con extensión `bxx` ($x \in \{0, 9\}$), donde cada archivo se corresponde con un segmento. Estos segmentos se cargan en un área aprobada por el cargador para tal fin. Si no es posible cargar algún segmento, la imagen se rechaza.

5. El cargador valida los segmentos restantes realizando un hash de cada segmento y comparándolo con su entrada correspondiente en la tabla hash. Si algún hash no coincide, la imagen es rechazada.

2.4 Desbloqueo dispositivos Android

Un dispositivo Android ofrece diversas opciones para establecer una pantalla de bloqueo y así salvaguardar la información personal contra accesos no autorizados. A continuación, se detallan cada una de ellas:

- **'Desliza para desbloquear'**: Este método carece de protección, ya que simplemente implica deslizar el dedo de un extremo a otro de la pantalla.
- **Patrón**: Se dibuja un patrón sobre una matriz de 3x3.
- **PIN**: Código numérico de cuatro a dieciséis dígitos.
- **Contraseña**: Código alfanumérico de cuatro a dieciséis caracteres, considerada como la opción más segura.
- **Desbloqueo facial**: Introducido en la versión 4.0 de Android [6], utiliza la cámara frontal para comparar el aspecto del usuario con una imagen de su rostro previamente registrada. Para poder emplear este método, previamente se requiere el establecimiento de un PIN, patrón o contraseña. La necesidad de contar con un primer método de autenticación no biométrico para desbloquear el dispositivo, se debe a que el resultado de introducir el PIN, patrón o contraseña siempre produce el mismo resultado, a diferencia de los datos biométricos. Cuando por ejemplo se utiliza el desbloqueo facial, este compara dos imágenes: la registrada y la captada. Sin embargo, factores como la luminosidad pueden afectar la captura de la imagen, lo que puede resultar en que el mismo usuario que registró el rostro no pudiera pasar de la pantalla de bloqueo.
- **Desbloqueo mediante huella dactilar**: Autenticación biométrica oficialmente introducida en la versión 6.0 de Android [7], considerada más segura que el desbloqueo facial. Utiliza un lector de huellas para comparar la huella del usuario con una registrada previamente. Al igual que con el desbloqueo facial, se requiere establecer previamente un PIN, patrón o contraseña.

2.4.1. Enroll

Cuando se enciende el dispositivo después de un restablecimiento de fábrica, todos los autenticadores (`Gatekeeper`, `Fingerprint` y `Face`) están listos para registrar credenciales de usuario. En esta etapa inicial, el usuario debe establecer un PIN, patrón o contraseña. Este registro crea un identificador seguro de usuario (SID) de 64 bits generado aleatoriamente que sirve como identificador para el usuario y como token vinculante para el material criptográfico de este [8]. Cabe mencionar que este SID es la representación TEE de un usuario sin una conexión sólida con un ID de usuario de Android.

Cuando un usuario desea cambiar una credencial, debe presentar una credencial existente. Si esta credencial existente se verifica correctamente, el SID asociado a ella se transfiere a la nueva credencial, lo que permite al usuario seguir accediendo a las claves después de cambiar una credencial. Este proceso se conoce como inscripción confiable. Sin embargo, si el usuario no presenta una credencial existente, la nueva credencial se registra con un SID de usuario completamente aleatorio. Esto implica que, aunque el usuario aún

[6] `https://developer.android.com/about/versions/android-4.0-highlights`
[7] `https://www.android.com/intl/en_us/versions/marshmallow-6-0/`
[8] `https://source.android.com/docs/security/features/authentication`

puede acceder al dispositivo, las claves asociadas al SID de usuario anterior se pierden de forma permanente. Este proceso se conoce como inscripción no confiable.

Una vez que se ha establecido un PIN, patrón o contraseña, se genera un *password handle*. Este handle contiene el producto de aplicar un HMAC al SID del usuario y la contraseña establecida, y queda almacenado en el gestor de contraseñas. Este handle se empleará tanto si el usuario decide configurar un segundo método de autenticación, como para autenticar al usuario frente a la pantalla de bloqueo [9].

2.4.2. Autenticación

Una vez que se ha establecido la credencial de acceso, el sistema puede llevar a cabo un proceso de autenticación para permitir o no al usuario realizar acciones. Por ejemplo, si el teléfono se encuentra bloqueado, la autenticación servirá para desbloquearlo. Otro uso común de autenticación sería el acceso a una aplicación bancaria mediante huella dactilar. Aunque el proceso subyacente es exactamente igual para los casos descritos, a continuación, se explica el proceso de autenticación para desbloquear el dispositivo.

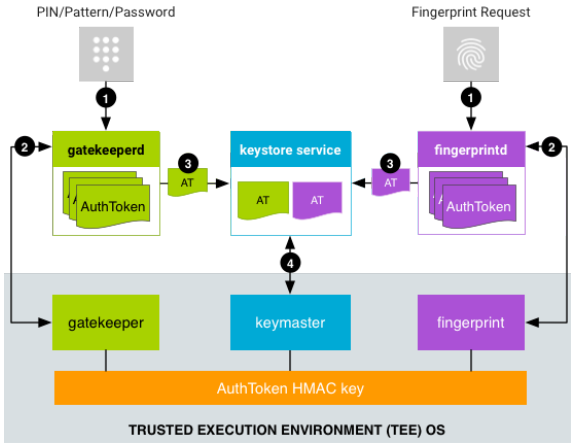


Figura 2.10: Proceso de autenticación biométrico en Android. Adaptado de [8].

1. El usuario escoge un método de autenticación e introduce las credenciales.

2. La pantalla de bloqueo realiza una solicitud al *daemon* correspondiente. Para PIN, patrón y contraseña se realiza una solicitud a `gatekeeperd`, para la autenticación mediante huella dactilar la solicitud se dirige a `fingerprintd` y si se utiliza el desbloqueo facial la solicitud irá dirigida a `faced`.

3. Dependiendo del método de autenticación elegido, en el paso anterior se invocará un daemon específico. Al final, se generará un token de autenticación cuyo formato se discute en C.

    - `gatekeeperd` enviará el hash del PIN, patrón o contraseña al trustlet `Gatekeeper` [10]. Si la autenticación en el TEE resulta exitosa, `Gatekeeper` devuelve un token de autenticación firmado con la clave *AuthToken HMAC key* que contiene el SID de usuario correspondiente[11]

[9] `https://cs.android.com/android/platform/superproject/main/+/main:prebuilts/vndk/v34/arm/include/hardware/libhardware/include/hardware/gatekeeper.h;l=49`

[10] `https://source.android.com/docs/security/features/authentication/gatekeeper`

[11] La clave AuthToken HMAC key se genera en cada reinicio del dispositivo y es compartida por todos los componentes del TEE (Gatekeeper, Keymaster y trustlets biométricos compatibles).

- `fingerprintd` enviará la lectura de la huella dactilar al trustlet `Fingerprint` [12]. `Fingerprint` devolverá un token de autenticación firmado con la clave AuthToken HMAC key que contiene el SID de usuario correspondiente, si la comparación resultase exitosa.
- Para cualquier otro método de autenticación biométrica, el daemon biométrico apropiado permanecerá a la escucha de eventos biométricos para transmitirlos al trustlet biométrico apropiado.

4. El daemon recibe el token firmado y se lo proporcionan al servicio `keystore`.
   - `gatekeeperd` también notificaría a `keystore` cuando el dispositivo se vuelve a bloquear y cuando se cambia la contraseña del dispositivo.
5. `keystore` proporciona el token al trustlet `KeyMaster` para su verificación, utilizando la clave compartida AuthToken HMAC. `KeyMaster` confía en la marca de tiempo del token como la última hora de autenticación y toma una decisión sobre la liberación de la clave basada en este valor para permitir que una aplicación utilice la clave.

2.5 Android disk encryption

2.5.1. Full-Disk Encryption

Android Full-Disk Encryption (FDE) es una funcionalidad de seguridad introducida en la versión 3.0 de Android [13], el cual ha estado disponible hasta la versión 9 [14] del SO, con mejoras significativas en las versiones **4.4** y **5.0** [37]. Este método de cifrado implica que, tras el inicio del dispositivo, el usuario debe proporcionar las credenciales establecidas antes de poder acceder a cualquier parte del disco. Esto significa que funciones como alarmas, servicios de accesibilidad o la recepción de llamadas no estarán disponibles hasta que el usuario introduzca dichas credenciales.

FDE utiliza el subsistema **dm-crypt** para implementar el cifrado transparente del disco. Una vez activada la encriptación, todas las escrituras en disco encriptan automáticamente los datos antes de guardarlos y todas las lecturas desencriptan automáticamente los datos antes de devolverlos. La clave utilizada en la encriptación del disco se genera aleatoriamente, y tiene un tamaño de 128 bits. En la literatura esta clave es referida como *master key* o *Device Encryption Key* (DEK) [38]. Esta clave permanece invariable a menos que se elimine la partición `/data` o se realice un restablecimiento de fábrica.

En la primera iteración de FDE (disponible desde la versión 3.0 de Android hasta la versión 4.3), se cifraban todos los sectores de la memoria eMMC del dispositivo. El algoritmo de cifrado utilizado es AES-CBC. AES-CBC necesita vectores de inicialización (IV) para cifrar cada bloque de forma independiente, de modo que el resultado obtenido al cifrar dos bloques cualesquiera no coincide, incluso habiendo usado la misma clave. Para generar un IV distinto para cada bloque, se utiliza el método *encrypted salt-sector initialization vector* (ESSIV) junto con el algoritmo de hash SHA-256.

[12] `https://source.android.com/docs/security/features/authentication/fingerprint-hal`

[13] `https://source.android.com/docs/security/features/encryption/full-disk`

[14] Solo los dispositivos que se iniciaron con Android 9 o inferior pueden usar cifrado de disco completo. Android 10-12 admite el cifrado de disco completo solo para dispositivos que se actualizaron desde una versión anterior de Android. Android 13 elimina por completo la compatibilidad con el cifrado de disco completo.

Los parámetros de encriptación se almacenan en un área no encriptada del dispositivo conocida como *crypto footer*, la cual se corresponde con la partición `/metadata`. Aunque esta estructura comparte similitudes con la cabecera de *Linux Unified Key Setup* (LUKS), es significativamente más sencilla, ya que omite diversas características presentes en LUKS. Por ejemplo, mientras que LUKS permite el uso de múltiples *key slots* para desbloquear el disco mediante diversas claves, crypto footer solo guarda una copia encriptada de la *Device Encryption Key* (DEK).

En esta primera versión de FDE, la estructura se compone de los siguientes campos:

```
struct crypt_mnt_ftr {
    __le32 magic;
    __le16 major_version; // Version mayor FDE: 1
    __le16 minor_version; // Version menor FDE: 0
    __le32 ftr_size;
    __le32 flags;
    __le32 keysize; // Tamaño de la clave (128 bits)
    __le32 spare1;
    __le64 fs_size;
    __le32 failed_decrypt_count;
    unsigned char crypto_type_name[MAX_CRYPTO_TYPE_NAME_LEN]; // aes-cbc-essiv:sha256
};
```

Listing 2.1: Formato de la estructura crypto footer.

Como se puede apreciar, la estructura abarca información como la versión de FDE, el tamaño de la clave, y una cadena de texto que especifica el modo de cifrado (aes-cbc-essiv:sha256), entre otros atributos. En esta versión, ciertos parámetros como la DEK, no se han incorporado explícitamente en la crypto footer, dado que se consideran implícitos. Debido a esta consideración, la estructura se guarda junto con una copia encriptada de la clave DEK y un valor aleatorio de 128 bits (*salt*), utilizado para desencriptar la DEK. En el disco, toda esta información se almacena de la forma que se especifica en la siguiente figura:

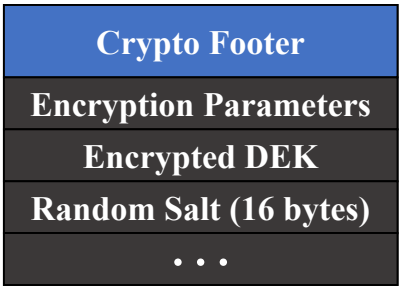


Figura 2.11: Versión primeriza de la crypto structure.

Para almacenar la copia de la DEK, se utiliza una clave de 128 bits conocida como *Key Encryption Key* (KEK), la cual se deriva del PIN o contraseña del dispositivo establecida por el usuario utilizando 2000 iteraciones de PBKDF2. El uso de esta clave, permite cambiar el PIN o contraseña sin la necesidad de volver a cifrar el dispositivo con una nueva clave. Cuando se deriva esta clave, también se genera un IV que se utiliza para cifrar la DEK utilizando el algoritmo AES-CBC. Al arrancar un dispositivo cifrado, Android toma el PIN o contraseña que el usuario introduce, lo procesa utilizando la función PBKDF2 y finalmente descifra la clave DEK, que se utilizará para montar la partición de datos del usuario cifrada. Este proceso puede observarse en la siguiente figura.

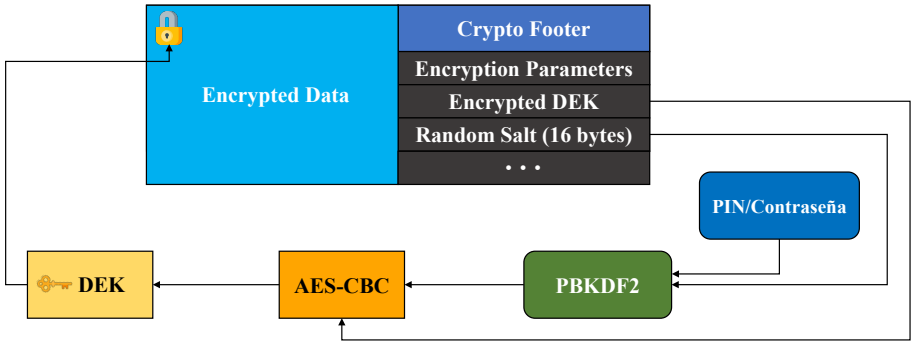


Figura 2.12: Descifrado de la clave DEK en FDE.

El esquema de encriptación presentado en esta primera versión se considera relativamente seguro, ya que está implementado completamente en software. Su seguridad dependerá de la calidad de la clave de encriptación. Dado que en Android se reutiliza el PIN o contraseña del dispositivo (con una longitud máxima de 16 caracteres), en la práctica, la mayoría de los usuarios tiende a establecer una contraseña débil. Para dotar de mayor robustez al esquema, se seleccionó el algoritmo de derivación de claves PBKDF2 para generar la KEK. Este algoritmo fue diseñado para manejar entradas de baja entropía y requiere un costo computacional significativo para resistir ataques de fuerza bruta. En la época en que se implementó esta versión de FDE, las 2000 iteraciones utilizadas para la generación de la KEK no eran un obstáculo para realizar ataques de fuerza bruta, incluso con el uso de tarjetas gráficas [39].

En la siguiente iteración de FDE (implementada en la versión 4.4 de Android), el cambio más destacado consiste en la sustitución de la función de derivación de claves PBKDF2 por scrypt [15]. La concepción de scrypt se orienta específicamente a resistir ataques de fuerza bruta mediante el uso de hardware altamente paralelizable, como las tarjetas gráficas. La limitada capacidad de memoria en las tarjetas gráficas impide la ejecución simultánea de múltiples tareas scrypt, ya que cada una demanda una cantidad considerable de memoria [40].

Dentro del procedimiento de actualización a la versión 4.4, Android realiza automáticamente la actualización de la estructura crypto footer para incorporar el uso de scrypt y procede a volver a cifrar la clave maestra. En consecuencia, todos los dispositivos que ejecuten Android 4.4 (exceptuando aquellos que utilicen un esquema FDE propietario del proveedor) deben contar con su clave DEK protegida mediante una clave derivada de scrypt.

[15] `https://www.tarsnap.com/scrypt.html`

En esta iteración de FDE, la crypto footer incorpora nuevos atributos, como se puede apreciar a continuación:

```
struct crypt_mnt_ftr {
    __le32 magic;
    __le16 major_version; // Version mayor FDE: 1
    __le16 minor_version; // Version menor FDE: 2
    __le32 ftr_size;
    __le32 flags;
    __le32 keysize; // Tamaño de la clave (128 bits)
    __le32 spare1;
    __le64 fs_size;
    __le32 failed_decrypt_count;
    unsigned char crypto_type_name[MAX_CRYPTO_TYPE_NAME_LEN]; // aes-cbc-essiv:sha256
    __le32 spare2;
    unsigned char master_key[MAX_KEY_LEN]; // DEK encriptada
    unsigned char salt[SALT_LEN]; // Salt aleatorio
    __le64 persist_data_offset[2];
    __le32 persist_data_size;
    __le8 kdf_type;
    /* Parametros configurables de scrypt */
    __le8 N_factor;
    __le8 r_factor;
    __le8 p_factor;
};
```

Listing 2.2: Formato actualizado de la estructura crypto footer.

En esta versión actualizada, se incorpora el campo `kdf_type`, que especifica la función KDF utilizada para derivar la clave KEK, así como la clave DEK encriptada o el salt aleatorio. Además, se incluyen los valores de los parámetros de inicialización de scrypt (`N_factor`, `p_factor` y `p_factor`). El tamaño de la clave y el modo de encriptación (aes-cbc-essiv:sha256) se mantienen inalterados en comparación con la versión anterior. Como se puede observar en la siguiente figura, ahora la estructura crypto footer si engloba a la DEK y el salt aleatorio.

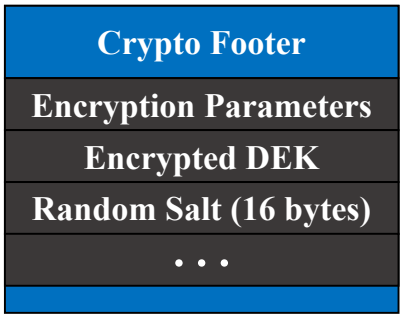


Figura 2.13: Nueva crypto structure.

La última iteración de FDE fue introducida en la versión 5.0 de Android, incorporando las siguientes novedades [16]:

- Cifrado rápido, donde solo se cifran los bloques usados en la partición de datos, para evitar que el primer arranque demore mucho tiempo.
- Se agrega soporte para patrones y cifrado sin contraseña.
- Utilización de almacenamiento respaldado por hardware para la clave de cifrado, utilizando la capacidad de firma de TEE.

[16] `https://source.android.com/docs/security/features/encryption/full-disk`

Al incorporar un TEE en la ecuación, ahora la DEK quedará asociada al dispositivo donde fue generada. Por ende, al obtener una copia de la estructura crypto footer y de la partición encriptada `/data`, resulta infructuoso intentar desbloquear la partición fuera del dispositivo.

`KeyMaster` es el trustlet responsable de vincular la clave DEK al dispositivo. `KeyMaster` puede emplearse para generar claves de cifrado y llevar a cabo operaciones criptográficas sin revelar las claves al NWd [41]. Al generar una clave en `KeyMaster`, esta se cifra mediante una clave de cifrado respaldada por el hardware del TEE, creando un *key blob* que será devuelto al NWd. Cuando una aplicación del NWd desea efectuar una operación utilizando la clave generada por `KeyMaster`, debe suministrar el key blob obtenido previamente al trustlet correspondiente, a modo de autenticación. El trustlet puede entonces, descifrar la clave almacenada, utilizarla para realizar la operación criptográfica deseada y devolver el resultado al NWd.

Atendiendo a este nuevo modo de funcionamiento, se presenta a continuación el actualizado proceso de generación de claves en el esquema FDE, que puede encontrarse resumido en la figura 2.14:

1. Se genera una clave DEK aleatoria de 128 bits, junto con un salt también aleatorio de 128 bits.
2. Se aplica scrypt a la contraseña establecida por el usuario y al salt para obtener una clave intermedia (IK1) de 256 bits.
3. Se *paddea* la clave IK1 con el valor `0x0`, hasta alcanzar los 2048 bits de la *Hardware-bound private key* (HBK). El padding se realiza de la siguiente manera:
   - 3.1 Se añade al inicio 8 bits con el valor `0x0`.
   - 3.2 Se incorporan los 256 bits de la clave IK1.
   - 3.3 Se concatenan 1784 bits con el valor `0x0`.
4. Una vez obtenidos los 2048 bits, se firman utilizando la clave privada del TEE (HBK) para producir una clave intermedia (IK2) de 2048 bits también. La generación de esta clave intermedia puede verse como el paso que posteriormente acabará vinculando la DEK al dispositivo.
5. Se aplica una segunda vez scrypt, pero esta vez a la clave intermedia producida en el paso anterior y empleando el mismo salt del paso 2 para obtener una tercera clave intermedia (IK3) de 256 bits.
6. Finalmente, se encripta la clave DEK (generada aleatoriamente) mediante AES-CBC. Como clave se utilizarán los primeros 128 bits de la clave IK3 y como IV los últimos 128 bits.

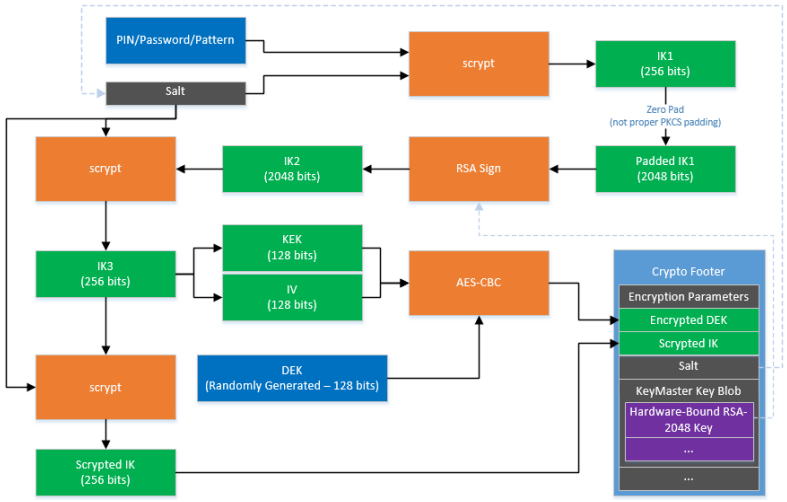


Figura 2.14: Nuevo proceso de generación de la clave DEK. Adaptado de [41].

Para vincular la DEK al dispositivo, se añade un campo adicional en la estructura crypto footer: un key blob generado por `KeyMaster`. Este key blob contiene una clave privada RSA-2048 cifrada por `KeyMaster`, que se utiliza para firmar la clave IK1 en el paso 4. Además, la estructura crypto footer incluye un campo adicional que no tiene un propósito directo en el proceso de descifrado; es el valor devuelto al ejecutar scrypt en la clave IK3 (Scrypted IK en el diagrama anterior). Este valor permite a Android determinar cuándo una clave de cifrado es válida, incluso cuando el propio disco se encuentra defectuoso.

2.5.2. File-Based Encryption

Android File-Based Encryption (FBE) es una característica de seguridad incorporada en la versión 7.0 de Android [17], siendo obligatoria para los dispositivos que se inician con Android 10 o versiones superiores. Contrariamente al cifrado FDE, que encripta todo el almacenamiento a nivel de bloque, FBE se enfoca en cifrar archivos y directorios específicos de manera individual

FBE utiliza el subsistema **fscrypt** para implementar el cifrado de archivos. Al operar a nivel de sistema de archivos, es posible cifrar diferentes archivos con claves diferentes y tener al mismo tiempo archivos sin encriptar [18]. Sin embargo, a excepción de los nombres de fichero, fscrypt no cifra los metadatos del sistema de archivos.

Al permitir encriptar cada archivo con una clave diferente, en Android se establecen dos áreas de almacenamiento cifradas con políticas de cifrado separadas:

- **Device Encrypted (DE) storage**: Espacio de almacenamiento disponible inmediatamente después de iniciar el dispositivo, antes de la autenticación del usuario. Cuando solo se tiene acceso a esta área, el dispositivo opera en modo **direct boot**. En este modo, el dispositivo arranca directamente hasta la pantalla de bloqueo, permitiendo el funcionamiento de: aplicaciones que tienen notificaciones programadas, como las alarmas; aplicaciones que proporcionan notificaciones importantes al usuario, como las de mensajes SMS; o aplicaciones que brindan servicios de accesibilidad como TalkBack. Anteriormente, en dispositivos cifrados con FDE, los usuarios debían ingresar la contraseña del dispositivo antes de acceder a cualquier dato.

[17] `https://source.android.com/docs/security/features/encryption/file-based`
[18] `https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html`

- **Credential Encrypted (CE) storage**: Espacio de almacenamiento predeterminado. Solo está disponible después de que el usuario haya desbloqueado el dispositivo.

La división mencionada aumenta la seguridad de los perfiles personales y de trabajo, ya que posibilita la protección de múltiples usuarios simultáneamente. Esto es debido a que ahora no se depende exclusivamente de una contraseña de inicio, sino que ahora cada perfil es protegido mediante la contraseña de desbloqueo establecida por cada usuario.

En FBE, el proceso de encriptación se divide en dos partes: primero se encripta el nombre de un archivo y luego su contenido. La razón detrás de esta separación radica en la forma en que se almacena la información de un archivo y su nombre en un sistema Linux. En Linux, se utilizan dos estructuras distintas para gestionar esta información [19]:

- **inode**: Estructura de datos en sistemas de archivos Unix y derivados que almacena información sobre un archivo o un directorio, excluyendo su nombre y su contenido real. Cada archivo o directorio en un sistema de archivos tiene un inode asociado, y este inode contiene metadatos importantes sobre el archivo, como por ejemplo el número de inode, el usuario propietario, los atributos extendidos, el tipo de archivo, etc.
  - **Atributos extendidos (xattr)**: Metadatos adicionales asociados a un archivo o directorio en sistemas de archivos Unix y derivados. En el contexto de un inode, los xattr proporcionan información adicional sobre el archivo o directorio, como atributos extendidos específicos del sistema de archivos, permisos especiales o cualquier otra información relevante que no esté relacionada directamente con el nombre o contenido del archivo. En FBE, estos atributos son críticos para el manejo de información importante para el cifrado. En concreto, se almacena una política de encriptación que incluye información básica como el tipo de cifrado utilizado para encriptar los nombres de los archivos o el tipo de cifrado utilizado para encriptar el contenido de los archivos. Además, se incluye un *nonce* único y aleatorio de 128 bits por archivo, utilizado para derivar la clave de encriptación de ese archivo, también conocida como *per-file-key* [42].
- **dentry**: Estructura de datos utilizada para representar y administrar las entradas de directorio en el sistema de archivos. El dentry asocia un nombre de archivo con su correspondiente inode. Cuando se accede a un archivo en un directorio, el SO utiliza la información almacenada en el dentry para localizar y acceder rápidamente al inode correspondiente.

En el ámbito de las claves empleadas para el cifrado en FBE, se identifican dos claves principales: la clave del área DE y la clave del área CE. La clave DE se genera automáticamente al iniciar el dispositivo. Por otro lado, la clave CE (DEK) está resguardada por la clave KEK, la cual se deriva a partir del PIN, patrón o contraseña de desbloqueo establecido por el usuario, y se almacena de forma segura en el TEE [43]. En contraste con FDE, la clave DEK en FBE tiene un tamaño de 512 bits.

La DEK puede conceptualizarse como una 'superclave'. Al aplicar el algoritmo AES-128-ECB junto con el nonce único asignado a cada archivo, se genera la per-file-key para ese archivo [20]. Posteriormente, se aplica el algoritmo AES-256-XTS (por defecto) o Adiantum [21] para cifrar el contenido del archivo. En cuanto al cifrado del nombre del archivo,

[19] `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`

[20] Debido a que los nombres de archivo forman parte de los datos del contenido de la carpeta principal, todos los nombres de archivos en la misma carpeta se cifran con la misma clave. En consecuencia, a nivel de carpeta, la encriptación de los nombres de archivo está vinculada y comparte la misma clave.

[21] `https://source.android.com/docs/security/features/encryption/adiantum`

se utilizan 256 bits de la per-file-key junto con el algoritmo AES-256-CTS (por defecto), AES-256-HEH o AES-256-HCTR2 [22], junto con un IV de 0. También es posible emplear el algoritmo Adiantum. El nombre a cifrar se codifica utilizando el conjunto [a-zA-Z0-9_+] antes de aplicarle el cifrado, y el resultado del cifrado se almacena codificado en **base62**, asegurando que el nombre cifrado siga siendo válido en el sistema de archivos.

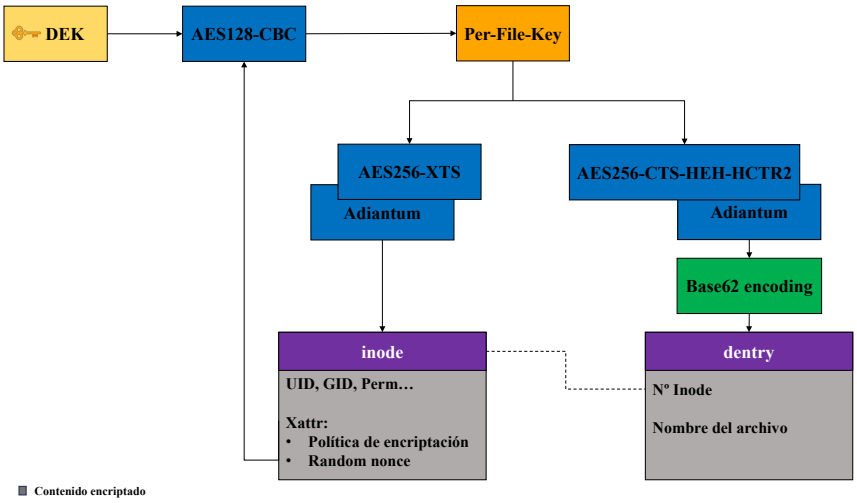


Figura 2.15: Encriptación individual de cada archivo en FBE.

Similar a FDE, la DEK queda protegida por la KEK mediante un complejo esquema de derivación de claves que puede observarse en la siguiente figura:

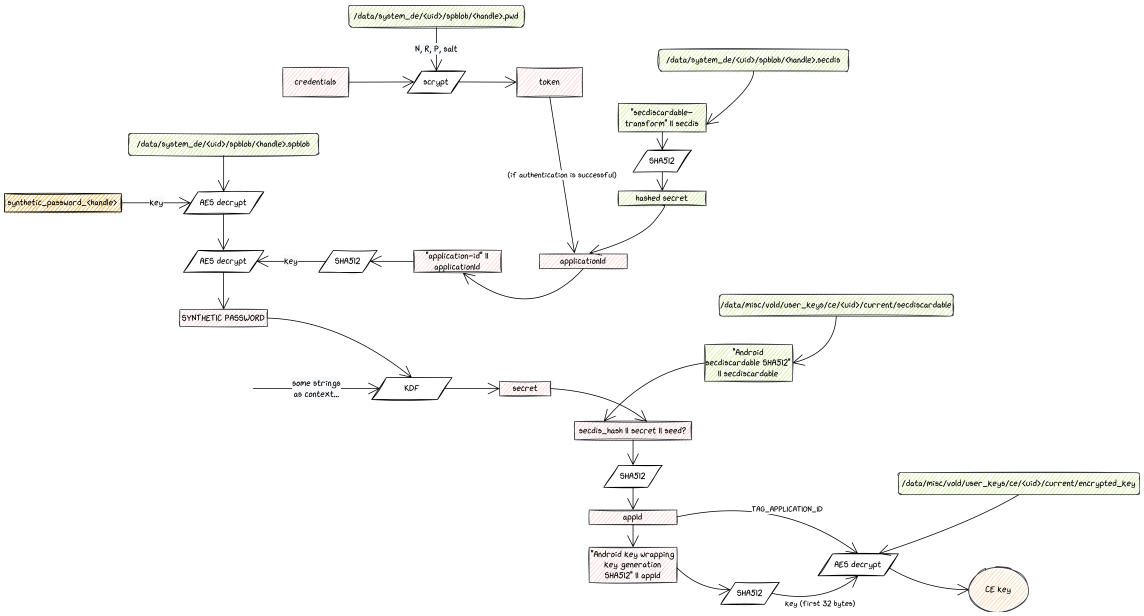


Figura 2.16: Derivación de la clave KEK en FBE. Adaptado de [44].

[22] Desde Android 14, AES-256-HCTR2 es el modo preferido.

En este esquema se suceden una serie de pasos hasta producir la KEK, que se introducirá junto con la DEK encriptada en el algoritmo AES-256-GCM. El proceso de derivación de la KEK sigue los pasos que se detallan a continuación [44]:

1. Se obtiene un `auth token` generado por `Gatekeeper`. Este token se genera utilizando las credenciales del usuario derivadas del PIN, patrón o contraseña de desbloqueo, complementadas con un salt y sometidas a un proceso de hash para ser convertidas en una entrada válida para scrypt.
   - El resultado de aplicar scrypt a las credenciales es un token, cuyo HMAC se compara con el HMAC del password handle generado la primera vez que el usuario establece las credenciales. Si coinciden, se genera el `auth token`. Adicionalmente, el sistema cuenta con un `applicationId` válido, obtenido a partir del hash SHA-512("$secdiscardable - transform$" || `secdis file`), que se utilizará en la siguiente fase.
2. Se deriva la *Synthetic Password*. Esta contraseña actúa como una clave intermedia permitiendo el cambio de PIN, patrón o contraseña sin afectar la clave CE [45].
   2.1 Se desbloquea el synthetic password handle utilizando el `auth token` obtenido previamente.
   2.2 A continuación, se descifra el contenido del synthetic password handle mediante el algoritmo AES, con una clave protegida por `Keystore`. Esta operación de descifrado se lleva a cabo en el TEE por el trustlet `KeyMaster`.
   2.3 Finalmente, se descifra el resultado anterior con el algoritmo AES-GCM, utilizando como clave el hash SHA-512("$application - id$" || `applicationId`).
3. Se introduce la synthetic password en una KDF para producir un secreto (`secret`).
4. Se crea un `appId` de la siguiente manera $appId =$ `secdiscardable hash` || `secret` [23]:
   4.1 Se añade el `secdiscardable hash` al inicio. Este hash se obtiene al aplicar el algoritmo SHA-512 a 16 KiB de datos aleatorios del archivo `secdiscardable`, almacenado individualmente para cada usuario. El hash se emplea como protección *anti rollback*.
   4.2 Se concatena el secreto `secret` al `secdiscardable hash`.
5. Se realiza el hash
   SHA-512("*Android key wrapping key generation SHA*512" || `appId`) [24].
6. Finalmente, se toman los primeros 256 bits (32 bytes) del resultado anterior como clave KEK.

En dispositivos que incluyen un chip de seguridad como pueden ser los de la familia Pixel de Google, el proceso anterior sufre ligeras modificaciones que se pueden observar en la siguiente figura.

[23] `https://cs.android.com/android/platform/superproject/main/+/main:system/vold/KeyStorage.cpp;l=606`

[24] `https://cs.android.com/android/platform/superproject/main/+/main:system/vold/KeyStorage.cpp;l=484`

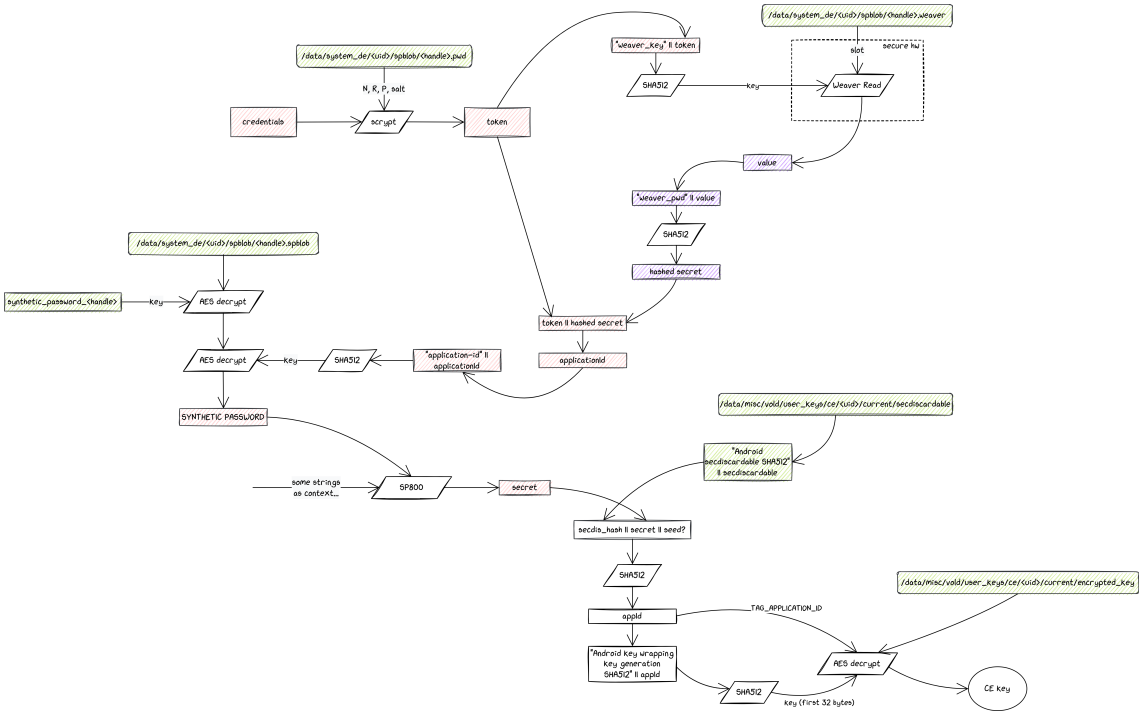


Figura 2.17: Derivación de la clave KEK en dispositivos con chip de seguridad. Adaptado de [44].

En este tipo de dispositivos, los pasos referentes a la generación del `applicationId` (etapa 1) sufren pequeñas modificaciones para hacer uso del chip:

1. Se obtiene un `auth token` generado por `Gatekeeper`. Este token se utiliza para generar una clave que será utilizada por `Weaver`. La clave se obtiene de aplicar el algoritmo SHA-512("*weaver_key*" || `auth token`).
    - Cuando el chip recibe la clave, utiliza también un *slot number* almacenado en el archivo `/data/system_de/<uid>/spblob/<handle>.weaver`. Si la clave y este slot number coinciden, el chip de seguridad envia de vuelta el `value`. Este valor se utiliza para producir el `hashed secret`, tras aplicar el algoritmo SHA-512("*weaver_pwd*" || `value`).
    - En última instancia, se concatenan el `auth token` y el `hashed secret` para producir el `applicationId`.

2.5.3. Metadata Encryption

La última característica de encriptación de datos ofrecida por Android es conocida como metadata encryption [25], que funciona en conjunto con FBE. Se introdujo a partir de la versión 9 de Android, motivada por el hecho de que FBE no cifra metadatos como la disposición de los directorios, el tamaño de los archivos, los permisos de los ficheros, etc.

Con el cifrado de metadatos, una única clave (independiente de las utilizadas en FBE) presente en el momento del arranque, cifra cualquier contenido que no esté cifrado por FBE. Esta clave está protegida por `KeyMaster`, que a su vez está protegida por un arranque verificado. El cifrado de metadatos siempre está habilitado en el almacenamiento adoptable cuando se activa FBE. El cifrado de metadatos también se puede habilitar en el almacenamiento interno, siendo obligatoria su activación para aquellos dispositivos lanzados con Android 11 o superior.

[25] `https://source.android.com/docs/security/features/encryption/metadata`

CAPÍTULO 3
Estudio de TEE en Dispositivos Móviles

Este capítulo presenta un análisis exhaustivo de las implementaciones de *Trusted Execution Environment* (TEE) realizadas por los principales fabricantes de dispositivos móviles. Se examinan las soluciones ofrecidas por MediaTek, Qualcomm, Apple, Samsung y Google, proporcionando una visión detallada de las características y enfoques adoptados por cada uno de ellos en el desarrollo de entornos seguros para dispositivos móviles.

3.1 Mediatek

El principal fabricante de *System on a Chip* (SoC) por cuota de mercado [1] no ofrece una solución propietaria como el siguiente en esta lista, en su lugar, según al mercado al que se dirija el SoC incorporará una solución diferente. Desde el 2013, en el mercado europeo se utiliza el TEE Knibi [2]. Mientras tanto, en el mercado chino, los SoCs de Mediatek implementan el TEE Beanpod o el TEE Mitee [46]. En esta sección, solo se va a tener en cuenta Knibi, que se expone a continuación.

El Trusted OS Knibi, también conocido como T-Base o Mobicore, tiene su origen en la empresa Trustonic. Entre los TEE que se van a comentar, Knibi es uno de los SO que sigue los estándares definidos por el consorcio *GlobalPlatform* (GP). Knibi está formado por los componentes presentes en la figura 3.1, donde destacan los siguientes [47] [48]:

- **MTK**: El microkernel de Knibi proporciona llamadas al sistema a *drivers* y trustlets, gestiona la comunicación entre procesos y proporciona aislamiento entre tareas, entre otras funciones. Un fragmento de código en el monitor seguro retransmite las interrupciones `SMC` del *Normal World* (NWd) a MTK, permitiendo así la comunicación entre mundos.

- **RTM**: El *Run-Time Manager* situado en el *Secure World* (SWd), es el proceso principal de Knibi y sería equivalente al proceso `init` de Linux. Su función principal es la gestión de sesiones entre aplicaciones del NWd y trustlets. Cuando una aplicación en el NWd abre una nueva sesión, RTM primero verifica si el trustlet ya está cargado. Además, este componente se encarga de descifrar las TAs antes de cargarlas, ya que en esta solución una TA puede estar encriptada.

[1] `https://www.counterpointresearch.com/insights/global-smartphone-ap-market-share/`

[2] `https://www.trustonic.com/news/mediatek-licences-trustonic-trusted-execution-environment/`

- **McLib**: La *Mobicore Library* es utilizada por trustlets y drivers y consta de una API interna propietaria en conjunto con la GlobalPlatform Internal API [49]. Esta librería se mapea en cada driver o trustlet en una dirección fija. Las funciones expuestas para los trustlets se prefijan con `tlApi`, mientras que las funciones expuestas solo para drivers se prefijan con `drApi`.

- **Trusted Drivers**: Estos controladores se ejecutan en el mismo nivel de privilegios que las TAs, sin embargo, tienen acceso a un conjunto más rico de funcionalidades dentro de McLib. Estas funcionalidades adicionales permiten a los controladores acceder a llamadas `SVC` adicionales, asignar memoria física, utilizar subprocesos y realizar llamadas `SMC`, así como permitir a las TAs acceder a los periféricos del dispositivo. Además de los controladores generales, Kinibi incluye un controlador criptográfico (`drcrypto`) y el controlador del servicio de almacenamiento seguro (`STH2`), ambos incorporados en el propio SO y ejecutándose como tareas independientes. `drcrypto` proporciona servicios criptográficos, y `STH2` proporciona servicios de almacenamiento seguro a TAs y controladores de confianza.

- **Secure Objects**: Los objetos seguros no son un componente de Knibi, más bien son una serie de 'entidades' ofrecidas por Knibi a las aplicaciones del NWd. Estos objetos contienen datos cifrados y protegidos frente la pérdida de integridad. Una vez cifrados, los datos pueden almacenarse en localizaciones inseguras como el almacenamiento secundario del dispositivo. La clave para descifrar el objeto seguro se deriva de la clave maestra del dispositivo y es única para cada TA.

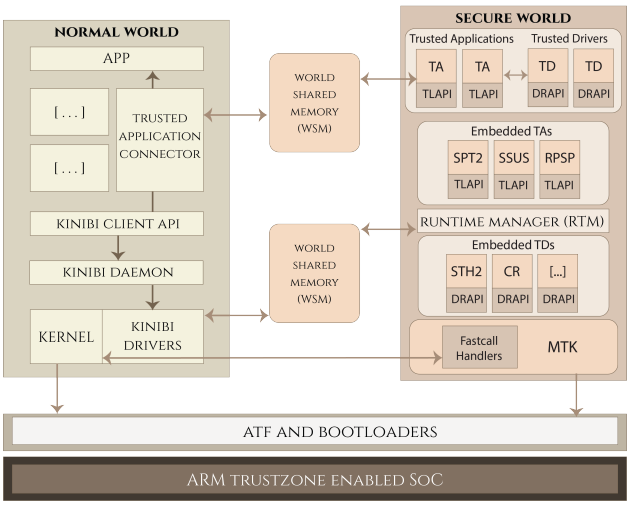


Figura 3.1: Principales componentes Knibi. Adaptado de [48].

Analizando el proceso de comunicación entre aplicaciones y TAs, una aplicación en el NWd hace uso de la API Knibi Mobicore Client API para interactuar con las *Trusted Applications* (TAs). Las aplicaciones acceden a esta API a través de la librería libMcClient (`libMcClient.so`), la cual ofrece APIs para solicitar *handles* a sesiones y retransmitir notificaciones entre el NWd y el SWd mediante buffers de memoria compartida. La API del cliente MobiCore funciona retransmitiendo las peticiones de las aplicaciones al `MobiCore Daemon` que se ejecuta al mismo nivel que Android. Este daemon retransmite la petición al driver `MobiCore`, que se comunica con RTM a través de la interfaz *Mobicore Communication Interface* (MCI).

Cuando una aplicación situada en el NWd desea comunicarse con un trustlet, primero solicitará al `Trusted Application Connector` la creación de una sesión, haciendo uso de la llamada `mcOpenDevice()`. Esta llamada realiza una solicitud al `Mobicore Daemon`, que proporcionará un descriptor de fichero para interactuar con el dispositivo virtual `/dev/mobicore-user`. Este dispositivo virtual se utiliza para comunicarse con el `Mobicore Driver` situado en el NWd. Una vez se tiene acceso al dispositivo virtual, se puede solicitar al `Trusted Application Connector` la creación de una región de memoria compartida entre mundos (WSM), haciendo uso de la función `mcMallocWSM()`.

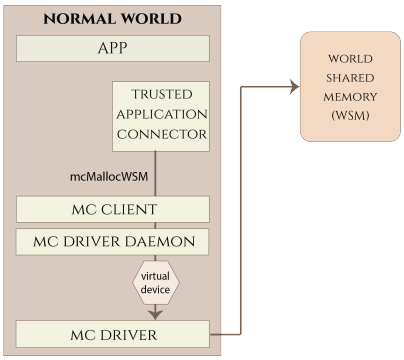


Figura 3.2: Creación de una región de memoria compartida. Adaptado de [48].

En este punto, el `Trusted Application Connector` intentará establecer una sesión con el trustlet correspondiente haciendo uso de la llamada `mcOpenTrustlet()`. Esta llamada utiliza como parámetros el ID de sesión del dispositivo, un puntero a la dirección de memoria que contiene la TA a cargar y un puntero y el tamaño del buffer TCI contenido en la región de memoria WSM. La solicitud se transmite al `Mobicore Driver`, para que retransmita la petición al SWd. Esta petición tendrá como resultado el ID de la sesión creada.

El protocolo de comunicación que seguirá la aplicación a la hora de intercambiar mensajes con la TA, se define como específico de la aplicación. Un uso extendido consiste en emplear los cuatro primeros bytes del buffer para enviar el `command_id`, dejando el resto del espacio para almacenar los parámetros. Una vez se proporcionan estos datos, se debe notificar al trustlet que el comando está listo para ser procesado, utilizando la función `mcNotify()`. La aplicación esperará la respuesta a su solicitud utilizando la función `mcWaitNotification()`. Una TA que haya sido cargada, puede hacer uso de la función `tlApiWaitNotification()` para esperar a recibir comandos.

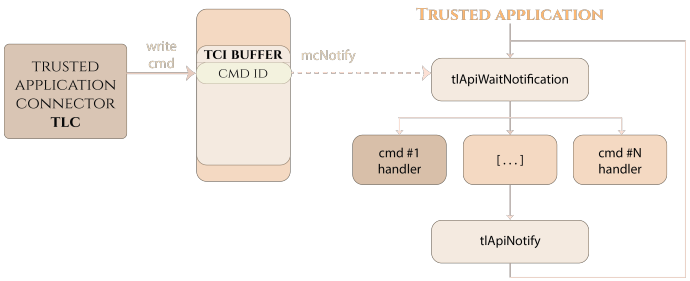


Figura 3.3: Protocolo de recepción de comandos. Adaptado de [48].

Una vez se ha notificado a la TA correspondiente, esta lee el buffer de memoria compartida para extraer el `command_id`, así como los parámetros. Una vez terminada la operación, el trustlet escribe la respuesta en el buffer de memoria compartida y ejecuta la función `tlApiNotify()`. Esta llamada será retransmitida al NWd, haciendo que la lla-

mada `mcWaitNotification()` termine. En este momento, la aplicación podrá revisar el buffer para obtener la respuesta.

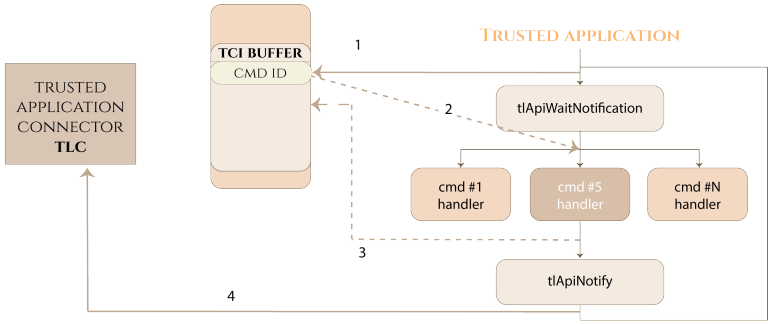


Figura 3.4: Protocolo de respuesta a un comando. Adaptado de [48].

Por último, comentar el formato que presentan los trustlets. En Knibi los trustlets utilizan el formato *Mobicore Loadable Format* (`MCLF`) [50]. Cada archivo se compone de una cabecera que contiene información como el tipo de fichero, el UUID o la dirección y tamaño de sus segmentos. A nivel de segmentos, todas las TAs se componen de tres segmentos: los segmentos de código (`text`), datos inicializados (`data`) y datos no inicializados (`bss`). Tras la cabecera y los segmentos de código y datos, el binario contiene una clave pública y un *blob* de firma. Cuando se carga un trustlet en el TEE, el hash de la clave pública se compara con un hash contenido en el binario de Knibi y, a continuación, se verifica la firma del trustlet.

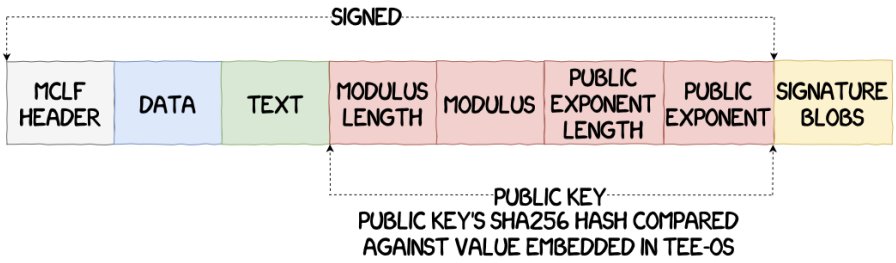


Figura 3.5: Formato de un trustlet MCLF. Adaptado de [50].

3.2 Qualcomm

La solución desarrollada por Qualcomm para garantizar la seguridad de sus chips se conoce como Qualcomm Trusted Execution Environment (QTEE), antes conocida como Qualcomm Secure Execution Environment. Esta solución se encuentra presente en una amplia variedad de dispositivos debido a la sólida posición de mercado que ostenta la marca. QTEE integra los estándares establecidos por el consorcio GlobalPlatform, junto con una serie de APIs propietarias. A continuación, se presenta una figura con los componentes principales de esta solución.

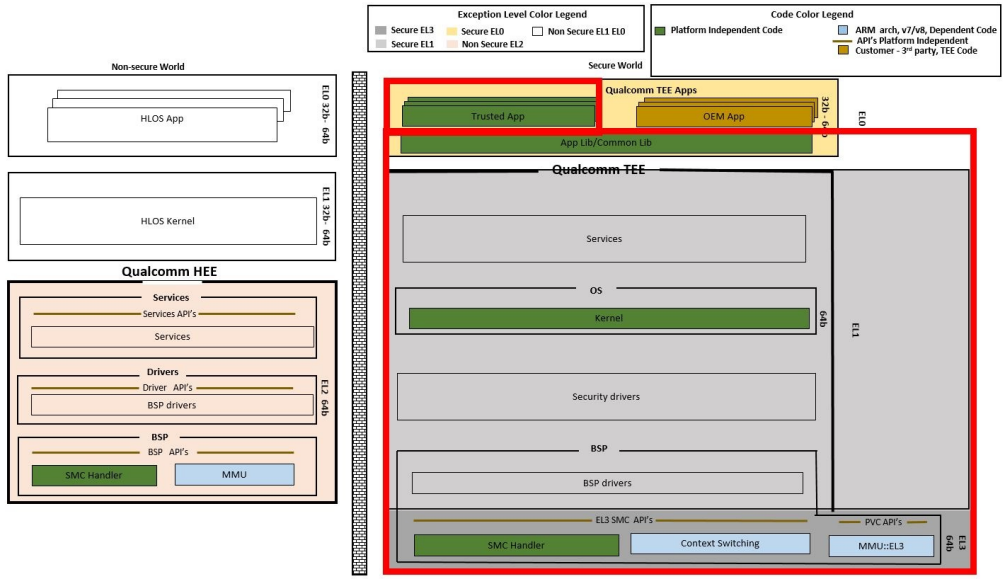


Figura 3.6: Componentes de Qualcomm Trusted Execution Environment. Adaptado de [51].

En contraste con Knibi, esta solución presenta una estructura más simplificada. QTEE está compuesto por las TAs, una librería que recibe el nombre de *App Lib*, un SO y un driver en Android (`qseecom`) [51] [34]. Las TAs implementan la API TEE Internal Core API y hacen uso de la librería mencionada para utilizar los servicios del SO, mientras que las aplicaciones de Android utilizan la API propietaria QSEEComAPI para poder comunicarse con el driver `qseecom`. Debido al riesgo de seguridad que supondría permitir a cualquier aplicación de usuario interactuar con el driver, solo cuatro procesos pueden comunicarse con este [52]:

- **surfaceflinger**: Se ejecuta con el ID de usuario 'system'.
- **drmserver**: Se ejecuta con el ID de usuario 'drm'.
- **mediaserver**: Se ejecuta con el ID de usuario 'media'.
- **keystore**: Se ejecuta con el ID de usuario 'keystore'.

El mecanismo utilizado para solicitar operaciones a un trustlet, sigue un esquema similar al implementado por Knibi. En esta solución, uno de los cuatro procesos antes mencionados utilizará la llamada al sistema `ioctl()` para comunicarse con el driver `qseecom`. Este driver a continuación, transmitirá el comando solicitado al kernel QTEE que en última instancia lo enviará al trustlet apropiado. La respuesta seguirá este mismo proceso, pero en orden inverso [53].

Profundizando en este proceso de comunicación, el método para solicitar operaciones consta de los siguientes pasos:

1. El proceso solicita un buffer ION del *heap* `qsecom` [3]. Utilizar un heap de este tipo garantiza que la memoria solicitada sea contigua en el espacio de direcciones físicas.
2. A continuación, el proceso podrá emplear la llamada al sistema `mmap()` para escribir directamente los datos en el buffer anterior.

[3] `https://android.googlesource.com/kernel/msm/+/refs/tags/android-13.0.0_r0.80/drivers/staging/android/uapi/msm_ion.h#41`

3. El proceso envía una solicitud `ioctl` del tipo `QSEECOM_IOCTL_SEND_CMD_REQ` al driver `qseecom`. Este `ioctl` contiene la ubicación de los datos en el buffer ION y la ubicación donde se deberán escribir la respuesta. En el siguiente listing, se puede observar el aspecto de esta solicitud.

```
/*
* struct qseecom_send_cmd_req - for send command ioctl request
* @cmd_req_len - command buffer length
* @cmd_req_buf - command buffer
* @resp_len - response buffer length
* @resp_buf - response buffer
*/
struct qseecom_send_cmd_req {
    void *cmd_req_buf; // Buffer de entrada
    unsigned int cmd_req_len;
    void *resp_buf; // Buffer de respuesta
    unsigned int resp_len;
}
```

Listing 3.1: Formato de una solicitud `ioctl` para procesar comandos.

4. El driver transfiere la dirección física [4] de los buffers de solicitud y respuesta al kernel QTEE.

5. El kernel proporciona la dirección física de los buffers para que el trustlet adecuado atienda la petición.

6. El trustlet realiza la operación solicitada y escribe la respuesta en el buffer de respuesta.

7. Finalmente, se notifica al programa de usuario que la solicitud ha sido completada, pudiendo acceder al buffer de respuesta.

El protocolo de comunicación descrito se vuelve insuficiente cuando los trustlets necesitan acceder a la memoria de otros heap ION. Para abordar este escenario, se modifica la solicitud, incluyendo un puntero a dicho buffer. Sin embargo, esta nueva situación plantea dos problemas importantes que deben ser solucionados:

- Una aplicación de usuario no puede escribir la dirección física de un buffer ION ajeno. Esto se debe a que la aplicación no conoce dicha dirección; además, permitir a una aplicación de usuario controlar la dirección física a utilizar presenta un riesgo de seguridad.
- El kernel QTEE debería de dar acceso al buffer ION al trustlet.

Teniendo en cuenta los dos problemas planteados, el proceso de comunicación variará ligeramente:

1. En lugar de enviar un `ioctl` del tipo `QSEECOM_IOCTL_SEND_CMD_REQ`, esta vez se utiliza el tipo `QSEECOM_IOCTL_SEND_MODFD_CMD_64_REQ`. La distinción entre estos dos tipos de `ioctl` radica en que el segundo incluye hasta cuatro descriptores de fichero ION, cada uno con un *offset* en el buffer de entrada.

[4] QTEE opera en direcciones virtuales, pero solo están reflejando la memoria física subyacente, por lo que las direcciones virtuales son las mismas que las físicas. Así es como el kernel de QTEE limita los trustlets para que solo tengan acceso a cierta memoria.

```
#define MAX_ION_FD 4

/*
* struct qseecom_ion_fd_info - ion fd handle data information
* @fd - ion handle to some memory allocated in user space
* @cmd_buf_offset - command buffer offset
*/
struct qseecom_ion_fd_info {
    int32_t fd;
    uint32_t cmd_buf_offset;
};
/*
* struct qseecom_send_modfd_cmd_req - for send command ioctl request
* @cmd_req_len - command buffer length
* @cmd_req_buf - command buffer
* @resp_len - response buffer length
* @resp_buf - response buffer
* @ifd_data_fd - ion handle to memory allocated in user space
* @cmd_buf_offset - command buffer offset
*/
struct qseecom_send_modfd_cmd_req {
    void *cmd_req_buf; // Buffer de entrada
    unsigned int cmd_req_len;
    void *resp_buf; // Buffer de respuesta
    unsigned int resp_len;
    struct qseecom_ion_fd_info ifd_data[MAX_ION_FD];
};
```

Listing 3.2: Formato de la solicitud para procesar memoria de otros heap ION.

2. Al recibir la solicitud, `qseecom` toma las direcciones físicas de cada búfer ION del *array* `ifd_data` y las escribe en el búfer de solicitud, en la posición indicada por el campo `cmd_buf_offset` [5]. En este punto se ha conseguido solucionar el primero de los problemas. Para permitir que el kernel QTEE conozca la ubicación de los otros buffers ION a los que el trustlet necesita acceder, se convierte el array `ifd_data` en un array del tipo `sglist_info`.

```
struct sglist_info {
    uint32_t indexAndFlags; // Offset
    uint32_t sizeOrCount; // Tamaño
};
```

Listing 3.3: Formato de la estructura `sglist_info`.

   Cada entrada de este nuevo tipo de datos conserva el mismo offset que estaba presente en la estructura `qseecom_ion_fd_info`, pero ahora, en lugar de contener el descriptor de fichero (`fd`), contiene el tamaño del buffer ION.

3. En este punto, el kernel QTEE recibe las entradas `sglist_info` junto con los buffers de solicitud y respuesta. Al procesar estas entradas, el kernel QTEE extrae la dirección física del buffer ION del buffer de solicitud en el offset especificado. Luego, esta dirección se utiliza para permitir el acceso al trustlet correspondiente.

En este último punto se analizará el aspecto de un trustlet. En QTEE los trustlets se organizan como ficheros regulares ELF firmados, cuyo aspecto se comenta en la sección 2.3. Como parte de esta organización, se incluye una sección específica conocida como *Hash Table Segment* [54]. Esta sección adicional incorpora:

[5] `https://android.googlesource.com/kernel/msm/+/refs/tags/android-13.0.0_r0.80/drivers/misc/QTEEcom.c#4074`

- La cabecera de la tabla hash.
- Una tabla hash encabezada por SHA-256(`Elf_Ehdr` || `Elf_Phdr`). Después, tras una entrada vacía, se incluye el hash SHA-256 de cada segmento del programa.
- Firma de bloque. Esta firma se calcula a partir del hash, la cabecera del segmento, los metadatos de la imagen y la tabla hash.
- Una cadena de certificados que servirá para verificar la TA.

En el sistema, los ficheros ELF se encuentran divididos en varias partes, las cuales se cargan conjuntamente cuando se requiere utilizar un trustlet [55]. Estas partes incluyen un fichero de cabecera con extensión `mdt`, que contiene la cabecera ELF, la *program header table* y el segmento hash. También, existen múltiples archivos con extensión `bxx` ($x \in \{0, 9\}$), que albergan el contenido de cada segmento restante del fichero ELF.

3.3 Apple

La tercera solución contemplada en este documento es la desarrollada por Apple, conocida como Apple Secure Enclave (ASE). Lanzado en 2013, fue incorporado en el SoC Apple A7 [6] que venía incluido en productos como el iPhone 5s o el primer iPad Air [7]. Actualmente, ASE se encuentra en gran variedad de productos de la marca, desde el *smartwatch* Apple Watch hasta el altavoz Homepod.

Una diferencia notable respecto a las soluciones comentadas anteriormente, es que ASE **se implementa** sobre un coprocesador ARMv7-A 'Kingfisher' que coexiste con el procesador principal dentro del SoC. El acceso a este coprocesador no está permitido desde el procesador principal, incluso cuando este último se encuentra en el nivel de excepción EL3. Con las versiones A11 y S4 de los SoCs, se incluye un motor de encriptado de memoria, arranque seguro, un generador de números aleatorios dedicado y un motor criptográfico AES dedicado [56]. Por último, comentar que en el momento de la redacción del documento no existe información disponible que certifique que la solución sigue los estándares definidos por el consorcio GlobalPlatform.

Debido a la limitada información pública disponible sobre el procesador, resulta difícil determinar aspectos tales como los servicios expuestos, los privilegios necesarios para su uso o los métodos de acceso a estos servicios [57]. Sin embargo, se sabe que el *Secure Enclave Processor* (SEP) dispone de una serie de periféricos accesibles mediante memory-mapped IO que no están disponibles para el procesador principal [58]. Además de estos periféricos, el SEP comparte algunos periféricos con el procesador principal, como el controlador de memoria. El sistema operativo que se ejecuta en este procesador utiliza un núcleo derivado del micronúcleo L4, basado en Darbat/L4Ka::Pistachio. Apple ha realizado modificaciones en el núcleo para implementar sus propios controladores, servicios y aplicaciones, los cuales se compilarán como archivos Mach-O regulares. Es importante destacar que la mayor parte del funcionamiento se lleva a cabo en modo usuario. La arquitectura de este sistema operativo se puede apreciar en la siguiente figura.

[6] `https://en.wikipedia.org/wiki/Apple_A7`
[7] `https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web`

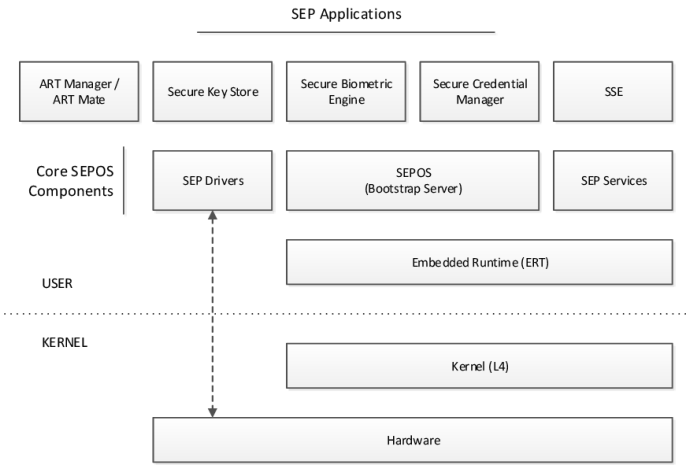


Figura 3.7: Componentes de Secure Enclave Processor OS. Adaptado de [57].

De la figura anterior comentar el núcleo L4. El kernel únicamente se encarga de inicializar el sistema y gestionarlo. Para este último fin se provee un conjunto de unas veinte llamadas al sistema que se dividen en dos grupos: privilegiadas y no privilegiadas. El grupo de las privilegiadas está formado por las relacionadas con el manejo de memoria e hilos de ejecución. Estas llamadas al sistema especiales, únicamente pueden ser utilizadas por el proceso raíz.

El proceso raíz se encarga de inicializar el resto de aplicaciones, mantener una estructura de contexto de cada aplicación e invocar al *bootstrap server*. El bootstrap server implementa la mayoría de funcionalidades del SO, exportando los métodos para el manejo del sistema, memoria e hilos. Las entidades exportadas son accesibles para las aplicaciones mediante *Remote Procedure Call* (RPC). Adicionalmente, este servidor permite a las aplicaciones realizar operaciones privilegiadas como la creación de un hilo.

A la hora de establecer una comunicación entre el procesador principal y SEP, se utiliza un hardware compartido llamado *Secure Mailbox*, que ofrece una bandeja de entrada y otra de salida. Este hardware utiliza los registros de E/S del coprocesador: cuando se desea enviar un mensaje, se escribe en el registro *inbox* del mailbox y cuando se devuelve una respuesta, la escritura se realiza en el registro *outbox*. El formato que presentan los mensajes tiene un tamaño de 8 bytes y se corresponde de los siguientes campos:

```
struct {
    uint8_t endpoint; // destination endpoint number
    uint8_t tag;      // message tag
    uint8_t opcode;   // message type
    uint8_t param;    // optional parameter
    uint32_t data;    // message data
} sep_msg;
```

Listing 3.4: Estructura de un mensaje enviado a través del Secure Mailbox.

En SEPOS existe un *Endpoint Manager* que permite a los drivers registrar un endpoint. Un endpoint ofrece la funcionalidad de envío y recepción de mensajes. Cuando el procesador principal desea enviar un mensaje, escribirá el mensaje en el registro inbox. Este mensaje será encolado por el Endpoint Manager en el endpoint correspondiente. Una vez encolado, una interrupción avisará al coprocesador de su recepción. Cuando se genere la respuesta, el coprocesador escribirá los datos en el registro outbox, momento en el cual se notificará vía interrupción de su llegada al procesador principal, que podrá leer los datos.

Cuando se desea transmitir mensajes más grandes, el mecanismo anterior resulta ineficiente. Para este fin existe un mecanismo alternativo el cual emplea *Out-of-line buffers*. Estos buffers son un espacio de memoria compartida gestionado desde el SEPOS mediante el endpoint de control `EP0`. Este tipo especial de endpoint se encarga de configurar los buffers de solicitud y respuesta y de asignar estos buffers al endpoint adecuado.

3.4 Samsung

La solución ofrecida por el fabricante surcoreano es conocida como Samsung TEEGRIS. Desde el año 2019, con el lanzamiento del dispositivo Samsung Galaxy S10, todos los productos de la marca incorporan esta solución, habiendo utilizado previamente el TEE Knibi [59]. TEEGRIS ha obtenido una evaluación de seguridad [8] de acuerdo con los requerimientos de seguridad especificados por GlobalPlatform.

Al igual que en soluciones anteriores, TEEGRIS presenta una estructura sencilla que está compuesta por un kernel seguro, los trustlets y un controlador en el NWd. Analizando primero el kernel, su tamaño es reducido e integra una serie de controladores que pueden ser utilizados por los trustlets. El kernel se ejecuta en modo de 64 bits y admite trustlets y controladores tanto de 32 como de 64 bits, que se ejecutan en el espacio de usuario. Este kernel implementa una serie de llamadas al sistema compatibles con el estándar POSIX junto con una serie de llamadas al sistema específicas [60].

Los drivers en TEEGRIS se han implementado siguiendo las especificaciones POSIX. La forma de interactuar con los controladores utilizando esta interfaz es similar a la que se utiliza en sistemas operativos como GNU/Linux: primero se abre el archivo especial `/dev` mediante la llamada al sistema `open()`, para luego solicitar operaciones mediante `ioctl()`. Los controladores tienen un nombre que generalmente comienza con "dev://", y pueden ser accedidos abriendo el archivo correspondiente desde una TA. Estos archivos se operan con un conjunto de llamadas al sistema disponibles para las TAs. Dentro del kernel, se utiliza una estructura para almacenar la implementación de las llamadas al sistema disponibles para cada controlador.

Algunos trustlets en TEEGRIS se incluyen como objetos inmutables que el kernel carga durante el arranque y se almacenan en el archivo `startup.tzar`, en la partición de arranque del dispositivo. Existen otro tipo de trustlets que son cargados dinámicamente por Android. Todos los trustlets, independientemente de su tipo, se organizan en grupos, a los cuales se les pueden asignar permisos estáticos según la funcionalidad del trustlet. Esta organización por grupos permite conceder acceso específico a una serie de controladores o llamadas al sistema, garantizando que solo las TAs permitidas puedan acceder a la funcionalidad restringida mediante un registro de permisos mantenido por el kernel.

El formato de los trustlets puede variar dependiendo de su ubicación. Si una TA se encuentran en la partición de arranque, se organizarán como un fichero regular ELF. Por otro lado, si se encuentran fuera de la partición de arranque, la estructura será la siguiente:

- **Cabecera**: Tiene un tamaño de 8 bytes, que se reparten como 4 bytes para la versión ("SEC2", "SEC3" o "SEC4") y los 4 restantes para el tamaño de la sección 'contenido'.
- **Contenido**: Esta sección es un fichero regular ELF que alberga el contenido del trustlet. Si la versión es 'SEC4', esta sección estará encriptada.

[8] `https://globalplatform.org/certified-products/samsung-teegris-v4-1`

- **Metadatos**: Contiene el grupo al que pertenece el trustlet. A partir de la versión SEC3, hay un campo adicional que contiene un número de versión. Esta versión es utilizada por el trustelt `root_task` en combinación con el trustlet `ACSD` para gestionar la protección *anti-rollback*. Cada vez que se carga un trustlet SEC3 o SEC4, se extrae el número de versión y se compara con una versión almacenada en el almacenamiento RPMB. Si la versión es inferior a la almacenada, el trustlet no puede cargarse y se devuelve un error. Si es superior, el número de versión contenido en RPMB se actualiza para que coincida con la versión del trustlet, de modo que las copias más antiguas del mismo ya no puedan cargarse.

- **Firma**: Contiene una firma RSA de las secciones anteriores. Esta firma sigue el formato X. 509 y se valida por la TA `ACSD`.

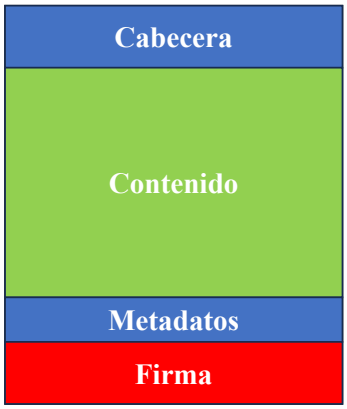


Figura 3.8: Estructura de un trustlet en TEEGRIS. Extraído de: `https://www.riscure.com/tee-security-samsung-teegris-part-1`

Dado que esta solución sigue los estándares de GP, las aplicaciones del NWd implementan la API TEE Client API. Esta API ofrece una interfaz a las aplicaciones para que puedan comunicarse con las TA de forma segura. En esta interfaz, destacan cuatro funciones:

1. **TEEC_OpenSession()**: Cuando se desea establecer una comunicación con una TA, como consecuencia de utilizar esta función, se invoca primero a la función `TA_CreateEntryPoint()` y luego a `TA_OpenSessionEntryPoint()`.

2. **TEEC_InvokeCommand()**: Esta función es la responsable de que se llame a la función `TA_InvokeCommandEntryPoint()`.

3. **TEEC_CloseSession()**: Finalmente, está función es la responsable de finalizar la comunicación. Como resultado, se invocará a la función `TA_CloseSessionEntryPoint()`, y si la sesión en curso fuera la última, también se llamaría a `TA_DestroyEntryPoint()`.

4. **TEEC_RequestCancellation()**: Esta función se utiliza para cancelar una sesión.

Por otro lado, las TAs implementan la API TEE Internal Core API. Esta API se utiliza para desarrollar TAs y ofrece funcionalidades como el manejo de memoria, comunicación con las aplicaciones del NWd o la gestión en el almacenamiento seguro. En la API también se incluyen cinco funciones que gestionan todas las fases del ciclo de vida de una TA, desde su creación hasta su destrucción. A continuación, se especifican estas funciones:

1. **TA_CreateEntryPoint()**: Función constructor que es invocada para crear una nueva instancia.
2. **TA_OpenSessionEntryPoint()**: Esta función es llamada cuando una aplicación de Android desea establecer una sesión. Si la sesión se establece correctamente, la TA puede incluir un puntero (`context`) que será recuperado en las siguientes llamadas a la TA dentro de la sesión.
3. **TA_InvokeCommandEntryPoint()**: Una vez que se ha establecido la sesión y se desea enviar un comando al trustlet, se invocará a esta función. En esta función se recibe el comando a utilizar y sus parámetros.
4. **TA_CloseSessionEntryPoint()**: Esta función es la encargada de cerrar la sesión entre la aplicación de Android y el trustlet. Al invocar esta función, la TA recibirá el `context`, teniendo la obligación de liberar la memoria asociada a este.
5. **TA_DestroyEntryPoint()**: Función destructor que es invocada cuando se desea destruir la instancia creada.

A la hora de solicitar una operación a una TA, esta puede aceptar argumentos. Los argumentos adoptan dos tipos distintos: `memref`, que representa una referencia a memoria, o `value`, que simboliza un valor. En el formato `memref`, se especifica un buffer y su tamaño, mientras que en el formato `value`, se definen directamente los valores de los argumentos [61].

```
typedef union
{
    struct
    {
        void *buffer;
        size_t size;
    } memref;
    struct
    {
        uint32_t a;
        uint32_t b;
    } value;
} TEE_Param;
```

Listing 3.5: Formato de los argumentos aceptado para operar con una TA.

El tipo `memref` es utilizado en aquellas operaciones cuya solicitud/respuesta impliquen el uso de buffers. Las aplicaciones del NWd proporcionan una referencia a un búfer en su propia memoria privada, que TEEGRIS mapeará en el espacio de direcciones del trustlet y rellenará los miembros `buffer` y `size`.

Por otra parte, el tipo `value` se utiliza en operaciones cuya solicitud/respuesta sea proporcionada como valores. Los valores estarán contenidos en los campos `a` y `b`. En las funciones como `TA_InvokeCommandEntryPoint`, existe un argumento etiquetado como `paramTypes` que es utilizado para indicar el tipo de parámetro que se va a recibir. Los tipos de parámetros disponibles se detallan en la siguiente tabla.

Nombre	Tipo	Dirección
TEE_PARAM_TYPE_NONE	-	-
TEE_PARAM_TYPE_VALUE_INPUT	Valor	Entrada
TEE_PARAM_TYPE_VALUE_OUTPUT	Valor	Salida
TEE_PARAM_TYPE_VALUE_INOUT	Valor	Entrada/Salida
TEE_PARAM_TYPE_MEMREF_INPUT	Memoria	Entrada
TEE_PARAM_TYPE_MEMREF_OUTPUT	Memoria	Salida
TEE_PARAM_TYPE_MEMREF_INOUT	Memoria	Entrada/Salida

Tabla 3.1: Tipo de parámetros aceptados por las TAs.

3.5 Google

El último fabricante contemplado en esta lista es Google. A pesar de que Google es más bien conocida por ofrecer otros servicios, también fabrica los dispositivos móviles Pixel [9].

Desde el lanzamiento del dispositivo Pixel 3 en 2018, Google ha fortalecido la seguridad de sus dispositivos móviles mediante la integración de un componente adicional: el SoC conocido como **Titan M** o `citadel` [62]. Este chip fue diseñado específicamente para los dispositivos Pixel, con el objetivo de mejorar significativamente su nivel de seguridad. Al ser un chip externo al SoC principal, Titan M proporciona una capa adicional de protección, mitigando amenazas como Spectre y Meltdown [63].

La arquitectura del SoC se basa en el procesador Arm Cortex-M3 [64], una familia de procesadores que no utiliza una *Memory Management Unit* (MMU), motivo por el cual carece del concepto de memoria virtual y, por ende, no puede incorporar medidas de seguridad como *Address Space Layout Randomization* (ASLR). En lugar de ello, Titan M implementa un esquema de memoria basado en la arquitectura *Protected Memory System Architecture* (PMSA), donde la memoria se divide en diferentes regiones con atributos específicos. A pesar de esta limitación, el firmware del chip implementa una medida de seguridad conocida como *stack canary*. Esta salvaguarda, originalmente diseñada para detectar desbordamientos de memoria en la pila, se utiliza inicializando aleatoriamente el valor del canario y comparándolo antes de retornar de una función. En el caso del firmware de Titan M, el canario se inicializa en la pila de cada proceso con la constante `0xdeadd00d`, convirtiéndolo en un mecanismo de detección de errores [65]. La siguiente figura proporciona una representación visual de los componentes de este chip.

[9] `https://store.google.com/en/category/phones`

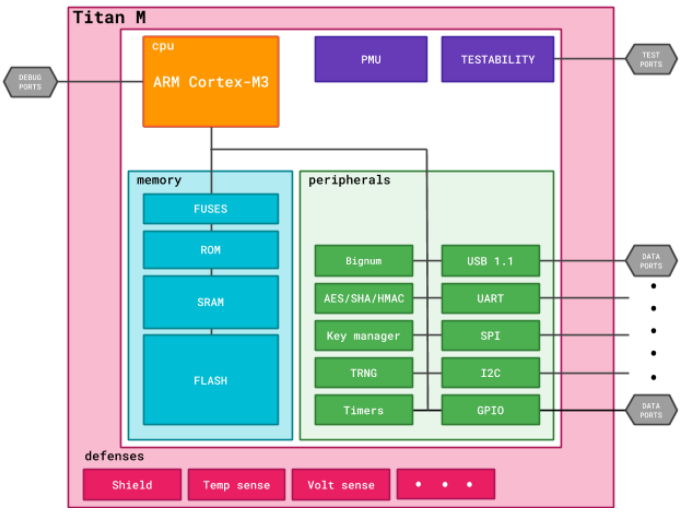


Figura 3.9: Componentes hardware de Titan M. Adaptado de [62].

El chip cuenta con una memoria flash interna y 64 KiB de memoria RAM. Debido al pequeño tamaño de la memoria RAM, el código se ejecuta directamente desde la memoria flash. Durante la ejecución, el chip puede adoptar dos estados de ejecución diferentes: el modo *handler* (privilegiado) y el modo *thread* (privilegiado y no privilegiado, según se configure). Los procesos se ejecutan en modo thread; para cambiar el estado de ejecución, se produce una interrupción software empleando la instrucción `SVC`.

El firmware del chip recibe el nombre en clave de *Nugget OS* y está basado en *Chromium Embedded Controller* (EC) [10], un SO de código abierto para microcontroladores desarrollado por Google. El mapa de memoria del firmware se puede encontrar en el archivo de cabecera `flash_layout` [11]. Este mapa de memoria se forma a partir de cuatro imágenes: dos RO y dos RW. El motivo por el que hay dos imágenes de cada tipo es para poder soportar las actualizaciones A/B [12]. Las imágenes RO contienen el bootloader, que arrancará el SO principal en la partición RW si no se detecta ningún error. Estas imágenes, a pesar de que sus nombres sugieren los permisos de las mismas pueden ser sobreescritas durante una actualización.

Durante su funcionamiento, el chip y el SoC principal interactúan mediante una arquitectura cliente-servidor. A nivel de hardware, esta comunicación se realiza sobre el bus SPI. Cuando un mensaje llega a este bus, se produce una interrupción y se ejecuta la rutina dedicada para atender la petición. Por encima de SPI, se ha implementado un protocolo que define un conjunto de comandos compuestos por una solicitud y una respuesta. El driver SPI en el lado de Titan, reconstruye el comando recibido en diferentes paquetes SPI, y lo copia en la memoria del proceso pertinente. En este punto, el sistema activa un evento que avisa al proceso para que pueda empezar a procesar el mensaje. Durante este procesamiento se extrae el identificador del comando para determinar a qué subrutina invocar de una lista presente en un área global de memoria. Al finalizar la operación se sigue un proceso similar, pero en orden inverso: se escribe la respuesta en un buffer de memoria, para a continuación, notificar al driver (en modo handler) para que envíe al otro extremo la respuesta en forma de *raw bytes*.

Los mensajes intercambiados se codifican empleando *Protocol Buffers* (*Protobuff*). Este tipo de codificación se define como un mecanismo extensible, independiente del lenguaje

[10] `https://chromium.googlesource.com/chromiumos/platform/ec/`
[11] `https://android.googlesource.com/platform/external/nos/host/generic/+/refs/heads/android13-release/nugget/include/flash_layout.h`
[12] `https://source.android.com/docs/core/ota/ab`

y de la plataforma para serializar datos estructurados. El firmware utiliza `nanopb`, una implementación de protobuff para microcontroladores escrita en C [13].

Para facilitar la comunicación entre el chip y el SoC principal, se utilizan varios componentes a nivel de software. Tomando como ejemplo una aplicación que solicite una clave AES al servicio `Strongbox`, la aplicación comenzará la solicitud utilizando la API de `Strongbox` para realizar la llamada específica para obtener una clave AES. Esta llamada será transmitida al Android runtime, que se comunicará con el *daemon* `keystore`, situado por encima de la *Hardware Abstraction Layer* (HAL) y solo es accesible por los componentes internos. Cuando la ejecución llega a HAL, la solicitud se codifica utilizando protobuff y se envía al daemon `citadeld`. Este daemon utiliza el driver `/dev/citadel10` para comunicare con Titan M.

En 2021 se lanzó la siguiente iteración de Titan M: el **Titan M2** [66]. Esta nueva versión del chip se basa en la arquitectura RISC-V. Al igual que ocurría con su predecesor, el Titan M2 es un coprocesador externo que en este caso se integra para funcionar con un SoC llamado Tensor [67].

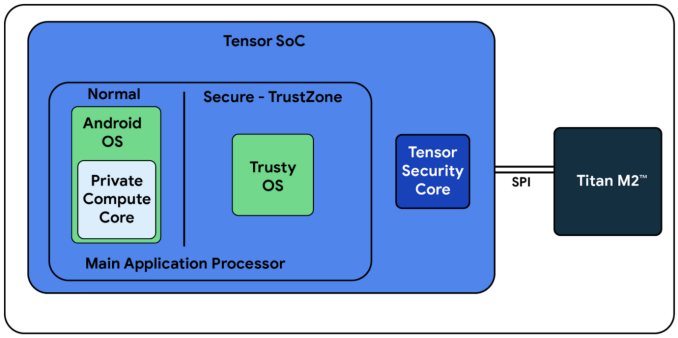


Figura 3.10: Componentes hardware de Titan M2. Adaptado de [68].

El procesador principal del SoC Tensor se basa en la arquitectura Arm y utiliza la tecnología Trustzone. El núcleo Tensor Security Core incluido en el SoC, se define como un subsistema personalizado dedicado a la preservación de la privacidad del usuario. Se diferencia tanto lógica como físicamente del procesador principal y está compuesto de: una CPU dedicada, memoria ROM, memoria *one-time-programmable* (OTP), motor criptográfico, memoria SRAM interna y memoria protegida DRAM. Este subsistema se utiliza para la protección de claves del usuario en tiempo de ejecución, refuerzo del arranque seguro y la interacción con el chip M2 [68]. Por último, destacar que tanto el SO que se ejecuta en Trustzone como en el Tensor Security Core es **Trusty**.

Google Trusty es un SO seguro desarrollado por Google para garantizar la seguridad y la privacidad en dispositivos móviles. Este SO es compatible con los procesadores Arm e Intel. En los procesadores Arm se utiliza la arquitectura de seguridad TrustZone; en cambio, en los procesadores Intel se utiliza la tecnología *Intel Virtualization Technology.*

Internamente, Trusty se compone de tres elementos:

- Un pequeño kernel derivado de Little Kernel [14].
- Un driver para transmitir datos entre entornos de ejecución.
- Una librería de usuario de Android para comunicarse con los trustlets a través del driver anterior.

[13] `https://github.com/nanopb/nanopb`
[14] `https://github.com/littlekernel/lk`

CAPÍTULO 4
Modelo de amenazas

En este capítulo, se presenta una arquitectura modelo y se analizan las posibles amenazas que puedan comprometer la seguridad de la arquitectura. Después de describir los detalles técnicos de la arquitectura se establece un marco de referencia que defina las posibles amenazas y escenarios de ataque que enfrenta el sistema. Este enfoque, conocido como modelo de amenazas o *threat model*, proporciona una comprensión más profunda de las capacidades y motivaciones de los posibles atacantes. En el contexto de la arquitectura presentada, el objetivo consiste en mantener segura la información de inicio de sesión del usuario.

4.1 Arquitectura modelo

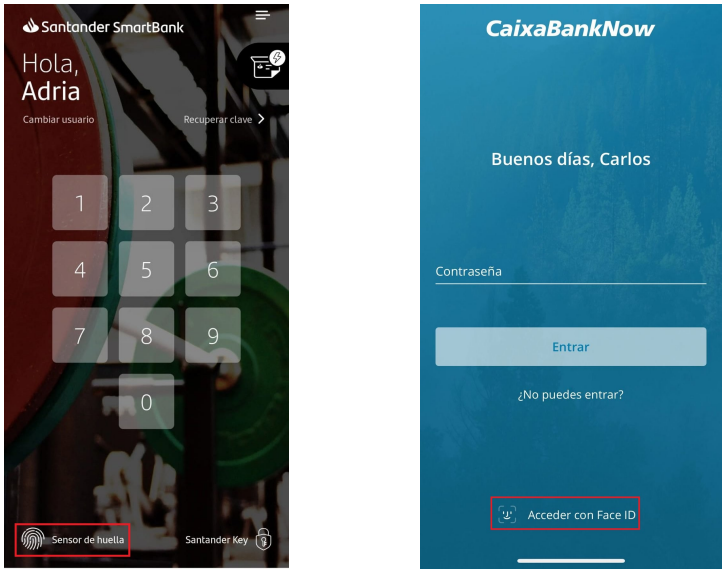
Antes de analizar en detalle los distintos tipos de amenazas a los que se puede enfrentar la siguiente arquitectura, resulta relevante destacar la diversidad de enfoques y necesidades que los desarrolladores de productos consideran al diseñar sus sistemas. Por ejemplo, plataformas ampliamente utilizadas como Apple Pay y Google Pay adoptan enfoques arquitectónicos distintos para garantizar la seguridad [69] [70]. Por ello, en esta sección se presentará inicialmente una arquitectura genérica, que servirá como punto de partida para comprender los principios fundamentales que guían la implementación de sistemas seguros de pago en dispositivos móviles.

En un sistema real, el usuario interactúa con una aplicación bancaria desarrollada por su entidad. Esta aplicación le permite al usuario realizar cualquier tipo de acción relacionado con su cuenta corriente: visualizar su saldo, realizar transferencias, comprar acciones, etc. Para que la aplicación le permita al usuario llevar a cabo cualquiera de estas acciones, primero el usuario deberá autenticarse. La primera vez que el usuario utilice la aplicación deberá introducir su usuario (DNI, correo electrónico, etc.) junto con la contraseña asociada a la cuenta. Para dotar de seguridad a este proceso, se crea una *trusted application* (TA) que se encargará de realizar todos los trámites más sensibles. Para que la TA pueda leer de forma exclusiva la información de inicio de sesión, se configura la pantalla para que sea tratada como un elemento de confianza, de modo que la lectura de los datos se realice dentro del propio TEE. Cabe mencionar que la TA incluye una clave pública incluida en un certificado válido dentro de la TA que se utilizará para firmar todos los mensajes que la TA intercambie con la aplicación del *Normal World* (NWd).

Una vez se han introducido los datos, la aplicación del NWd establecerá una comunicación cifrada con el servidor de autenticación que servirá para verificar los datos de inicio de sesión. En este aspecto, el servidor presenta en primera instancia un certificado respaldado por una autoridad certificadora de confianza, y la aplicación realiza el hash de este y lo compara en una *whitelist* embebida dentro de la TA. Una vez realizada esta com-

paración, el cliente resolverá un desafío propuesto por el servidor utilizando la información de inicio de sesión. El desafío será resuelto por la TA y su respuesta será transmitida al NWd mediante el uso de memoria compartida. La idea es que el usuario se autentique ante el servidor sin la necesidad de transmitir la información de inicio de sesión, de modo que se garantiza que, si un tercero interceptara la comunicación, primero debería descifrarla, y en caso de tener éxito, no sería capaz de extraer ninguna información útil. Otro de los objetivos es la autenticación del servidor por medio de certificados, de modo que el cliente tiene la certeza que está interactuando con un servidor de la entidad bancaria. Tras finalizar este proceso, si el usuario se autentica exitosamente, se le permite el inicio de sesión.

Tras este primer inicio de sesión, es común que la aplicación pregunte al usuario si desea establecer el inicio de sesión mediante el uso de datos biométricos como puede ser la huella dactilar o el rostro facial, como se observa a continuación.



(a) Inicio de sesión mediante huella dactilar.

(b) Inicio de sesión mediante rostro facial.

Figura 4.1: Métodos de autenticación biométricos.

Para que la aplicación pueda establecer dicho método de desbloqueo biométrico, es necesario que el usuario haya configurado previamente el desbloqueo biométrico como una opción de seguridad en su dispositivo móvil. El proceso para habilitar este método de autenticación es bastante sencillo: la aplicación guía al usuario a través de un proceso en el que se le solicita introducir su huella dactilar en el lector biométrico del dispositivo. Una vez que la huella dactilar proporcionada coincide con la huella registrada cuando se configuró el desbloqueo biométrico en el dispositivo, la aplicación se configurará para utilizar el desbloqueo biométrico en los futuros inicios de sesión. Para ofrecer mayor seguridad, en el momento de la lectura de la huella dactilar se configura el lector para que sea tratado como un elemento de confianza, de modo que la lectura sea realizada por la TA.

Al introducir la huella, se genera un token de autenticación que puede ser utilizado ante el trustlet `KeyMaster` para importar claves o generar una nueva clave y asociarla a la aplicación. De este modo, el primer inicio de sesión generará una clave que será presentada al servidor a modo de autenticación cada vez que se utilice el inicio de sesión biométrico.

4.2 Amenazas consideradas

Este modelo asume que el atacante tiene acceso físico al dispositivo y que posee la capacidad de infectar cualquier aplicación instalada en el dispositivo, así como de instalar nuevas aplicaciones. También se considera que el atacante es capaz de vulnerar el kernel, alterando el funcionamiento normal del sistema. Además, se contempla la posibilidad de que el atacante realice ataques a las comunicaciones de red. Cabe destacar que este modelo no incluye ataques físicos sobre el dispositivo como ataques de canal lateral a la caché del procesador, ataques al controlador DMA, ataques sobre interfaces físicas como JTAG o ataques *cold boot*. Tampoco se tienen en cuenta ataques contra las TAs, el sistema operativo seguro y el monitor de seguridad. Los ataques de denegación de servicio (DoS) sobre el dispositivo y las comunicaciones de red tampoco se tendrán en cuenta.

4.2.1. Ataques al kernel

Los ataques al kernel consisten en explotar vulnerabilidades en el núcleo del sistema operativo para obtener privilegios elevados. Una vez el atacante obtiene privilegios elevados de ejecución posee la capacidad de ejecutar código con distintos propósitos: acceder y filtrar información sensible del usuario, modificar configuraciones críticas del sistema, instalar software malicioso al nivel del SO, reemplazar aplicaciones legítimas por versiones maliciosas, desactivar mecanismos de seguridad, interceptar y modificar comunicaciones de red y comunicaciones entre procesos, crear puertas traseras para acceso futuro y manipular los registros de auditoría para ocultar sus acciones.

4.2.2. Ataques userland

Los ataques en el espacio de usuario (*userland*) se centran en explotar vulnerabilidades en las aplicaciones y procesos que se ejecutan con permisos de usuario, en lugar de permisos de sistema. Estos ataques otorgan al atacante la posibilidad de ejecutar código con los privilegios del usuario afectado con diversos fines: acceder a información sensible almacenada por las aplicaciones, robar credenciales de usuario, instalar malware que opere en el contexto del usuario, modificar o eliminar archivos del usuario, interceptar y alterar comunicaciones del usuario y registrar las pulsaciones de teclas para capturar contraseñas.

4.2.3. Ataque de suplantación

Un ataque de suplantación ocurre cuando un atacante obtiene o crea un certificado digital fraudulento y lo utiliza para hacerse pasar por una entidad legítima en Internet, permitiendo al atacante interceptar, manipular o espiar comunicaciones seguras entre los usuarios y el servicio falsificado.

4.2.4. Man-in-the-middle

Un ataque *Man-in-the-Middle* (MiTM) es un tipo de ataque en el que un atacante se posiciona entre dos partes que se están comunicando, interceptando la comunicación entre ellas. Este tipo de ataques están destinados a interceptar la comunicación entre dos partes conectadas a una red, permitiendo al atacante manipular el tráfico interceptado.

4.2.5. Ataque de repetición

Un ataque de repetición es un tipo de ataque en el que un atacante intercepta y retransmite mensajes válidos enviados a través de una red, con el objetivo de engañar al receptor y hacerle realizar acciones no autorizadas, explotando la falta de protección contra la reutilización de mensajes antiguos en el sistema.

4.3 Mitigaciones

En la arquitectura modelo todas las operaciones sensibles se realizan en el TEE y se devuelve el resultado firmado con la clave pública incluida en un certificado válido a la aplicación situada en el NWd para que esta lo pueda transmitir al servidor de autenticación. La TA además maneja la verificación de certificados y la resolución de desafíos, protegiendo estos procesos de cualquier interferencia del espacio de usuario comprometido. Adicionalmente, la configuración del desbloqueo biométrico y la generación de tokens de autenticación se realizan dentro del TEE. Al tratar el lector biométrico como elemento de confianza, se garantiza que los datos biométricos no puedan ser interceptados ni manipulados por aplicaciones maliciosas en el espacio de usuario, ni por el kernel. Aunque es posible interceptar los mensajes intercambiados entre la TA y la aplicación del NWd, su modificación no tendría éxito debido a que dichos mensajes están firmados con la clave pública incluida en un certificado válido dentro de la TA y son verificados con la clave privada de la entidad bancaria. Estas medidas en conjunto aseguran que la arquitectura descrita sea robusta y resistente frente a los ataques descritos en 4.2.1 y 4.2.2.

Por otro lado, el servidor de autenticación presenta un certificado respaldado por una autoridad certificadora de confianza, y la aplicación cliente puede verificar la autenticidad de este certificado mediante la comparación con la whitelist. Al mismo tiempo, el uso de este certificado permite a las dos partes establecer un canal cifrado, donde aún sigue siendo posible interceptar el tráfico, pero no descifrarlo. Por último, durante el proceso de autenticación, el cliente resuelve un desafío propuesto por el servidor utilizando la información de inicio de sesión del usuario. Este desafío y su respuesta son únicos para cada sesión de autenticación y se generan dinámicamente, lo que dificulta que un atacante intercepte y manipule la comunicación sin ser detectado. Todas estas características hacen que la aplicación sea resistentes a las amenazas descritas en 4.2.3, 4.2.4 y 4.2.5.

CAPÍTULO 5
Implementación

En este capítulo se detalla la implementación de TEEBank, una aplicación bancaria diseñada para aprovechar las características de un *Trusted Execution Environment* (TEE). TEEBank tiene como objetivo simular un escenario común: un usuario que inicia sesión en su aplicación bancaria desde un dispositivo móvil. El propósito es ofrecer al usuario un acceso seguro y práctico a sus servicios bancarios a través del dispositivo móvil, utilizando la seguridad proporcionada por el TEE para proteger la información confidencial durante la autenticación.

5.1 Elección TEE

La primera fase del desarrollo de la aplicación involucra la elección de un TEE OS. En este escenario, se han evaluado tres sistemas operativos de código abierto para identificar cuál se adapta de manera óptima a los requisitos del proyecto. A continuación, se ofrece un análisis de cada uno de los SO considerados:

- **Open-TEE**: El primer SO considerado es Open-TEE [1]. Este sistema tiene como objetivo implementar un TEE virtual, independiente del hardware y compatible con la especificación GlobalPlatform. Su propósito es permitir a los desarrolladores crear y depurar *trustlets* utilizando las mismas herramientas que emplean para desarrollar software [71]. El proyecto está liderado por el grupo de investigación Secure Systems Group como parte de sus actividades en el Intel Collaborative Research Institute for Secure Computing. No obstante, al momento de redactar este documento, el proyecto muestra signos de dejadez, evidenciada por la baja actividad del repositorio en los últimos años. Dada esta situación y la falta de una comunidad activa, se ha descartado este SO.

- **Trusty**: El siguiente SO valorado es Trusty [2]. Este SO, desarrollado por Google, está diseñado para funcionar con TrustZone en plataformas Arm o con Intel VT en plataformas x86 de Intel. El sitio web de Trusty ofrece una guía para compilar el sistema operativo y utilizarlo con el emulador Qemu junto con Android, además de proporcionar detalles sobre la API para el desarrollo de trustlets. A pesar de que Trusty fue considerado como el principal candidato para el desarrollo de la aplicación, se descartó debido a la escasa documentación y a la falta de una comunidad establecida en línea.

[1] `https://open-tee.github.io`
[2] `https://source.android.com/docs/security/features/trusty`

- **OP-TEE**: El último SO sopesado es OP-TEE [3]. Este SO está diseñado principalmente para depender de la tecnología TrustZone como el mecanismo de aislamiento de hardware subyacente. A pesar de estar diseñado principalmente para depender de TrustZone, OP-TEE ha sido estructurado de manera que sea compatible con cualquier tecnología de aislamiento apropiada para el concepto y los objetivos de un TEE. Esto incluye la posibilidad de ejecutarse como una máquina virtual o en una CPU dedicada. De acuerdo a estas características, OP-TEE implementa la API TEE Internal Core API v1.3.1 [72], que se expone a los trustlets, y la API TEE Client API v1.0, que describe cómo comunicarse con un TEE. Estas API están definidas en las especificaciones de GlobalPlatform. Por último, pero no menos importante, OP-TEE cuenta con una activa comunidad que contribuye al progreso del proyecto. Debido a la amplia cantidad de recursos en línea y su amplia comunidad, este es el SO con el que se llevará a cabo el desarrollo.

5.2 Trusted Applications

En OP-TEE se distinguen dos tipos de TAs:

- **Pseudo Trusted Applications**: Estas aplicaciones no encajan exactamente en la definición tradicional de TAs, sino que funcionan como interfaces proporcionadas por el núcleo OP-TEE para interactuar con las aplicaciones. Se integran estáticamente en el *blob* del núcleo OP-TEE y se ejecutan en el mismo nivel de privilegio que el código del núcleo.
- **User Mode Trusted Applications**: En contraste, las TAs de este tipo son desarrolladas por los usuarios y cargadas dinámicamente por el núcleo OP-TEE cuando alguna aplicación del *Normal World* (NWd) requiere interactuar con ellas. Se ejecutan en un nivel de privilegio de CPU inferior al del código del núcleo OP-TEE.

Para desarrollar una TA del segundo tipo, es necesario crear la siguiente estructura de directorios y ficheros [4]:

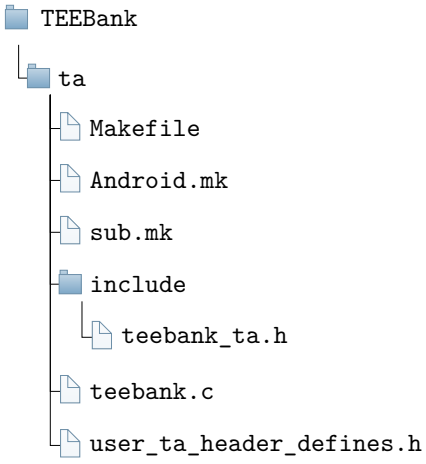


Figura 5.1: Estructura de directorios y ficheros.

La estructura mostrada es similar a la de un proyecto básico con Makefiles. En esta disposición, los archivos de configuración se agrupan en el mismo directorio junto con el código fuente. A continuación, se describen las funciones de cada archivo:

[3] `https://optee.readthedocs.io/en/latest`
[4] `https://optee.readthedocs.io/en/latest/building/trusted_applications.html`

- **Makefile**: Este fichero, además de establecer un par de variables de configuración, definirá en la variable `BINARY` el UUID de la aplicación. También incluye el TA_DEVKIT Makefile para que la aplicación se compile correctamente.
- **Android.mk**: Este fichero se invocaría si se fuera a compilar la TA para que funcione en OP-TEE sobre Android.
- **sub.mk**: Este archivo make enumera las fuentes que hay que compilar.
- **include/teebank_ta.h**: Este archivo de cabecera se exporta al NWd, para que las aplicaciones pueden interactuar con la TA.
- **teebank.c**: Código fuente de la TA. Al menos deberá implementar los siguientes puntos de entrada, como funciones externas:
  - `TA_CreateEntryPoint()`.
  - `TA_DestroyEntryPoint()`.
  - `TA_OpenSessionEntryPoint()`.
  - `TA_CloseSessionEntryPoint()`.
  - `TA_InvokeCommandEntryPoint()`.
- **user_ta_header_defines.h**: Un archivo de cabecera en el cual se definen las propiedades de la TA, como por ejemplo el UUID o la versión de la aplicación, además de los comandos soportados.

5.2.1. Compilación y firma

En OP-TEE las aplicaciones se compilan usando la *toolchain* de Arm, tanto de 32 como de 64 bits, según elija el usuario. Estas toolchains utilizan por defecto el compilador GCC, aunque es posible compilar las aplicaciones utilizando el compilador Clang [5]. El resultado final de compilar una aplicación es un archivo ELF sin símbolos, que contiene el código compilado y enlazado de manera estática. Tras la generación del fichero ELF, se obtendrá un archivo de extensión `ta` cuyo nombre se corresponde con el UUID de la aplicación. Este tipo de archivo formará parte de una de las tres siguientes clases [6]:

- **Legacy TAs**: Este tipo de TAs se firman, pero no se encriptan. A partir de la versión 3.7.0 de OP-TEE, ya no es posible crear este tipo de aplicaciones.
- **Bootstrap TAs**: Estas aplicaciones se firman con la clave utilizada para compilar el *blob* del núcleo OP-TEE, pero no se encriptan. En este proyecto, se desarrolla una aplicación de este tipo.
- **Encrypted TAs**: El último tipo de aplicaciones se manifiesta cuando se establece la variable `CFG_ENCRYPT_TA`. Estas TAs se firman y encriptan utilizando la clave especificada por la variable `TA_ENC_KEY`

A pesar de existir dos tipos diferentes de aplicaciones permitidos, en OP-TEE existen dos cabeceras que son comunes a ambos tipos. La primera cabecera recibe el nombre de `shdr` y proporciona información necesaria para la correcta interpretación y carga de la aplicación, ya que contiene atributos como el tipo de imagen, tamaño, o el algoritmo de firma utilizado. En el siguiente listing, se muestran los campos que componen esta cabecera.

[5] `https://optee.readthedocs.io/en/latest/building/toolchains.html`

[6] `https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html#ree-fs-ta-formats`

```
struct shdr {
    uint32_t magic;
    uint32_t img_type;
    uint32_t img_size;
    uint32_t algo;
    uint16_t hash_size;
    uint16_t sig_size;
};
```

Listing 5.1: Formato de la cabecera `shdr`.

La segunda cabecera recibe el nombre de `shdr_bootstrap_ta` y únicamente alberga el UUID de la aplicación y su versión, como se observa a continuación:

```
struct shdr_bootstrap_ta {
    uint8_t uuid[sizeof(TEE_UUID)];
    uint32_t ta_version;
};
```

Listing 5.2: Estructura de la cabecera `shdr_bootstrap_ta`.

Para firmar la aplicación, el código fuente de OP-TEE incluye una clave privada RSA-2048 que será utilizada a menos que el usuario reemplace dicha clave. Para realizar el proceso de firma, actualmente existen dos esquemas posibles:

- Esquema de firma PKCS#1 con relleno PSS y función de generación de máscara MGF1 con SHA-256 como algoritmo hash. Para el desarrollo de TEEBank, este ha sido el esquema utilizado.
- Esquema de firma PKCS#1 v1.5 con relleno clásico y la función hash SHA-256.

Antes de realizar el firmado, primero se realiza un hash. En el caso de las TA del tipo bootstrap, el hash se realiza como sigue: $H(shdr \mid\mid bootstrap\_shdr \mid\mid sttriped\_elf)$. Una vez se ha obtenido el hash, se aplica el esquema de firma para obtener una firma. Tras obtener la firma, se concatenan las cabeceras, el hash, la firma y el fichero ELF para obtener el fichero `ta` de la aplicación. En la siguiente figura se muestra la salida del proceso `make`, al momento de generar el archivo `ta`.

```
>>> optee_examples_ext 1.0 Building
PATH="/home/user/optee/out-br/per-package/optee_examples_ext/host/bin:/home/user/optee/out
al/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin:/home
Consolidate compiler generated dependencies of target teebank
[100%] Built target teebank
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
Building /home/user/optee/out-br/build/optee_examples_ext-1.0/TEEBank/ta/Makefile
make[3]: warning: jobserver unavailable: using -j1.  Add '+' to parent make rule.
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
  CC      out/teebank.o
-- Found OpenSSL: /home/user/optee/out-br/per-package/optee_test_ext/host/aarch64-buildroo
  GEN     out/dyn_list
  LD      out/fb8d0e80-518c-4f50-bb22-3a583d5013e3.elf
  OBJDUMP out/fb8d0e80-518c-4f50-bb22-3a583d5013e3.dmp
  OBJCOPY out/fb8d0e80-518c-4f50-bb22-3a583d5013e3.stripped.elf
  SIGN    out/fb8d0e80-518c-4f50-bb22-3a583d5013e3.ta
```

Figura 5.2: Generación del archivo ta tras la compilación, enlazado y firma.

5.2.2. Verificación

La carga de una TA conlleva implícitamente un proceso de verificación criptográfica, garantizando la integridad y autenticidad de la aplicación antes de su ejecución. Tal y como se ha descrito en 2.3, este proceso parte de un arranque seguro, en el cual se establece la cadena de confianza. En OP-TEE, por defecto se utiliza una clave preestablecida en el código fuente para establecer la raíz de confianza [7], como se ilustra a continuación:

```
/*
 * Override these in your platform code to really fetch device-unique
 * bits from e-fuses or whatever.
 *
 * The default implementation just sets it to a constant.
 */
__weak TEE_Result tee_otp_get_hw_unique_key(struct tee_hw_unique_key *hwkey)
{
    memset(&hwkey->data[0], 0, sizeof(hwkey->data));
    return TEE_SUCCESS;
}
```

Listing 5.3: Función utilizada para establecer la raíz de confianza.

En un producto listo para su uso en un entorno real, esta función debería ser sustituida para obtener la clave de un *eFuse* o de una memoria de solo lectura. En caso de haber configurado OP-TEE para que utilice la arquitectura Armv8-A, es posible configurar el arranque seguro para que utilice el firmware de autenticación Trusted Firmware-A [8].

Después de establecer la cadena de confianza que posibilite el arranque de OP-TEE, se procede a la carga de las TAs. La carga de una TA se efectuará cuando una aplicación de usuario busca establecer una sesión con dicha TA por primera vez. Este proceso de carga incluye la verificación de la aplicación. La verificación implica aplicar la clave pública sobre la firma y comparar el resultado con el hash de la aplicación. Si la comparación es exitosa, la TA se carga con éxito, permitiendo que cualquier aplicación de usuario establezca sesiones con ella. En el anexo A se incluye un script que permite extraer las distintas partes que forman la TA: las dos cabeceras, la firma y el archivo ELF sin símbolos. En el anexo B se proporciona un script para verificar una TA. Este script toma como entrada las partes extraídas con el script anterior y aplica el proceso de firma mencionado para verificar si la TA es válida.

5.3 Arquitectura de la aplicación

La aplicación desarrollada toma como referencia la arquitectura expuesta en 4.1, adaptándola para satisfacer los requisitos específicos motivados por la naturaleza académica y experimental del proyecto. En este aspecto se han introducido una serie de cambios con el objetivo de ofrecer una arquitectura simple pero equivalente a la modelo, de modo que permita un estudio detallado, al mismo tiempo que ofrezca la posibilidad de llevar a cabo una experimentación ágil. Los cambios introducidos están especialmente diseñados para simplificar el desarrollo y centrarse en los aspectos fundamentales del estudio. Estos cambios se pueden resumir en los siguientes puntos:

[7] https://optee.readthedocs.io/en/latest/architecture/porting_guidelines.html#hardware-unique-key

[8] https://optee.readthedocs.io/en/latest/architecture/secure_boot.html#secure-boot

- Se ha eliminado el canal cifrado entre el servidor de autenticación y la aplicación de Android, así como el uso de certificados de confianza por parte del servidor de autenticación.
- Debido a la falta de un dispositivo lector de huellas dactilares, se simulará la lectura de datos biométricos mediante la introducción de un *token*.
- A causa de no poder configurar el back end de la aplicación para que realice la lectura de los datos de inicio de sesión directamente en el TEE, estos datos serán leídos por el front end y transmitidos al TEE.

Los componentes principales de la aplicación se dividen en tres partes, las cuales están diseñadas para interactuar de manera conjunta:

- **Front end**: Interfaz gráfica de usuario que se ejecuta en un dispositivo Android emulado. La decisión de emular el dispositivo Android en esta implementación se basa en consideraciones prácticas y de eficiencia que impulsan el desarrollo del proyecto. En primer lugar, la emulación ofrece una manera conveniente y eficiente de realizar pruebas, eliminando la necesidad de depender de un dispositivo físico dedicado, lo que reduce los costos y la complejidad asociados con la adquisición y configuración de hardware específico. Esto es especialmente beneficioso en un entorno de desarrollo donde se requiere flexibilidad y agilidad para iterar y probar diferentes configuraciones y escenarios. Además, la emulación permite consolidar todo el entorno de desarrollo en una única máquina, simplificando la configuración y el despliegue.
- **Back end**: Elemento principal de la aplicación que se emula mediante Qemu. La elección de Qemu como herramienta de emulación en esta implementación se basa en varios factores clave que influyen en el desarrollo y la eficiencia del proyecto. En primer lugar, Qemu es reconocido por su versatilidad y potencia como emulador y virtualizador de código abierto. Esto permite desarrollar y probar sistemas operativos y aplicaciones en una amplia gama de arquitecturas, incluyendo AArch64, que es esencial para el proyecto centrado en OP-TEE. La capacidad de ejecutar y emular sistemas completos de otras arquitecturas en un entorno de desarrollo local sin depender de hardware externo es un aspecto fundamental. En el contexto del trabajo, donde OP-TEE presenta requisitos específicos de hardware y arquitectura, esta característica de Qemu es especialmente valiosa, ya que permite desarrollar, depurar y probar aplicaciones sobre OP-TEE de manera eficiente y sin la necesidad de adquirir hardware físico compatible. Además, la flexibilidad de Qemu para adaptarse a diferentes entornos de desarrollo y su amplia comunidad de usuarios y desarrolladores proporcionan un respaldo sólido y recursos adicionales para resolver problemas y optimizar el proceso de desarrollo. A nivel interno, el back end se subdivide en dos componentes:
  - **Proxy**: Aplicación localizada en el NWd, que sirve como puente entre el front end, el trustlet y el servidor de autenticación. Sus funciones principales consisten en respaldar el proceso de *enroll* en el TEE y administrar la autenticación ante el servidor de autenticación.
  - **Trustlet**: Aplicación localizada en el *Secure World* (SWd). El trustlet se encargará de generar y almacenar el material criptográfico necesario para autenticar al usuario una vez se haya producido el primer inicio de sesión.
- **Servidor de autenticación**: Servidor encargado del proceso de autenticación. Este servidor de autenticación pretende simular una entidad real para validar las credenciales de usuario de manera controlada. Al emular el servidor de autenticación, se

puede reproducir de manera precisa y controlada los diferentes escenarios de autenticación que pueden surgir durante el desarrollo de la aplicación.

Con un enfoque centrado en la experimentación, la infraestructura de comunicación de la aplicación se caracteriza por su simplicidad, prescindiendo del uso de criptografía debido a las características mencionadas previamente. Durante el ciclo de vida de la aplicación, se establecen las siguientes direcciones de comunicación:

- **Comunicación entre front end y proxy**: La comunicación entre el front end y el back end se realiza a través de un canal TCP sin cifrar.
- **Comunicación entre proxy y servidor de autenticación**: La comunicación entre el back end y el servidor de autenticación se realiza a través de un canal TCP no cifrado.
- **Comunicación entre proxy y trustlet**: La comunicación entre el proxy y su contraparte se realizará mediante buffers compartidos de memoria. En estos buffers se especificará un `command_id` que servirá para solicitar los servicios de enroll o autenticación.

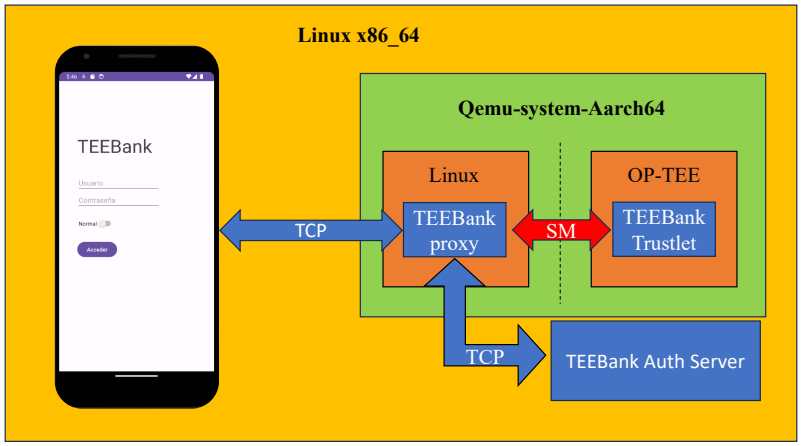


Figura 5.3: Comunicación de los diferentes elementos de TEEBank.

5.3.1. Front end

El front end está formado por una aplicación Android que se ejecuta en el emulador Android Emulator, disponible en el entorno de programación de Android Studio. Esta aplicación forma la interfaz gráfica de usuario que se organiza en varias vistas.

En la primera vista se da soporte al inicio de sesión mediante la combinación de usuario y contraseña. El campo “Usuario” se corresponde con el DNI de la persona titular de una cuenta bancaria, mientras que el campo “Contraseña” se corresponde con un código numérico de cuatro dígitos. Al introducir estos datos se generará un JSON con la información introducida, que será enviado al back end para realizar el proceso de autenticación. Si la autenticación transcurre sin ningún inconveniente, se procede con el inicio de sesión y se le presenta al usuario un token de acceso que podrá ser utilizado en los sucesivos inicios de sesión.

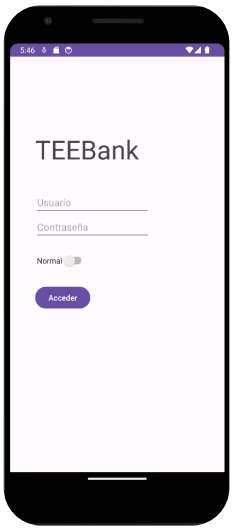


Figura 5.4: Vista inicio de sesión mediante usuario y contraseña.

La segunda vista da soporte al inicio de sesión mediante autenticación biométrica. Al no ser posible desarrollar la aplicación con soporte para el uso de datos biométricos, se propone una vista en la cual se introducirá el token obtenido durante el primer inicio de sesión. En línea con el escenario anterior, este token se enviará al back end para verificar su registro en el sistema, imitando de esta manera la validación de la lectura de la huella dactilar.

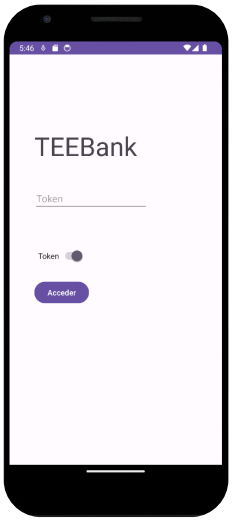


Figura 5.5: Vista inicio de sesión mediante token.

La última vista únicamente muestra el mensaje “Bienvenido” junto con el nombre del titular de la cuenta. El acceso a esta vista garantiza que se ha iniciado sesión de manera correcta. La única acción disponible en esta pantalla es el cierre de sesión.

Figura 5.6: Vista de sesión iniciada.

5.3.2. Back end

El back end de la aplicación consta de dos componentes: un proxy situado en el NWd y un trustlet ubicado en el SWd.

Proxy

El proxy es el elemento central de la aplicación. Este componente es el nexo de unión entre las distintas partes. Por un lado, atiende las solicitudes del front end para recibir los datos introducidos por el usuario e iniciar el proceso de autenticación en colaboración con el servidor de autenticación. Si la petición se corresponde con un inicio de sesión utilizando el usuario y la contraseña, el proxy implementa un cliente *Salted Challenge Response Authentication Mechanism-SHA-256* (SCRAM-SHA-256) [73]. La familia de mecanismos de autenticación SCRAM, utiliza un enfoque basado en desafío y respuesta utilizando la contraseña del usuario para autenticar a un usuario ante un servidor, sin la necesidad de transmitir la contraseña. Por el contrario, si los datos recibidos en la petición se corresponden con un token, entonces el proxy presentará un mensaje cifrado por la TA al servidor a modo de autenticación. Cabe mencionar que la comunicación entre el proxy y la TA pasa por el kernel de Linux en el NWd, que redirigirá las peticiones al monitor seguro para que este las transmita al kernel de OP-TEE para que redirija la petición a la TA, como se observa la siguiente figura.

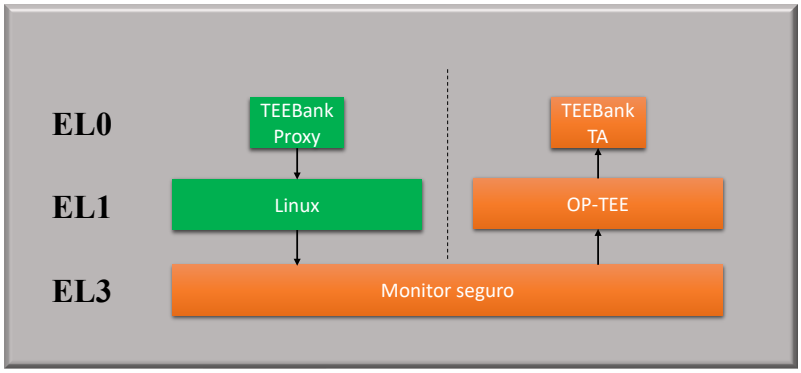


Figura 5.7: Comunicación entre el proxy y la TA.

Por otro lado, el proxy establecerá comunicación con el trustlet para que genere una clave simétrica que se utilizará en el inicio de sesión mediante token. También establecerá comunicación con el trustlet cuando se inicie sesión utilizando el token, para que este último cifre un mensaje que será presentado al servidor de autenticación.

Trustlet

El trustlet se encarga de las operaciones criptográficas. Durante el primer inicio de sesión y después de una autenticación exitosa, el proxy se comunicará con el trustlet para generar una clave simétrica (k). Una vez se genere la clave, esta será transmitida al servidor de autenticación. El propósito es que, al iniciar sesión utilizando el token, este desbloquee la clave k y se cifre el mensaje (m) "Hello server!" con dicha clave, para autenticar al usuario de manera similar a como ocurre en un dispositivo móvil cuando se utiliza la huella dactilar. El uso de criptografía simétrica en lugar de asimétrica viene motivado por permitir un estudio detallado, al mismo tiempo que se ofrece la posibilidad de llevar a cabo una experimentación ágil.

La clave simétrica se genera a partir del usuario y contraseña introducidos durante el primer inicio de sesión. Además, para ofrecer mayor seguridad se vinculará la clave simétrica al dispositivo en el cual fue generada, de modo que descifrar este mensaje fuera del dispositivo resulte infructuoso. Para lograr este fin, se incorporará la clave privada del TEE (P_K^-). Debido a que OP-TEE soporta claves AES GCM de 128 y 256 bits, la clave k se obtendrá a partir del resultado producido al aplicar el algoritmo SHA-256 de la siguiente forma: k = SHA-256($usuario$ || P_K^- || SHA-256($contraseña$)).

Una vez finalizada la operación anterior, el trustlet generará un token aleatorio compuesto de ocho caracteres alfanuméricos en el rango [0-9,a-f]. Este token se le devolverá al usuario para que lo utilice en los sucesivos inicios de sesión. Cuando el usuario intente iniciar sesión utilizando el token, este será utilizado por el trustlet para desbloquear la clave k, de modo similar a como funciona la autenticación biométrica en un dispositivo real.

5.3.3. Servidor de autenticación

La última pieza de la aplicación es un servidor que simula la entidad bancaria TEE-Bank. El objetivo de este servidor es autenticar a los usuarios que intenten iniciar sesión en la aplicación. Si el usuario no está registrado en el dispositivo, el servidor atenderá las peticiones del proxy para autenticar al usuario utilizando el mecanismo SCRAM-SHA-256. Si la autenticación resultase exitosa, el proxy enviará la clave simétrica generada por el trustlet para que, en los sucesivos inicios de sesión, el servidor pueda descifrar el mensaje "Hello server!". Por otra parte, si el usuario ya estuviera vinculado en el dispositivo, el servidor esperaría recibir el mensaje cifrado, que intentaría desbloquear con la clave simétrica obtenida durante el primer inicio de sesión del usuario.

CAPÍTULO 6
Conclusiones y trabajo futuro

El resultado de este proyecto es un estudio exhaustivo que aporta una visión integral de la tecnología *Trusted Execution Environment* (TEE) en dispositivos móviles, destacando su importancia, aplicaciones prácticas y beneficios. Para tratar de entender que mejoras ofrece el uso de esta tecnología se ha propuesto un caso de uso simple y frecuente en el día a día. Los resultados alcanzados se pueden resumir en los siguientes puntos:

- Este estudio ha proporcionado una visión integral sobre la tecnología TEE en dispositivos móviles, destacando su relevancia, aplicaciones prácticas y beneficios. Los hallazgos resaltan la importancia del TEE para garantizar la seguridad y protección de datos sensibles en el contexto de dispositivos móviles, así como su papel fundamental en el proceso de arranque seguro para preservar la integridad del sistema operativo y las aplicaciones.
- Se ha profundizado en la diversidad de implementaciones de TEE realizadas por diversos fabricantes, subrayando la variedad de enfoques y soluciones adoptadas para asegurar la seguridad de sus dispositivos.
- Se ha propuesto un modelo de amenazas que identifica posibles riesgos y vulnerabilidades en el contexto de la implementación de TEE en dispositivos móviles, ofreciendo una guía para mitigar estos riesgos y fortalecer la seguridad de las aplicaciones.
- Se ha presentado un caso de uso concreto que ilustra los beneficios y la aplicabilidad práctica de un TEE en una aplicación móvil. Se evidencia cómo la utilización de un TEE puede mejorar la seguridad y la privacidad de los datos del usuario.

6.1 Trabajo futuro

Para futuras investigaciones, se plantean varias líneas de trabajo prometedoras que pueden ampliar y mejorar el alcance y la utilidad de la tecnología TEE en dispositivos móviles:

- **Exploración de nuevos casos de uso**: Investigar y desarrollar nuevos casos de uso prácticos en sectores como la salud, las finanzas o el Internet de las Cosas (IoT), demostrando cómo el TEE puede beneficiar aplicaciones críticas en estos dominios.
- **Integración con otras tecnologías de seguridad**: Estudiar cómo los TEEs pueden integrarse con otras tecnologías de seguridad, como blockchain, inteligencia artificial y *machine learning*, para desarrollar soluciones más robustas y complejas.
- **Evaluación continua de vulnerabilidades**: Implementar programas de evaluación continua para detectar y mitigar nuevas vulnerabilidades en las implementaciones de TEE, manteniendo así la seguridad de los dispositivos móviles actualizada y efectiva contra amenazas emergentes.
- **Políticas de privacidad y gobernanza de datos**: Explorar cómo los TEEs pueden ser utilizados para implementar y reforzar políticas de privacidad y gobernanza de datos, asegurando un manejo ético y seguro de la información sensible.
- **Investigación en recuperación de fallos**: Desarrollar y probar mecanismos para la recuperación de fallos en sistemas que utilizan TEEs, garantizando la continuidad y resiliencia del servicio ante incidentes.

Bibliografía

[1] A. Kivva, "The mobile malware threat landscape in 2023." https://securelist.com/mobile-malware-report-2023/111964/, 2024. [En línea] Consulta: 26 de febrero de 2024.

[2] E. Calle, "Los mossos avisan de una nueva técnica de 'smishing': "vigila con sms como este"." https://www.elperiodico.com/es/sociedad/20240205/mossos-avisan-nueva-tecnica-smishing-vigila-sms-dv-97482247, 2024. [En línea] Consulta: 26 de febrero de 2024.

[3] A. Higuera, "Así es grandoreiro, el peligroso troyano bancario que ha sido desarticulado con ayuda de españa." https://www.20minutos.es/tecnologia/ciberseguridad/descubre-grandoreiro-troyano-bancario-desarticulado-5214917, 2024. [En línea] Consulta: 26 de febrero de 2024.

[4] J. Alcalde, "Pegasus, el espía perfecto: cómo funciona, cuánto cuesta y por qué es casi indetectable." https://www.larazon.es/espana/20220508/gc7lpsr7drffteanpkpexmytnm.html, 2022. [En línea] Consulta: 26 de febrero de 2024.

[5] H. Nasir, "The rise of arm in computing: More than just mobile?." https://www.xda-developers.com/rise-arm-computing-more-than-mobile/, 2024. [En línea] Consulta: 26 de febrero de 2024.

[6] S. Matala, T. Nyman, and N. Asokan, "Historical insight into the development of mobile tees." https://blog.ssg.aalto.fi/2019/06/historical-insight-into-development-of.html, 2019. [En línea] Consulta: 19 de julio de 2023.

[7] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security enabling trusted computing in embedded systems," *Information Quarterly*, vol. 3, no. 4, 2004.

[8] O. Limited, "Advanced trusted environment: Omtp tr1," tech. rep., OMTP, 2009.

[9] G. D. Technology, "Tee client api specification," tech. rep., GlobalPlatform, 2010.

[10] G. Technology, "Introduction to trusted execution environments," tech. rep., GlobalPlatform, 2018.

[11] G. Technology, "Tee system architecture v1.3," tech. rep., GlobalPlatform, 2022.

[12] D. Lee, *Building Trusted Execution Environments*. PhD thesis, EECS Department, University of California, Berkeley, May 2022.

[13] M. Sabt, M. Achemlal, and A. Bouabdallah, "The dual-execution-environment approach: Analysis and comparative evaluation," in *ICT Systems Security and Privacy*

Protection (H. Federrath and D. Gollmann, eds.), (Cham), pp. 557–570, Springer International Publishing, 2015.

[14] W. Gayde, "How arm came to dominate the mobile market." https://www.techspot.com/article/1989-arm-inside/, 2020. [En línea] Consulta: 21 de agosto de 2023.

[15] A. Holdings, "Learn the architecture - introducing the arm architecture," tech. rep., ARM, 2023.

[16] A. Holdings, "Cortex-a8 technical reference manual," tech. rep., ARM, 2010.

[17] A. Holdings, "Arm architecture reference manual for a-profile architecture," tech. rep., ARM, 2023.

[18] A. Holdings, "Arm cortex-a series programmer's guide for armv8-a," tech. rep., ARM, 2015.

[19] A. Holdings, "Learn the architecture - trustzone for aarch64," tech. rep., ARM, 2021.

[20] A. Holdings, "Arm architecture reference manual armv7-a and armv7-r edition," tech. rep., ARM, 2018.

[21] A. Holdings, "Cortex-a5," tech. rep., Arm, 2016.

[22] A. Holdings, "Arm architecture reference manual security extensions supplement," tech. rep., ARM, 2005.

[23] A. Holdings, "Arm security technology building a secure system using trustzone technology," tech. rep., ARM, 2009.

[24] A. Holdings, "Learn the architecture - aarch64 exception model," tech. rep., ARM, 2022.

[25] A. Holdings, "Smc calling convention system software on arm platforms," tech. rep., ARM, 2013.

[26] N. Pasham, "Demystifying arm trustzone for microcontrollers (and a note on rust support)." https://medium.com/swlh/demystifying-arm-trustzone-for-microcontrollers-and-a-note-on-rust-support-54efc62c290, 2020. [En línea] Consulta: 14 de septiembre de 2023.

[27] A. Holdings, "Amba axi protocol specification," tech. rep., ARM, 2023.

[28] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," vol. 51, (New York, NY, USA), Association for Computing Machinery, jan 2019.

[29] R. Wilkins and T. Nixon, "The chain of trust," 2016.

[30] C. Shepherd, "Extract forensic information for leas from encrypted smartphones," tech. rep., Royal Holloway, University of London, 2021.

[31] D. Handley and C. Garcia-Tobin, "Trusted firmware deep dive." https://www.linaro.org/app/resources/Connect%20Events/Trusted_Firmware_Deep_Dive_v1.0_.pdf, 2013. [En línea] Consulta: 13 de abril de 2024.

[32] V. Zimmer and M. Krau, "Establishing the root of trust," 2016.

[33] G. Technology, "Root of trust definitions and requirements," tech. rep., GlobalPlatform, 2022.

[34] F. Khalid and A. Masood, "Vulnerability analysis of qualcomm secure execution environment (qsee)," *Computers & Security*, vol. 116, p. 102628, 2022.

[35] A. W. Dent, "Secure boot and image authentication," tech. rep., Qualcomm, 2019.

[36] R. Nakamoto, "Secure boot and image authentication," tech. rep., Qualcomm, 2016.

[37] N. Elenkov, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*, ch. 10, pp. 258–267. No Starch Press, 2014.

[38] N. Elenkov, "Revisiting android disk encryption." `https://nelenkov.blogspot.com/2014/10/revisiting-android-disk-encryption.html`, 2014. [En línea] Consulta: 28 de noviembre de 2023.

[39] A. Visconti, O. Mosnáček, M. Broz, and V. Matyas, "Examining pbkdf2 security margin — case study of luks," *Journal of Information Security and Applications*, vol. 46, pp. 296–306, 06 2019.

[40] C. Percival, "Stronger key derivation functions via sequential memory-hard functions," 01 2009.

[41] G. Beniamini, "Extracting qualcomm's keymaster keys - breaking android full disk encryption." `https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html`, 2016. [En línea] Consulta: 28 de noviembre de 2023.

[42] Guillaume, "A glimpse of ext4 filesystem-level encryption." `https://blog.quarkslab.com/a-glimpse-of-ext4-filesystem-level-encryption.html`, 2015. [En línea] Consulta: 05 de enero de 2024.

[43] D. G. Bradley, M. Baumann, R. van Galen, and R. de Vries, "Android 7 file based encryption and the attacks against it," 2017.

[44] M. R. Bellom and D. Melotti, "Android data encryption in depth." `https://blog.quarkslab.com/android-data-encryption-in-depth.html`, 2023. [En línea] Consulta: 14 de diciembre de 2023.

[45] B. Muthukumaran, "File based encryption [qualcomm snapdragon 855 mobile platform and beyond]," tech. rep., Qualcomm, 2019.

[46] S. Makkaveev, "Researching xiaomi's tee to get to chinese money." `https://research.checkpoint.com/2022/researching-xiaomis-tee/#single-post`, 2022. [En línea] Consulta: 01 de agosto de 2023.

[47] D. Berard, "Kinibi tee: Trusted application exploitation." `https://www.synacktiv.com/en/publications/kinibi-tee-trusted-application-exploitation.html`, 2018. [En línea] Consulta: 02 de agosto de 2023.

[48] M. Markstedter, "Trustonic's kinibi tee implementation." `https://azeria-labs.com/trustonics-kinibi-tee-implementation/`, 2020. [En línea] Consulta: 02 de agosto de 2023.

[49] G. Technology, "Tee internal api specification," tech. rep., GlobalPlatform, 2011.

[50] A. Adamski, J. Guilbon, and M. Peterlin, "A deep dive into samsung's trustzone (part 1)." `https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html`, 2021. [En línea] Consulta: 20 de agosto de 2023.

[51] Q. Technologies, "Qualcomm® trusted execution environment (tee) v5.8 on qualcomm® snapdragon™ 865 security target lite," tech. rep., Qualcomm, 2021.

[52] G. Beniamini, "Qsee privilege escalation vulnerability and exploit (cve-2015-6639)." https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016. [En línea] Consulta: 26 de julio de 2023.

[53] T. Zahavi-Brunner, "Attacking the android kernel using the qualcomm trustzone." https://tamirzb.com/attacking-android-kernel-using-qualcomm-trustzone, 2022. [En línea] Consulta: 26 de julio de 2023.

[54] S. Makkaveev, "The road to qualcomm trustzone apps fuzzing." https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/, 2019. [En línea] Consulta: 27 de julio de 2023.

[55] G. Beniamini, "Exploring qualcomm's secure execution environment." https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, 2016. [En línea] Consulta: 27 de julio de 2023.

[56] Apple, "Apple platform security," tech. rep., Apple, 2022.

[57] T. Mandt, M. Solnik, and D. Wang, "Demystifying the secure enclave processor." https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf, 2016. [En línea] Consulta: 11 de agosto de 2023.

[58] S.-Y. Y. Manu Gulati, Michael J. Smith, "Security enclave processor for a system on a chip," U.S. Patent 8 832 465, Sep. 2014.

[59] E. Sanfelix, "Tee exploitation on samsung exynos devices (i/iv) : Introduction." https://labs.bluefrostsecurity.de/blog/2019/05/27/tee-exploitation-on-samsung-exynos-devices-introduction/, 2019. [En línea] Consulta: 17 de agosto de 2023.

[60] F. Menarini, "Breaking tee security part 1: Tees, trustzone and teegris." https://www.riscure.com/tee-security-samsung-teegris-part-1/, 2021. [En línea] Consulta: 10 de julio de 2023.

[61] F. Menarini, "Breaking tee security part 2: Exploiting trusted applications (tas)." https://www.riscure.com/tee-security-samsung-teegris-part-2/, 2021. [En línea] Consulta: 20 de agosto de 2023.

[62] 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/, 2018. [En línea] Consulta: 02 de agosto de 2023.

[63] B. Barrett, "The tiny chip that powers up pixel 3 security." https://www.wired.com/story/google-titan-m-security-chip-pixel-3/, 2021. [En línea] Consulta: 07 de agosto de 2023.

[64] 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, 2018. [En línea] Consulta: 07 de agosto de 2023.

[65] D. Melotti, M. Rossi-Bellom, and A. Continella, "Reversing and fuzzing the google titan m chip," in *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'21, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, 2022.

[66] C. Wankhede, "What is the titan m2 security chip in google's pixel phones?." https://www.androidauthority.com/titan-m2-google-3261547/, 2023. [En línea] Consulta: 02 de agosto de 2023.

[67] J. Hildenbrand and H. Jonnalagadda, "Google tensor soc: Everything you need to know." https://www.androidcentral.com/google-tensor, 2021. [En línea] Consulta: 07 de agosto de 2023.

[68] D. Kleidermacher, J. Seed, B. Barbello, and S. Somogyi, "Pixel 6: Setting a new standard for mobile security." https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html, 2021. [En línea] Consulta: 07 de agosto de 2023.

[69] R. Glushach, "Secure mobile payments: Behind the scenes how apple pay and google pay protect your sensitive card info." https://romanglushach.medium.com/secure-mobile-payments-behind-the-scenes-how-apple-pay-and-google-pay-protect-your-sensitive-card-be7a24ddf392, 2023. [En línea] Consulta: 26 de febrero de 2024.

[70] M. Szczepanik, "A deep dive into google pay and apple pay." https://medium.com/mobilepeople/a-deep-dive-into-google-pay-and-apple-pay-d56dab7194a0, 2023. [En línea] Consulta: 26 de febrero de 2024.

[71] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-TEE – an open virtual trusted execution environment," tech. rep., Aalto University, 2015.

[72] G. Technology, "Tee internal core api specification," tech. rep., GlobalPlatform, 2021.

[73] A. Menon-Sen, A. Melnikov, N. Williams, and C. Newman, "Salted Challenge Response Authentication Mechanism (SCRAM) SASL and GSS-API Mechanisms." RFC 5802, July 2010.

APÉNDICE A

Script para extraer las partes de una Trusted Application

```
#!/usr/bin/bash

: '
 * This program extracts the different parts of
 * an OP-TEE Trusted Application.
 * Author: Josep Comes <jocosan6@inf.upv.es>
 * Date: 15/01/2024
'

# https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html
header_size=20
header_bootstrap_size=20
output_path="$PWD"
ta_path=""

extract_shdr() {
  dd if="$1" bs=1 count="$header_size" of="$2/shdr.bin" >/dev/null 2>&1
}

extract_hash() {
  temp_file=$(mktemp)
  dd if="$1" bs=2 count=1 skip=8 conv=swab of="$temp_file" >/dev/null 2>&1
  size=$(xxd -p -l 2 "$temp_file")
  declare -g hash_size=$(printf "%d" "$((16#$size))")
  dd if="$1" bs=1 count="$hash_size" skip="$header_size" of="$2/hash.bin" >/dev/null 2>&1
  rm -f $tempfile
}

extract_digest() {
  temp_file=$(mktemp)
  to_skip=$((header_size + hash_size))
  dd if="$1" bs=2 count=1 skip=9 conv=swab of="$temp_file" >/dev/null 2>&1
  size=$(xxd -p -l 2 "$temp_file")
  declare -g signature_size=$(printf "%d" "$((16#$size))")
  dd if="$1" bs=1 count="$signature_size" skip="$to_skip" of="$2/signature.bin" >/dev/null
       2>&1
  rm -f "$tempfile"
}

extract_bootstrap() {
  to_skip=$((header_size + hash_size + signature_size))
  dd if="$1" bs=1 count="$header_bootstrap_size" skip="$to_skip" of="$2/bootstrap.bin" >/
      dev/null 2>&1
}
```

```
extract_elf() {
  to_skip=$((header_size + hash_size + signature_size + header_bootstrap_size))
  dd if="$1" bs=1 skip="$to_skip" of="$2/stripped_elf.elf" >/dev/null 2>&1
}

usage() {
  echo "Usage: $0 [-h] [-o path] [-p path]" 1>&2
  echo "-h          Display this help message and exit"
  echo "-o path     Specify the output path"
  echo "-p path     Specify the path where the TA is located"
  exit $1
}

# Parse command-line arguments
while getopts ":o:p:" opt; do
  case ${opt} in
  h)
    usage 0
    ;;
  o)
    output_path="$OPTARG"
    ;;
  p)
    ta_path="$OPTARG"
    ;;
  :)
    echo "Option -$OPTARG requires an argument" >&2
    usage 1
    ;;
  ?)
    echo "Invalid option: -$OPTARG" >&2
    usage 1
    ;;
  esac
done

extract_shdr "$ta_path" "$output_path"
extract_hash "$ta_path" "$output_path"
extract_digest "$ta_path" "$output_path"
extract_bootstrap "$ta_path" "$output_path"
extract_elf "$ta_path" "$output_path"
```

Listing A.1: Script de bash para extraer las distintas partes de una TA de OP-TEE.

APÉNDICE B

Script para verificar una Trusted Application

```python
#!/usr/bin/env python3

"""
 * This program verifies an OPTEE TA.
 * Author: Josep Comes <jocosan6@inf.upv.es>
 * Date: 15/01/2024
"""

import argparse
import hashlib

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding, utils


def checkDigest(shdr_location: str, bootstrap_location: str, elf_location: str,
    hash_location: str) -> bytes:
    print("[!] Checking hash")
    sha256 = hashlib.sha256()
    with open(shdr_location, "rb") as shdr_file:
        sha256.update(shdr_file.read())

    with open(bootstrap_location, "rb") as shdr_bootstrap_file:
        sha256.update(shdr_bootstrap_file.read())

    with open(elf_location, "rb") as elf_file:
        sha256.update(elf_file.read())

    computed_hash = sha256.digest()
    with open(hash_location, "rb") as hash_file:
        expected_hash = hash_file.read()

    if expected_hash == computed_hash:
        print(f"[+] Hash is correct: {computed_hash.hex()}")
    else:
        print("[-] Invalid hash")

    return computed_hash


def signatureCheck(digest: bytes, path_to_pkey: str, signature_location: str) -> None:
    print("[!] Checking signature")
    with open(path_to_pkey, "rb") as key_file:
```

```
        private_key = serialization.load_pem_private_key(key_file.read(), password=None,
            backend=default_backend())
        public_key = private_key.public_key()

    with open(signature_location, "rb") as signature_file:
        signature = signature_file.read()

        try:
            public_key.verify(
                signature,
                digest,
                padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=len(digest)),
                utils.Prehashed(hashes.SHA256()),
            )
        except InvalidSignature:
            print("[-] Invalid signature")
        else:
            print(f"[+] Signature is correct: {signature.hex()}")


if "__main__" == __name__:
    parser = argparse.ArgumentParser()
    parser.add_argument("-s", "--shdr", help="shdr file path", type=str, required=True)
    parser.add_argument("-b", "--bootstrap", help="bootstrap file path", type=str,
        required=True)
    parser.add_argument("-e", "--elf", help="elf file path", type=str, required=True)
    parser.add_argument("--hash", help="hash file path", type=str, required=True)
    parser.add_argument("-p", "--public-key", help="public key file path", type=str,
        required=True)
    parser.add_argument("--signature", help="signature file path", type=str, required=True
        )
    args = parser.parse_args()
    digest = checkDigest(args.shdr, args.bootstrap, args.elf, args.hash)
    signatureCheck(digest, args.public_key, args.signature)
```

Listing B.1: Script de python para verificar una TA de OP-TEE.

APÉNDICE C

Formato del token de autenticación

Con el fin de asegurar el intercambio de tokens y garantizar la compatibilidad entre los trustlets, daemons y servicios involucrados en los procesos de enroll y autenticación, el formato `AuthToken` se describe en [1]. Este token se genera tras una llamada al comando `verify()`, utilizado cada vez que el usuario intente autenticarse. Este token contiene el SID de usuario y se asocia a todas las claves del almacén de claves vinculadas a la autenticación. Sin embargo, cabe mencionar que una llamada no confiable a la función `enroll()` cambiará el SID del usuario, lo que inutilizará las claves vinculadas a esa contraseña.

El formato de este token es un protocolo de serialización simple con campos de tamaño fijo que se muestran a continuación:

```
typedef struct __attribute__((__packed__)) {
    uint8_t version;
    uint64_t challenge;
    uint64_t user_id;
    uint64_t authenticator_id;
    uint32_t authenticator_type;
    uint64_t timestamp;
    uint8_t hmac[32];
} hw_auth_token_t;
```

Listing C.1: Estructura del token de autenticación.

Seguidamente, se describe cada uno de estos campos:

- **version**: Número de versión del token.
- **challenge**: Un número entero aleatorio para evitar ataques de repetición.
- **user_id**: Identificador de seguridad (SID) no repetitivo vinculado criptográficamente a todas las claves asociadas con la autenticación del dispositivo. Este campo no guarda relación con el ID de usuario de Android.
- **authenticator_id**: Se utiliza para indicar diferentes permisos de autenticación. Normalmente indica el ID de la base de datos de huellas dactilares.
- **timestamp**: Marca de tiempo del último arranque del sistema.
- **authenticator_type**: `0x00`: Gatekeeper; `0x01`: Fingerprint.
- **hmac**: MAC SHA-256 con clave de todos los campos excepto del campo `hmac`. Este campo protege la integridad del token.

[1] `https://cs.android.com/android/platform/superproject/main/+/main:hardware/libhardware/include_all/hardware/hw_auth_token.h;l=39?`

APÉNDICE D
Objetivos de Desarrollo Sostenible

El 25 de septiembre de 2015, fue aprobada por la Asamblea General de Naciones Unidas la Agenda 2030 de Desarrollo Sostenible. Esta agenda está compuesta por un conjunto de objetivos globales para erradicar la pobreza, proteger el planeta y asegurar la prosperidad para todos como parte de una nueva agenda de desarrollo sostenible. Cada objetivo tiene metas específicas que deben alcanzarse en los próximos 15 años.

Los Objetivos de Desarrollo Sostenibles (ODS), también conocidos como Objetivos Mundiales, quedarán fijados para la posteridad como el mayor esfuerzo organizado que la humanidad haya hecho para elevar el nivel y la calidad de vida de las personas en el mundo. Este esfuerzo ha proporcionado una estrategia única, una agenda colectiva que sociedades y gobiernos del planeta pueden seguir simultáneamente, sirviendo como una lista de prioridades que favorezcan el desarrollo de las naciones. Ahora, cuando ya se ha recorrido casi la mitad del trayecto propuesto para alcanzar los ODS, será fundamental el papel de la educación, de los gobiernos y de las empresas para poder lograrlos.

Los 17 ODS están integrados, ya que reconocen que las intervenciones en un área afectarán los resultados de otras y que el desarrollo debe equilibrar la sostenibilidad medioambiental, económica y social. A continuación, se presenta una tabla con los Objetivos de Desarrollo Sostenibles, indicando el grado de relación de cada uno con el trabajo realizado.

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No procede
ODS 1. **Fin de la pobreza**				**X**
ODS 2. **Hambre cero**				**X**
ODS 3. **Salud y bienestar**				**X**
ODS 4. **Educación de calidad**		**X**		
ODS 5. **Igualdad de género**				**X**
ODS 6. **Agua limpia y saneamiento**				**X**
ODS 7. **Energía asequible y no contaminante**				**X**
ODS 8. **Trabajo decente y crecimiento económico**			**X**	
ODS 9. **Industria, innovación e infraestructura**	**X**			
ODS 10. **Reducción de las desigualdades**				**X**
ODS 11. **Ciudades y comunidades sostenibles**				**X**
ODS 12. **Producción y consumos responsables**			**X**	
ODS 13. **Acción por el clima**				**X**
ODS 14. **Vida submarina**				**X**
ODS 15. **Vida de ecosistemas terrestres**				**X**
ODS 16. **Paz, justicia e instituciones**				**X**
ODS 17. **Alianzas para lograr objetivos**				**X**

Tabla D.1: Grado de relación del trabajo con los ODS.

De los anteriores objetivos de desarrollo sostenibles mencionados, el presente trabajo está relacionado con:

- **Educación de calidad**: Personalmente creo que la idea presentada promueve el desarrollo de contenidos por los que merece la pena pagar, esto es contenidos que tienen calidad y por los que una persona estaría dispuesta a pagar una cantidad de dinero porque le aporta valor. Además, la idea en la que se basa este trabajo ha sido ampliamente discutida por la comunidad hacker internacional; esto supone que haya multitud de documentación técnica de calidad, la cual hemos aprovechado en este trabajo.
- **Trabajo decente y crecimiento económico**: Este ODS puede estar relacionado con el trabajo si se trata desde el punto de vista de la ciberseguridad. Pienso que puede ayudar a proteger activos esenciales, mejorando el producto y por consiguiente un aumento en el crecimiento económico.
- **Industria, innovación e infraestructura**: A su vez también creo que este ODS está relacionado. Durante el trabajo, se estudian todos aquellos puntos que posteriormente son necesarios en la creación de la aplicación, por lo que este proceso también sirve para poder crear una aplicación más robusta o ampliar la aplicación desarrollada para cubrir otros casos de uso.
- **Producción y consumos responsables**: Al tratarse de un producto digital no contribuye al procesamiento medioambiental de materias primas.